Runtime Verification: Passing on the Baton

Christian Colombo[1][0000−0002−2844−5728], Gordon J. Pace[1], and Gerardo Schneider[2][0000−0003−0629−6853]

[1] University of Malta, Malta
[2] University of Gothenburg, Gothenburg, Sweden
{christian.colombo,gordon.pace}@um.edu.mt, gersch@chalmers.se

Abstract. Twenty years have passed since the first workshop on runtime verification — the area has grown and evolved with hundreds of papers published and a sizeable number of mature tools developed. In a special occasion like this it is good to look back, but it is also good to look forward to the future. In this paper, we outline a very brief history of runtime verification, and propose a way of passing the knowledge down to future generations of academics and industry practitioners in the form of a roadmap for teaching runtime verification. The proposal, based on our experience, not only equips students with the fundamental theory underpinning runtime verification, but also ensures they have the required skills to engineer it into industrial systems. Our hope is that this would increase uptake and eventually give rise to the establishment of industry-grade tools.

1 Introduction

The field of *runtime verification* (RV) is a relatively new research area within formal methods that complements classical static (and usually exhaustive) verification techniques such as static analysis, model checking and theorem proving, as well as dynamic techniques like testing. Runtime verification is a *dynamic* technique, i.e., it is used while running the system or to analyse real executions of the system. Runtime verification has a more practical approach than static techniques in the sense that analysing a *single* (as opposed to all) execution trace of a system (usually the current execution) is significantly more manageable.

Researchers in the area have taken different takes on the exact definition of runtime verification: sometimes it is also referred to as runtime monitoring, trace analysis, dynamic analysis, etc. That said, the idea of using *verification* in its name implies a notion of correctness with respect to some predefined property.

Despite the variations in the runtime verification community regarding terminology and scope, there are at least two things researchers in the area agree on. First, the runtime verification field (as a community) grew out of the mainstream formal verification one starting from the RV workshop established in 2001[3] which was organised by Klaus Havelund and Grigore Rosu.[4] In fact, the

[3] http://www.runtime-verification.org/rv2001.

[4] The workshop was held for many years till it became a conference in 2010 [BFF+10]; the conference has held every year since then.

term *runtime verification* was first introduced as the name of that workshop. Second, a key person responsible for making the area what it is today was Klaus Havelund[5] and indeed, Havelund presented a number of papers (co-authored with Rosu and Giannakopoulou) in 2001 on the new, and still unnamed, area of runtime verification [HR01a,HR01b,GH01].

In this paper, written for Klaus Havelund's 65-year *Festschrift*, we focus on how the *RV baton* has been passed since the beginning of the area, via successive scientific results, and how we can pass it on to future generations via teaching. We provide a very brief overview of some historical facts on RV, and present a roadmap for teaching runtime verification. One of the main reasons for choosing this as a topic for such an occasion is that Havelund has, since the very beginning of his research in runtime verification, been interested in promoting the area via numerous talks at conferences and tutorials in summer and winter schools.

Despite the fact that runtime verification has frequently been touted as being industry-friendly, and to have a low barrier to entry, uptake in industrial projects is surprisingly rare. And although one finds many features of runtime verification used in the industry, whether it is in the form of assertions or that of behavioural sanity checks, what is rare is the adoption of the whole philosophy of separation-of-concerns between specification and implementation. From our experience, the more sophisticated specification techniques, e.g., state transition models, are mostly used as ex ante *drivers* for design and development, and not as ex post *oracles* for ensuring correctness. One of the most positive outcomes of projects with industry partners we have had was that aspects of the notion of monitoring of complex properties and specifications spilled over into in-house tools and internal procedures — either in the form of sanity checking executed on data collected in log files, or in the form of internal logic to ensure the well-formedness of runtime behaviour of objects.

There are different paths which can be taken when designing a course on runtime verification. One can go for a primary focus on the theoretical foundations of the discipline or one can take a more pragmatic, hands-on approach and dedicate a substantial part of the course on exposing tools and their design to the students. In an ideal world, one would cover both bases, however, with limited time one has to determine a primary (if not sole) focus. With the discipline being rooted in formal methods and elements of theoretical computer science, we have noted that often the focus many take when lecturing runtime verification is primarily a theoretical one, with a practical element using a (usually in-house) runtime verification tool to illustrate the concepts. Alas, given that there are no standard runtime verification tools yet which the industry has adopted, and given that academic tools are, by their very nature, typically not sufficiently dependable to be used in the real-world, we found that the content of the course rarely spilled into the students' working life after completing their degree programme.

In this context, going beyond the use of an existing tool and providing the students with deeper understanding of design issues arising when building a

[5] https://www.havelund.com/.

runtime verification tool, has the foreseen advantage that ideas and principles from the field of runtime verification can slowly permeate into industrial projects through the development of in-house libraries and tools. This is the approach we advocate in this paper.

This paper is organised as follows: We first provide a high-level introduction to RV including a brief history of the area in Section 2. Next, in Section 3, we outline the main topics of a proposed hands-on course on runtime verification. This is followed by an account on our experience of teaching various instances of the course in Section 4 and we conclude in Section 5.

2 Runtime Verification: A Very Brief Introduction and History

This section is divided in three subsections. We first discuss what is runtime verification and its connection with monitoring. The second part is concerned with the concept of monitorability from a historical perspective. We finally give a very brief historical overview of the development of the area.

2.1 On Runtime Verification and Monitoring

The term *runtime verification* appears to have been used for the first time in 2001 as the name of a workshop, *RV'01 — First Workshop on Runtime Verification*, organised by Klaus Havelund and Grigore Rosu. According to Havelund et al. [HRR19]: *"[our] initial interest in RV started around 2000 [...] Our initial efforts were inspired by Doron Drusinky's Temporal Rover system [30] for monitoring temporal logic properties, and by the company Compaq's work on predictive data race and deadlock detection algorithms [36]."*[6]

But what is runtime verification? There are many slightly different definitions in the literature and researchers use one or the other depending on their own personal views or on the aspect they want to emphasise. For our purposes we take here the definition given by Leucker [Leu11]:

> *Runtime verification* is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny (SUS) satisfies or violates a given correctness property. Its distinguishing research effort lies in synthesising monitors from high level specifications.

Given a crucial component in runtime verification is the concept of *monitoring*, one may wonder what the big deal is about this new area; after all monitoring techniques have been around for a very long time and used before 2000 — if not by the formal methods community, then by computer scientists in other research communities and by practitioners. Indeed, we are all familiar

[6] In this paper, reference [30] appears as [Dru00], and reference [36] as [Har00].

with *monitoring programs* we execute in our computers to let us know which processes are running and how much CPU they are using, etc.

Even though monitoring is indeed a very central element in runtime verification, we may say that runtime verification goes beyond simply observing a system's behaviour. Is thus "monitoring" a part of runtime verification, or *vice versa*? You can see it in both ways depending on your own choice of use of terms. If monitoring is seen as a generic term for having a program that interacts (runs in parallel) with the SUS, then runtime verification could be considered a particular case where such "interaction" is determined by observing events and flagging whether a property (encapsulated in the monitor) is satisfied or violated. On the other hand, runtime verification may be seen as a general area of which monitoring is a specific subarea (a particular case where the monitor just observes and accepts all executions of the SUS).

Different runtime verification researchers have (historically) used different terms for different things in different ways, and although taxonomies have been proposed e.g. [FKRT18], one still does not find uniform use in the literature. For the sake of this paper, we distinguish between the following different kinds of monitoring activity:

Observers are monitors that collect information from the SUS. This collection (logging events) might be done for different purposes, most of the time for *a posteriori* analysis to: perform statistics, identify faults, identify security breaches, assign blame (e.g., for legal reasons), etc.

Verifiers are monitors that encapsulate a verification property. Verifiers may be written to detect satisfaction or violations of the given property.

Reaction injectors handle the introduction of additional logic into the SUS at points when the verifier matches a specification. This has been used in a variety of ways in the literature, which can be categorised broadly as follows:

- **Enforcers** are reactions intended to enforce a given property. They usually act by intercepting events observed, identifying when a violation might occur, and modifying them (or more rarely modifying the system state) so the property is not violated.
- **Controllers** are reactions directly affecting the SUS meant to introduce new behaviour. Depending on observed events, the reaction logic would ensure that the SUS behaves in specific ways.[7] This approach includes monitoring-oriented programming in which the controller may introduce completely new features to the SUS.
- **Reparators** are reactions used for when the system has already violated a given property so, for instance, noting unexpected behaviour by a user, the reparators may disable that user in order to allow for further investigation before proceeding. In a way, reparations can be seen as enforcers running late (or enforcers as reparations running early), but

[7] One may prefer the term *actuators* instead in order to not confuse it with the controllers as used in the controller synthesis research area (they have similar purpose but we are not here concerned with synthesising controllers that are correct for *all* executions of the SUS).

there still lies a difference in that the former tries to fix what already went wrong, while the latter redirects behaviour to avoid what is still to happen.

Compensators are a specific form of reparation in which the additional logic will keep track of recent behaviour of the SUS so that, if a violation occurs, recent behaviour can be undone in order to restore the SUS to a sane state. For instance, if a financial transaction is decomposed into smaller sub-transactions, if a later one fails, violating the property, the effect of the earlier ones is undone.

The prevalent view in the runtime verification community is that monitors and verification logic should not be programmed directly, but should rather be extracted from (semi-) formal descriptions. This is even the case for observers, as we might be interested in events in a particular context, or ones which happen when a particular property is satisfied.[8]

So, in runtime verification we usually follow the steps below:

1. Choosing a suitable (formal) specification language;
2. Specifying (un)desired system behaviour;
3. Generating an event-listening monitor from the specification;
4. Synthesising the event verification algorithm;
5. Connecting a monitor to the SUS;
6. Analysing the verdict of the monitor, if suitable;
7. Triggering a reaction of a fitting kind, if applicable.

2.2 On the Concept of Monitorability

Another, related but somehow orthogonal aspect, is the question: What can(not) be monitored? This plays the same role as *decidability* in more standard formal verification. In the runtime verification community, this is called *monitorability*.

The first definition of monitorability was given by Kim et al. [KKL$^+$02], where the objective of monitoring was set only to detect violations of safety properties over infinite executions.

Pnueli and Zaks [PZ06] generalised the notion of monitorability, for both acceptance and violations of LTL formulae. The idea is that the purpose of a monitor is to determine whether every (possibly infinite) continuation of the current execution is a model or a counter-example of the monitored property: A monitor should thus give a definite verdict as soon as one is possible.

Additional results concerning monitorability were later presented by Bauer et al. [BLS07,BLS11], where it was shown that the set of monitorable LTL properties is a superset of the union of safety and co-safety properties, and Falcone et al. showed that it is a superset of the set of obligation properties [FFM09,FFM12].

More recently, Havelund and Peled [HP18] introduced a finer-grained taxonomy of monitorable properties, among other things distinguishing between

[8] There are exceptions though. For instance, if the observer monitor just logs *all* events in the execution trace, there is no need to write any formal specification.

always finitely satisfiable (or refutable), and *sometimes* finitely satisfiable where only some prefixes are required to be monitorable (for satisfaction). They also presented a connection between monitorability and classical safety properties.

Monitorability for hyperproperties, i.e., properties over sets of sets of traces, in particular for HyperLTL [CFK$^+$14], was first studied in [AB16]. This was later generalised to the full fragment of alternation-free formulas using formula rewriting [BSB17], which can also monitor alternating formulas but only with respect to a fixed finite set of finite traces.

In more recent work, Stucki et al. [SSSB19] presented richer definitions of monitorability for both trace- and hyper-properties, including the possibility of making use of information from the system being monitored (an approach they call *grey-box monitoring*). Gray-box monitoring combines static analysis/verification and runtime verification, enabling the monitoring of properties that are non-monitorable in a black-box manner. This new taxonomy of monitorability generalises over previous definitions given in the literature, including that of Havelund and Peled mentioned above, and it also covers the computability limitations of monitors as programs, not expressed in previous definitions.

2.3 A Very Brief History of Runtime Verification

Now that we have given some definitions (and intuition) about runtime verification and discussed the notions of monitorability, we give a very brief (non-exhaustive and without much detail) synthetic account on some historical developments in runtime verification.

What can be considered to be the first runtime verification paper appeared in the RV'01 workshop as mentioned above. Havelund and Rosu [HR01a] presented a general framework for analysing execution traces based on LTL specifications (only with future operators first, extended with past later), embodied in the Java PathExplorer, JPaX. The limitations of the framework, most notably the lack of data handling, were addressed by Havelund and other colleagues in the years that followed. Two such extensions appeared in 2003–2004: MOP [CR03] and EAGLE [BGHS04], supporting the monitoring of events with data, and allowing user-defined temporal operators (using a logic similar to linear time μ-calculus). One year later, the HAWK system was developed, described by the authors as *"an attempt to tie EAGLE to the monitoring of Java programs with automated code instrumentation using aspect-oriented programming, specifically AspectJ"*.

The idea of using four-valued semantics (introduced in [Dru00]) was used by Leucker and colleagues in the context of runtime verification in 2007 [BLS07]. The main motivation was that models of classical temporal logic formulae are usually infinite streams of events, whereas at runtime, we are mostly concerned with finite (prefixes of) traces, and the partial observation needs to give inconclusive verdicts. That paper introduced a four-valued semantics for LTL over finite traces extending the classical LTL semantics.

In 2008, Havelund et al. investigated the use of monitors *as* aspects [HRR19]: *"In previous solutions (such as HAWK and MOP) we have seen monitors translated* to *aspects. A more radical approach is to take the view that monitors* are

aspects. [...] An (unfinished) attempt in this direction was XspeC [50], designed to be an extension of ACC (an aspect-oriented programming framework for C) with data parameterised monitoring using state machines."[9] This idea is echoed in the proposition of an AspectJ extension enabling the detection of race conditions at runtime [BH10].

The same year, Chen et al. [CSR08] developed a *predictive* runtime analysis technique to "predict" concurrency errors that did not happen in the observed execution but which could have happened under a different thread scheduling. The approach was implemented in the tool jPredictor for Java.

In two subsequent articles published in 2008 and 2009 Colombo et al. proposed an automata-based approach to runtime verification, based on symbolic automata with stopwatches (the specification language was called DATE: Dynamic Automata with Timers and Events) [CPS08,CPS09a]. The approach was implemented into the LARVA tool [CPS09b] targeting Java programs (using AspectJ for the instrumentation).

In 2012 Zhang et al. provided a predictive semantics for runtime verification monitors [ZLD12], which was used later in 2016 by Pinisetty et al. to produce a predictive enforcement mechanism: by using *a priori* knowledge of the system some events could be produced faster in order to allow for earlier verdicts [PPT+16].

Another automaton-based runtime verification approach was proposed in 2012 by Barringer et al. [BFH+12]: Quantified Event Automata (QEA) are automata supporting quantification over data.

Runtime verification has been applied to one of the recent trends in verification and security: Hyperproperties. In 2017 Finkbeiner et al. [FHST17] proposed an automaton-based algorithm for monitoring HyperLTL, also for formulas with quantifier alternation, but for a fixed trace set. The complexity of monitoring different fragments of HyperLTL was studied in detail in [BF18].

One of the areas still in development is the combination of runtime verification with other validation and verification techniques. Bodden et al.'s pioneering work in the combination of static and dynamic analysis was embodied in their tool Clara [BL10], followed up by other such as Chimento et al.'s approach to combine control- and data-oriented properties for deductive analysis and runtime verification [CAPS15,ACPS17] and Azzopardi et al.'s work to address static analysis with symbolic state [ACP17].

In recent years, runtime verification has also been combined with testing in different ways e.g. [CAS18,CCF+16,DLT13]. For other (later) developments (log analysis, DSLs, etc), see the accounts in [HR18,HRR19], and for the use of runtime verification beyond software see [SSA+19].

Since 2014, besides continuing with the RV yearly conference, the community has also been coordinating an International Competition on Runtime Verification (CRV) [BFB+19], with the aim to compare and evaluate runtime verification tools. During the period December 2014 till December 2018, the European scientific network for the COoperation in Science and Technology (COST) on

[9] Reference [50] in the quote appears as [HW08] here.

Runtime Verification beyond Monitoring (ARVI) was approved, funded within the European Horizon 2020 framework programme.[10] ARVI included scientists from 26 European countries and Australia. Besides workshops and meetings, the ARVI COST Action organised a series of Graduate Schools on runtime verification, where top researchers in runtime verification, including Klaus Havelund, have given lectures and tutorials.

See Havelund's two retrospective papers [HR18,HRR19] on his own (and his co-authors') account on the historical context and research development based on his own contributions to runtime verification. For a taxonomy of runtime verification see, for instance [Leu11,FKRT18].

In the rest of the paper we present a roadmap to instruct new (under)graduate students in computer science in this exciting and still growing research area, which starts to be noticed by industry as an alternative and complement to testing, and other formal verification techniques.

3 A Hands-On Runtime Verification Course

In this section we outline the narrative of a course on runtime verification, focusing on ensuring that the students are exposed to relevant practical issues going beyond the use of existing tools to the actual development of one. Through the chain of concepts presented, we mention how salient issues in runtime verification can be communicated to the students as they build their tools.

3.1 Introduction to Runtime Verification

The course is proposed to start with an introduction to runtime verification, including the theoretical justification and terminology. The course can be motivated through the need for verification of systems, and a runtime approach justified by highlighting to the students the difference in complexity between the language inclusion and the word inclusion problems. The choice of a three-valued representation of verification outcome can be thus justified and explained.

Furthermore, students are exposed to the complexity of runtime checking of different types of properties e.g. showing how a property such as *"Function* recordTemperature() *should never be called with a value greater than 50"* requires no additional monitoring memory, whilst a property such as *"Function* recordTemperature() *can only be called if* initialiseThermometer() *was called earlier"* requires one bit of memory (recording whether the initialisation function was called), and a property such as *"The difference between the two most common parameter values passed to function* recordTemperature() *will never exceed 20"* requires keeping a histogram of parameters passed to the function, which may grow linearly with respect to the length of the history, thus effectively resulting in a memory leak.

On the basis of this view of runtime verification, one can then look at the architecture of the runtime verified system, highlighting how the event observers,

[10] https://www.cost.eu/actions/IC1402

verification algorithms and reaction code interact with the underlying SUS. Students can thus be exposed to different instrumentation approaches, from inlined to separation of specification and implementation concerns. Finally, drawing from the discussion of complexity which came earlier, one can motivate different verification architectures — online vs. offline, synchronous vs. asynchronous, etc.

After such an introduction, students will be ready to be exposed to the practical considerations of runtime verification.

3.2 Introduction to Monitoring through Assertions

Throughout the unit, we propose the adoption of a single SUS — one which is not trivial, yet tractable to understand by the students. The choice and design of such a system should ensure that it contains sufficient complexity to justify the verification techniques and specification languages presented in the course. Students would find the use of a logic such as LTL to be overkill (thus perceived as impractical) if applied to a simple input-output use case, so a more sophisticated business case is required in order to motivate the need of certain sophisticated techniques used in runtime verification.

In this paper, we will illustrate our proposed approach by using examples from a financial transaction system, based on one which two of the authors have used (see Section 4). However, this is done to illustrate our proposed approach and, needless to say, other real-world examples serve equally well.

To make it easy for students to latch on to runtime verification, we are proposing to start with its simplest form of monitoring and sanity checking of a systems — the use of assertions, a technique with which students but also software developers and testers are already familiar. Such an assertion is useful to check a particular condition and raise a labelled flag if found to be false.

A simple assertion might involve checking that a user is not from a particular country or region before being granted *Gold User* status. This is an example of a point property[11] which does not involve any references to other states of the system's timeline. By having the students analyse the code and identify the place where a user's status is being lifted to *Gold*, and adding an assertion to check for the user's country, exposes them to the use of assertions in runtime sanity checking of the system state.

The aim of such simple examples is to highlight the ease-of-use of assertions. However, through the introduction of more complex properties, the students quickly realise that the additional code to store information about the context (e.g. execution history) becomes an additional burden on the SUS. Properties involving sequences of events e.g. *"No user should log into the system before it is initialised"* highlights the difference arising from the fact that checking such a property requires linking two points in the code — the initialisation point (where one would have to set a flag), and the login point (where one would have to check for the status of the flag). With even more complex properties

[11] *Point* as in being possible to verify by checking for a particular condition at a particular point in the control logic of the system.

such as *"The administrator must reconcile accounts every 1000 attempted external money transfers or whenever an aggregate total of one million dollars in attempted external transfers is reached"*, students are exposed to the limitations of such a manual approach to monitoring, and that it is far from the ideal way of runtime checking a system due to:

Non-modular code: Students need to locate several points in the code and add variable declarations which are accessible from such points. This brings with it all the disadvantages of poor modularity, including a maintenance nightmare.

Error-prone code: It is easy to make mistakes in the code which is meant to check the system, hence defeating the whole purpose. A substantial part of this issue is due to the non-modularity mentioned in the previous point. However, it is also because the monitor is expressed at the same level of abstraction as the system itself.

3.3 Instrumenting Monitors

In order to address the issue of non-modularity of runtime verification code, students are then introduced to aspect-oriented programming (AOP).[12] This provides a modular way of programming cross-cutting concerns, i.e., aspects of a system which span other modules. Similar to event logging (another main application of AOP), runtime verification interacts with virtually all the modules of a system — making it an ideal application of AOP. At the same time, having been exposed to the spaghetti code resulting from merging the monitoring and verification code in the SUS, the students understand the need for tool support.

After providing basic AOP syntax, students can modify all their monitoring checks and verification code (including variable declarations for additional monitoring state) into a single aspect file independent of the SUS, but which can be compiled in order to automatically combine the two together. In this manner, students are exposed to the double benefit of (i) having all runtime verification-related code in one place, while also (ii) avoiding mistakes resulting from manual instrumentation.

At this point in the course, the students will have been exposed to tools to separate the SUS from the event monitoring, verification and reaction code. However, the latter part remains a combination of the three concerns mentioned, which we will now seek to separate.

3.4 Verification Algorithms

With the modularity attained through the use of AOP tools, the focus can now be turned to the separation between the monitoring, verification and reaction code. Students will be guided to start building *ad hoc* verification code for particular properties with increasing complexity, mostly to highlight how this results

[12] In a course using Java one can adopt a tool such as AspectJ.

in an increase in monitoring state and complexity of the monitoring code not resolved through the use of AOP. Starting with a simple point property e.g. *"Only approved users may be given gold status"*, we proceed to simple temporal properties e.g. *"The approval of a user will always fail unless their profile was previously verified"* and more complex ones e.g. *"The user cannot be reported for more than three times without being disabled"*. Finally, we add more complex elements including real-time constraints e.g. *"If a user performs transactions of more than €5000 in total in any 24 hour window without having undergone due diligence checks in the previous month, he or she must be blacklisted within 2 hours"*.

It quickly becomes evident that the level of complexity of the verification code reaches a stage where bugs could easily arise, subverting the advantages of verification in the first place. This leads to the use of specification languages and automated compilation of verification algorithms for them, thus abstracting the verification code up to a logic specification.

In order to build the complexity of verification code in a stepwise manner, we present the student with a sequence of abstraction levels, each of which simplifies the expression of the specification at the cost of the use of a more sophisticated and complex specification language:

1. We propose to start with a simple rule based language e.g. a guarded command language with rules of the form $\mathit{event} \mid \mathit{condition} \mapsto \mathit{action}$ which provides only a thin layer above the AOP code — events correspond to pointcuts, while the condition and action simply correspond to a standard conditional in the underlying programming language. In this manner, we start to abstract away from the AOP tool syntax, specialising to a specification language. However, clearly this can be shown to yield little gain in terms of specification abstraction.
2. We then move on to the runtime verification of safety properties on finite traces [HR02]. One way of expressing sequentiality properties is to use a graph-based formalism, an approach that students would typically already be familiar with from formal languages and automata courses. We propose to start with deterministic automata with transitions labelled by guarded commands, and show how they can be used to express properties which use a notion of temporal sequentiality. The students are given the task of building a translator from such automata into the underlying guarded command language. The exercise highlights the fact that new state is introduced at the lower level (keeping track of the state of the automaton), which is implicit in the abstract specification. This brings across the message that most abstract specification languages allow implicit encoding of states.
   Furthermore, the students are asked to consider adding non-determinism, and quickly run into a myriad of dead ends as they solve one problem only to encounter another. Results they would already be familiar with, showing the equivalence of the expressive power of deterministic and non-deterministic automata lulls them into a fake sense of ease, only to discover they cannot simply perform the transformation they are familiar with due to the actions

resulting in transitions having a side effect and that multiple runs of the SUS cannot be kept simultaneously. We feel that such an exercise gives the students a better understanding for the need of determinism (or at least not having the monitor interfere with the SUS) than any theoretical results would. In addition, the power of the symbolic state of the automata (any monitoring information maintained by the actions) is highlighted.

3. Moving away from graph-based formalisms we propose another formalism with which the students are familiar — regular expressions. Using regular expressions to write properties [SELS05], students can also be exposed to different interpretations of a single specification language — do we write a regular expression to express bad behaviour e.g. writing $(\mathit{login} \cdot \overline{\mathit{logout}}^* \cdot \mathit{logout})^* \cdot \overline{\mathit{login}}^* \cdot \mathit{read}$ to express the property that no reads should be done outside of a login session[13] or to express the good (expected behaviour) e.g. $(\mathit{login} \cdot (\mathit{read} + \mathit{write})^* \cdot \mathit{logout})^*$ to express the same property as above.
   Students start by programming a translation from regular expressions to finite state automata to understand better the complexity of coding the translation, the blowup in states and difficulty in handling non-determinism in regular expressions. This is used to motivate the use of Brzozowski's derivatives [Brz64], which results in a much simpler implementation. In this manner, students will be exposed to the notions of derivatives on a formalism they are already familiar with, and get to program it themselves.
4. Finally, we move on to the use of Linear Temporal Logic (LTL) and present Havelund et al.'s derivative-based approach [HR01b] which is now readily accessible to the students. In order to tie the knot back to where we started off from, we also present Leucker et al.'s automaton-based approach to LTL monitoring [BLS11].

The bulk of the course, in fact lies in this part of the course. In programming the different solutions themselves, the students get to explore different possibilities and understand better the pros and cons of different approaches.

3.5 Real-time properties

Given the ubiquity of real-time concerns in systems which are worth monitoring, if time permits, a course on runtime verification should ideally touch upon the monitoring of real-time properties. Students should appreciate the challenges that this brings about, particularly due to the fact that monitoring itself modifies the timing of the underlying system [CPS09b]. We note that this issue is not limited to timing, but to any element of the system which is self-reflective, e.g., if the SUS considers its own level of memory consumption.

When dealing with real-time issues, there are two main kinds of properties: those which can be monitored through system events and those which require

[13] We use standard regular expression syntax here, with a indicating the occurrence of that event, $\overline{a}$ to indicate the occurrence of any event other than a, $e \cdot e'$ to indicate sequential composition of two sub-expressions, $e + e'$ to indicate the choice and e^* to indicate any number of repetitions.

additional timer-events to wake the monitor up at crucial points in time. The former are lower-bound properties, e.g., there should not be more than five login attempts in a five minute period, while the latter are upper-bound, e.g., for the account to remain active, there should at least be one successful login each month. Students learn that it suffices to hook existing system events, e.g., login attempts, to monitor lower-bound properties, but a timer would be required for upper-bound ones, e.g., reset after each successful login and going off when a month elapses. At this point, it is useful to introduce the notion of detecting bugs "as soon as possible". Technically, upper-bound properties can also be detected through system events, albeit much later than their occurrence: e.g., a flag is raised when the user attempts to login in six months after their last interaction.

3.6 Offline monitoring

Several works applying runtime verification to industry (see, for example, reports on work on real-world systems such as [BGHS09,CPA12,Hav15,CP18a,CP18b]) have resorted to offline monitoring for minimal intrusion — effectively rendering it equivalent to event logging. This reality suggests that a course on runtime verification including offline monitoring, would provide the students with useful, applicable, practical knowledge.

Students are made aware that unlike the online version, where the system method call can be used to obtain system state on demand, in the offline case, anything which is not stored is not available. Similarly, another limitation is that reparation actions cannot be taken by the monitor when it is completely detached from the system.

Switching from online to offline monitoring requires that traces are stored in monitor-consumable format. Usually this involves keeping track of the method calls, the arguments passed, and possibly return values. Any system state which is required by the monitor also needs to be stored, e.g., globally accessible values which are never passed as parameters. Next, one would have to choose how to store the trace, with the most universal approach being a text file. While this is highly convenient, when dealing with complex objects appropriate toString and parse methods might need to be added.

Once students are made aware of the repercussions of switching to offline monitoring, the aspect-oriented mechanism previously used for monitoring, is reduced to just event logging in a text file. Next comes the task of reconfiguring the monitor to consume events from the file rather than directly through AOP. With little modifications to the code, students are guided to appreciate how existing monitors can be connected to a log file parser (instead of the original system) through AOP: Each time the parser consumes a line from the log file, a dummy method is called which triggers the monitor.

3.7 Advanced Topics

We can never expect that a runtime verification course is exhaustive; several aspects would naturally be left out. However, to help students see the relevance

of the topic and connect it to their work, the instructor can try to latch on to what they are already working on and discuss advanced topics. For example, for a student working on the analysis of security and privacy properties, studying runtime verification for hyperproperties becomes an imperative. Students with a background in statistics can perhaps appreciate more the need for on-the-fly calculation when dealing with stream processing. Those focusing on concurrent and distributed systems can be helped to appreciate the different questions and new challenges not present when analysing sequential and monolithic programs. For students with a more formal background, one can discuss questions such as what is monitorability and its limits, how specifications can be inferred from execution traces, how static analysis can alleviate dynamic analysis, etc. Similar connections can be found with other areas such as smart contracts, testing, embedded devices, etc.

As a means of wrapping up the unit, having a session on current challenges in runtime verification exposes the students to the frontiers of the research area. In the past we have discussed selected topics on the lines of those appearing in [SSA+19], including transactional information systems, contracts and policies, security and sampling-based runtime verification.

4 Lecturing Experience

One of the main drivers behind the setting up of a heavily hands-on unit offered to computer science students was that of ensuring that the students can see the practical relevance of runtime verification techniques. By exposing them to the theory but also putting a big emphasis on the practical side of how runtime verification tools can be built, we hoped to see, after the students graduate and integrate in development teams, an increase in awareness and adoption in the industry opening new opportunities for research collaboration.

It is worth mentioning also other biases in our lecturing approach. At the time we started working on runtime verification, in 2005, most of the work focused on verifying system-wide properties and (slightly later) per-object specifications.[14] When we sought out industry partners with whom to work, in order to ensure the practical relevance of our results, we repeatedly encountered properties which were neither system nor object-based, but rather transaction-based. In fact, a single transaction could transcend multiple objects.[15] This led to our own specification language and tool Larva [CPS08] which allows the verification engineer to adjust the notion of verification unit identity for the specification at hand. Another major departure from the *status quo* at the time, was to adopt the use of symbolic automata for specifications. This was also partially motivated by

[14] It is worth highlighting that the authors initial work in runtime verification was done jointly. However, some of the discussion which follows, regarding ongoing collaborative projects and lecturing is specific to the University of Malta, although much of what we describe was discussed between all three authors at the time.

[15] The transaction object before serialisation and the object after serialisation should be treated as the same object based on a logical identifier.

our experience in the projects we had with industry partners: (i) the graphical representation of the properties were digestible for the software engineers (justifying the use of automata); and (ii) the fact that variable declaration, querying, and manipulation was supported, not only made the notation more expressive, but also felt familiar for the same engineers (hence the choice of making them symbolic).[16]

Based on our research and collaboration at that time, we thus designed and delivered units on runtime verification both at undergraduate and graduate level. Although, over the years, much changed in terms of content and organisation, the philosophy behind the approach and the general content remained largely unchanged. At the undergraduate level, runtime verification was covered as part of a verification techniques course which introduced students to the ideas of formal specifications and formal verification techniques with particular focus on runtime verification, model checking, and model-based testing. At a Masters level, however, our aim was to go deeper and train students to be runtime verification engineers, i.e., by the end of the unit, they are able to build their own tools. We do this by incrementally providing skeleton code of a basic runtime verification tool, until by the end of the unit, the tool supports a number of specification languages (as discussed in this paper) for which it generates AspectJ and Java code which readily monitors a Java system. This material[17] was covered in a unit consisting of 28 student contact hours including practical hands-on sessions.

The practicality of the course, as well as the empowerment the students feel by building their own tool, has led to positive feedback from the students. A cut-down version of the course has also been given in a number of other contexts: ECI Winter School in Computer Science in Buenos Aires[18] (2013), the RV summer school in Madrid[19] (2016), and the RV winter school in Pratz sur Arly[20] (2018).

When we designed the course, it was envisaged as a means for the ideas and principles from the field of runtime verification to slowly permeate into the industry through the development of in-house libraries and tools. Anecdotally, based on newer collaborative projects and interaction with former students, we believe that this has happened. Obviously, it is unclear how much of this was due to the infusion of students aware of runtime verification theory and techniques into the industry, and how much of it is due to changing industry practice.

5 Conclusions

On this special occasion of Klaus Havelund's 65-year *Festschrift*, we take the opportunity to look back to the inception of the area of runtime verification

[16] There were other advantages for these choices, such as the possibility of also manipulating the monitored system state (as a reaction to observations), and the seamless introduction of a stopwatch API. However, these aspects fall outside the scope of this paper.

[17] Anyone interested in using the material may get in touch with the authors.

[18] https://meals-project.eu/node/67

[19] https://rv2016.imag.fr/?page_id=128

[20] https://www.youtube.com/watch?v=Vyz6kte4PVk

and also to the future — the history is logged in the workshop and conference proceedings, and yet like a monitor ourselves, it is good to reflect on the past and react to influence the future.

It is in this spirit that we provide a brief overview of how runtime verification evolved in the past 20 years, leading up to a substantial number of academically-mature tools by a wide community of researchers. At the current juncture of the story, we feel that the time is ripe to push for more practical use of the techniques, particularly in industrial settings. Thus, we propose a roadmap for teaching runtime verification to upcoming generations, leading them to become hands-on runtime verification engineers. Several instances of this roadmap have been delivered over the past years and the outcome and feedback have been encouraging.

For the future, as software continues to increasingly play a crucial part in the global life of humanity and runtime verification becomes ever more relevant in its promise of safety, dynamism, and immediacy, we hope that as a community we rise up to the occasion and make better known the theories and techniques which have been developed and honed in through years of hard work.

Acknowledgements. This research has been partially supported by the Swedish Research Council (*Vetenskapsrådet*) under Grant 2015-04154 "PolUser".

References

[AB16] Shreya Agrawal and Borzoo Bonakdarpour. Runtime Verification of k-Safety Hyperproperties in HyperLTL. In *CSF'16*, pages 239–252. IEEE CS Press, 2016.

[ACP17] Shaun Azzopardi, Christian Colombo, and Gordon J. Pace. Control-Flow Residual Analysis for Symbolic Automata. In *PrePost@iFM'17*, volume 254 of *EPTCS*, pages 29–43, 2017.

[ACPS17] Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. Verifying Data- and Control-Oriented Properties Combining Static and Runtime Verification: Theory and Tools. *Formal Methods in System Design*, 51(1):200–265, 2017.

[BF18] Borzoo Bonakdarpour and Bernd Finkbeiner. The Complexity of Monitoring Hyperproperties. In *CSF'18*, pages 162–174. IEEE CS Press, 2018.

[BFB+19] 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. *Int. J. Softw. Tools Technol. Transf.*, 21(1):31–70, 2019.

[BFF+10] Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors. *Runtime Verification - First International Conference (RV'10)*, volume 6418 of *LNCS*. Springer, 2010.

[BFH+12] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified Event Automata: Towards Expressive and

Efficient Runtime Monitors. In *FM'12*, volume 7436 of *LNCS*, pages 68–84. Springer, 2012.

[BGHS04] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-Based Runtime Verification. In *VMCAI'04*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.

[BGHS09] Howard Barringer, Alex Groce, Klaus Havelund, and Margaret H. Smith. An Entry Point for Formal Methods: Specification and Analysis of Event Logs. In *FMA'09*, volume 20 of *EPTCS*, pages 16–21, 2009.

[BH10] Eric Bodden and Klaus Havelund. Aspect-oriented race detection in java. *IEEE Transactions on Software Engineering*, 36(4):509–527, 2010.

[BL10] Eric Bodden and Patrick Lam. Clara: Partially Evaluating Runtime Monitors at Compile Time - Tutorial Supplement. In *RV'10*, volume 6418 of *LNCS*, pages 74–88. Springer, 2010.

[BLS07] Andreas Bauer, Martin Leucker, and Christian Schallhart. The Good, the Bad, and the Ugly, But How Ugly Is Ugly? In *RV'07*, volume 4839 of *LNCS*, pages 126–138. Springer, 2007.

[BLS11] Andreas Bauer, Martin Leucker, and Chrisitan Schallhart. Runtime Verification for LTL and TLTL. *ACM T. Softw. Eng. Meth.*, 20(4):14, 2011.

[Brz64] Janusz A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[BSB17] Noel Brett, Umair Siddique, and Borzoo Bonakdarpour. Rewriting-Based Runtime Verification for Alternation-Free HyperLTL. In *TACAS'17*, volume 10206 of *LNCS*, pages 77–93. Springer, 2017.

[CAPS15] Jesús Mauricio Chimento, Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider. StaRVOOrS: A Tool for Combined Static and Runtime Verification of Java. In *RV'15*, volume 9333 of *LNCS*, pages 297–305. Springer, 2015.

[CAS18] Jesús Mauricio Chimento, Wolfgang Ahrendt, and Gerardo Schneider. Testing Meets Static and Runtime Verification. In *FormaliSE'18*, pages 30–39. ACM, 2018.

[CCF+16] Abigail Cauchi, Christian Colombo, Adrian Francalanza, Mark Micallef, and Gordon J. Pace. Using Gherkin to Extract Tests and Monitors for Safer Medical Device Interaction Design. In *ACM SIGCHI EICS'16*, pages 275–280. ACM, 2016.

[CFK+14] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal Logics for Hyperproperties. In *POST'14*, volume 8414 of *LNCS*, pages 265–284. Springer, 2014.

[CP18a] Christian Colombo and Gordon J. Pace. Considering Academia-Industry Projects Meta-characteristics in Runtime Verification Design. In *ISoLA'18*, volume 11247 of *LNCS*, pages 32–41. Springer, 2018.

[CP18b] Christian Colombo and Gordon J. Pace. Industrial Experiences with Runtime Verification of Financial Transaction Systems: Lessons Learnt and Standing Challenges. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 211–232. Springer, 2018.

[CPA12] Christian Colombo, Gordon J. Pace, and Patrick Abela. Safer Asynchronous Runtime Monitoring using Compensations. *Formal Methods in System Design*, 41(3):269–294, 2012.

[CPS08] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties. In *FMICS'08*, volume 5596 of *LNCS*, pages 135–149. Springer, 2008.

[CPS09a] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper). In *SEFM'09*, pages 33–37. IEEE Computer Society, 2009.

[CPS09b] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Safe Runtime Verification of Real-Time Properties. In *FORMATS'09*, pages 103–117, 2009.

[CR03] Feng Chen and Grigore Rosu. Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. In *RV'03*, volume 89, pages 108–127. ENTCS, 2003.

[CSR08] Feng Chen, Traian-Florin Serbanuta, and Grigore Rosu. jPredictor: A Predictive Runtime Analysis Tool for Gava. In *ICSE'08*, pages 221–230. ACM, 2008.

[DLT13] Normann Decker, Martin Leucker, and Daniel Thoma. jUnit$^{\mathrm{RV}}$-Adding Runtime Verification to jUnit. In *NASA FM'13*, volume 7871 of *LNCS*, pages 459–464. Springer, 2013.

[Dru00] Doron Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN'00*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.

[FFM09] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime Verification of Safety-Progress Properties. In *RV'09*, volume 5779 of *LNCS*, pages 40–59. Springer, 2009.

[FFM12] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What Can You Verify and Enforce at Runtime? *International Journal on Software Tools for Technology Transfer (STTT)*, 14(3):349–382, 2012.

[FHST17] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. Monitoring Hyperproperties. In *RV'17*, volume 10548 of *LNCS*, pages 190–207. Springer, 2017.

[FKRT18] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. A Taxonomy for Classifying Runtime Verification Tools. In *RV'18*, volume 11237 of *LNCS*, pages 241–262. Springer, 2018.

[GH01] Dimitra Giannakopoulou and Klaus Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *ASE'01*, pages 412–416. IEEE Computer Society, 2001.

[Har00] Jerry J. Harrow. Runtime Checking of Multithreaded Applications with Visual Threads. In *SPIN'00*, volume 1885 of *LNCS*, pages 331–342. Springer, 2000.

[Hav15] Klaus Havelund. Rule-Based Runtime Verification Revisited. *Int. J. Softw. Tools Technol. Transf.*, 17(2):143–170, 2015.

[HP18] Klaus Havelund and Doron Peled. Runtime Verification: From Propositional to First-Order Temporal Logic. In *RV'18*, volume 11237 of *LNCS*, pages 90–112. Springer, 2018.

[HR01a] Klaus Havelund and Grigore Rosu. Monitoring Java Programs with Java PathExplorer. *ENTCS*, 55(2):200–217, 2001.

[HR01b] Klaus Havelund and Grigore Rosu. Monitoring Programs Using Rewriting. In *ASE'01*, pages 135–143, 2001.

[HR02] Klaus Havelund and Grigore Rosu. Synthesizing Monitors for Safety Properties. In *TACAS'02*, pages 342–356, 2002.

[HR18] Klaus Havelund and Grigore Rosu. Runtime Verification - 17 Years Later. In *RV'18*, volume 11237 of *LNCS*, pages 3–17. Springer, 2018.

[HRR19] Klaus Havelund, Giles Reger, and Grigore Rosu. Runtime Verification Past Experiences and Future Projections. In *Computing and Software Science* -

State of the Art and Perspectives, volume 10000 of *LNCS*, pages 532–562. Springer, 2019.

[HW08] Klaus Havelund and Eric Van Wyk. Aspect-Oriented Monitoring of C Programs. In *IARP-IEEE/RAS-EURON'08*, 2008.

[KKL+02] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Computational Analysis of Run-time Monitoring - Fundamentals of Java-MaC. In *RV'02*, volume 70 of *ENTCS*, pages 80–94, 2002.

[Leu11] Martin Leucker. Teaching Runtime Verification. In *RV'11*, volume 7186 of *LNCS*, pages 34–48. Springer, 2011.

[PPT+16] Srinivas Pinisetty, Viorel Preoteasa, Stavros Tripakis, Thierry Jéron, Yliès Falcone, and Hervé Marchand. Predictive Runtime Enforcement. In *ACM SAC'16*, pages 1628–1633. ACM, 2016.

[PZ06] Amir Pnueli and Aleksandr Zaks. PSL Model Checking and Run-time Verification via Testers. In *FM'06*, volume 4085 of *LNCS*, pages 573–586. Springer, 2006.

[SELS05] Usa Sammapun, Arvind Easwaran, Insup Lee, and Oleg Sokolsky. Simulation of Simultaneous Events in Regular Expressions for Run-Time Verification. In *RV'04*, volume 113 of *ENTCS*, pages 123–143, 2005.

[SSA+19] César Sánchez, Gerardo Schneider, Wolfgang Ahrendt, Ezio Bartocci, Domenico Bianculli, Christian Colombo, Yliès Falcone, Adrian Francalanza, Srdan Krstic, João M. Lourenço, Dejan Nickovic, Gordon J. Pace, José Rufino, Julien Signoles, Dmitriy Traytel, and Alexander Weiss. A Survey of Challenges for Runtime Verification from Advanced Application Domains (Beyond Software). *Formal Methods in System Design*, 54(3):279–335, 2019.

[SSSB19] Sandro Stucki, César Sánchez, Gerardo Schneider, and Borzoo Bonakdarpour. Gray-box Monitoring of Hyperproperties. In *FM'19*, volume 11800 of *LNCS*, pages 406–424. Springer, 2019.

[ZLD12] Xian Zhang, Martin Leucker, and Wei Dong. Runtime Verification with Predictive Semantics. In *NASA FM'12*, volume 7226 of *LNCS*, pages 418–432. Springer, 2012.