A Brief Account of Runtime Verification

Martin Leucker and Christian Schallhart

Technische Universität München and Technische Universität Darmstadt

Abstract

In this paper, a brief account of the field of runtime verification is given. Starting with a definition of runtime verification, a comparison to well-known verification techniques like model checking and testing is provided, and applications in which runtime verification brings out its distinguishing features are pointed out. Moreover, extensions of runtime verification such as monitor-oriented programming, and monitor-based runtime reflection are sketched and their similarities and differences are discussed. Finally, the use of runtime verification for contract enforcement is briefly pointed out.

1 Introduction

Software and software systems are increasingly ubiquitous in everyday life. Besides traditional applications such as word processors or spreadsheets running on workstations, software is an important part of consumer devices such as mobile phones or digital cameras, and functions as embedded control devices in cars or in power plants. Especially in such embedded application domains, is essential to guarantee that the deployed software works in a correct, secure, and reliable manner, as life may depend on it. For example, the software within a car's anti-skid system must speed with exactly the right velocity to stabilize the car. Moreover, for a power plant it is important that no intruder gets control over the plant and that it works also in case of a partial break-down of some of its parts.

Software engineering has been driven as a field by the struggle for guaranteed quality properties ever since, but nowadays and especially in the embedded domain, legislation and certification authorities are requiring proof of the most critical software properties in terms of a documented verification process.

In recent years, the idea of *software as services* has added a new paradigm to the architecture of software systems: They are seen more and more as autonomous agents acting according to certain *contracts*. For such systems,

verification is particularly challenging as the overall behavior of such systems depends heavily on the involved agents, which renders the analysis of such systems prior to execution next to impossible. One important aspect of verification is then to check whether each party acts according to the contract they have agreed on.

Traditionally, one considers three main verification techniques: *theorem proving* [12], *model checking* [20], and *testing* [49,16]. Theorem proving, which is mostly applied manually, allows to show correctness of programs similarly as a proof in mathematics shows correctness of a theorem. Model checking, which is an automatic verification technique, is mainly applicable to finite-state systems. Testing covers a wide field of diverse, often ad-hoc, and incomplete methods for showing correctness, or, more precisely, for finding bugs.

These techniques are subject to a number of forces imposed by the software to build and the development process followed, and provide different trade-offs between them. For example, some require a formal model, like model checking, give stronger or weaker confidence, like theorem proving over testing, or are graceful in case of error handling.

Runtime verification is being pursued as a *lightweight* verification technique complementing verification techniques such as model checking and testing and establishes another trade-off point between these forces. One of the main distinguishing features of runtime verification is due to its nature of being performed at runtime, which opens up the possibility to *act* whenever incorrect behavior of a software system is detected.

The aim of this paper is to give a brief account of runtime verification. Therefore, we first highlight basic ideas of runtime verification and discuss distinguishing features in comparison to other verification techniques. Then, in Section 3, we discuss runtime verification in the context of correctness properties specified in linear temporal logic. Some extensions working with more restricted or more expressive specification languages are summarized in Section 4. In Section 5, we sketch approaches which do not only monitor but also intervene in the system under scrutiny. Runtime verification for contracts is discussed in Section 6. Finally, we draw our conclusions in Section 7.

2 Runtime Verification

In this paper, we follow [25] and define a *software failure* as a deviation between the *observed* behavior and the *required* behavior of the software system. A *fault* is defined as the deviation between the current behavior and the expected behavior, which is typically identified by a deviation of the current

2

and the exspected state of the system. A fault might lead to a failure, but not necessarily. An error, on the other hand, is a mistake made by a human that results in a fault and possibly in a failure.

According to IEEE [1], *verification* comprises all techniques suitable for showing that a system satisfies its specification. Traditional verification techniques comprise theorem proving [12], model checking [20], and testing [49,16]. A relatively new direction of verification is *runtime verification*,[1] which manifested itself within the previous years as a *lightweight* verification technique:

Definition (Runtime Verification): 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 satisfies or violates a given correctness property.*

Runtime verification itself deals (only) with the *detection* of violations (or satisfactions) of correctness properties. Thus, whenever a violation has been observed, it typically does not influence or change the program's execution, say for trying to repair the observed violation. However, runtime verification is the basis for concepts also dealing with observed problems, as we discuss in Section 5.

2.1 Monitors

In this paper, a run of a system is understood as a possibly infinite sequence of the system's states, which are formed by current variable assignments, or as the sequence of actions a system is emitting or performing. Formally, a run may be considered as a possible infinite *word* or *trace*. An *execution* of a system is a *finite prefix* of a run and, formally, it is a finite trace. When running a program, we can only observe executions, which, however, restrict the corresponding evolving run as being their prefix. While, in verification, we are interested in the question whether a run, and more generally, all runs of a system adhere to given correctness properties, executions are the primary object analyzed in the setting of RV.

Checking whether an execution meets a correctness property is typically performed using a *monitor*. In its simplest form, a monitor decides whether the current execution satisfies a given correctness property by outputting either *yes/true* or *no/false*. Formally, when $[\![\varphi]\!]$ denotes the set of valid executions given by property φ, runtime verification boils down to checking whether the execution w is an element of $[\![\varphi]\!]$. Thus, in its mathematical essence, runtime verification answers the *word problem*, i.e. the problem whether a given

---

[1] `http://www.runtime-verification.org`

word is included in some language. Note that often, the word problem can be decided with lower complexity compared to, for example, the subset problem: Language containment for non-deterministic finite-automata is PSPACE-complete [58] (which is essential for model checking, see Section 2.2), while deciding whether a given word is accepted by a non-deterministic automaton is NLOGSPACE-complete [41].

However, to cover richer approaches to RV, we define the notion of monitors in a slightly more general form:

Definition (Monitor): *A* monitor *is a device that reads a finite trace and yields a certain* verdict.

Here, a verdict is typically a truth value from some truth domain. A truth domain is a lattice with a unique top element *true* and a unique bottom element *false*. This definition covers the standard two-valued truth domain $\mathbb{B} = \{true, false\}$ but also fits for monitors yielding a probability in $[0, 1]$ with which a given correctness property is satisfied. Sometimes, one might be even more liberal and consider also verdicts that are not elements of a truth domain, though we do not follow this view in this paper.

A monitor may on one hand be used to check the *current* execution of a system. In this setting, which is termed *online monitoring*, the monitor should be designed to consider executions in an *incremental fashion* and in an *efficient manner*. On the other hand, a monitor may work on a (finite set of) *recorded* execution(s), in which case we speak of *offline monitoring*.

For a monitor to be ideally suited for runtime verification, it should adhere to the two maxims *impartiality* and *anticipation*. *Impartiality* requires that a finite trace is not evaluated to *true* or, respectively *false*, if there still exists an (infinite) continuation leading to another verdict. *Anticipation* requires that once every (infinite) continuation of a finite trace leads to the same verdict, then the finite trace evaluates to this very same verdict. Intuitively, the first maxim postulates that a monitor only decides for *false*—meaning that a misbehavior has been observed—or *true*—meaning that the current behavior fulfills the correctness property, regardless of how it continues—only if this is indeed the case. Clearly, this maxim requires to have at least three different truth values: *true*, *false*, and *inconclusive*, but of course more than three truth values might give a more precise assessment of correctness. The second maxim requires a monitor to indeed report *true* or *false*, if the correctness property is indeed violated or satisfied. In simple words, *impartiality* and *anticipation,* guarantee that the semantics is neither premature nor overcautious in its evaluations. See [11,9] for a more elaborate discussion of these issues in the context of linear temporal logic.

In runtime verification, monitors are typically generated automatically from

some high-level specification. As runtime verification has its roots in model checking, often some variant of linear temporal logic, such as LTL [51], is employed. But also formalisms inspired by the linear μ-calculus have been introduced, for example in [24], which explains an accompanying monitoring framework . Actually, one of the key problems addressed in runtime verification is the generation of monitors from high-level specifications, and we discuss this issue for LTL in more detail in Section 3.

2.2 *Runtime Verification versus Model Checking*

In essence, model checking describes the problem of determining whether, given a model $\mathcal{M}$ and a correctness property φ, all computations of $\mathcal{M}$ satisfy φ. Model checking [20], which is an automatic verification technique, is mainly applicable to finite-state systems, for which all computations can exhaustively be enumerated.

In the automata theoretic approach to model checking [65], a correctness property φ is transformed to an automaton $\mathcal{M}_{\neg\varphi}$ accepting all runs violating φ. This automaton is put in parallel to a model $\mathcal{M}$ to check whether $\mathcal{M}$ has a run violating φ.

Runtime verification has its origins in model checking, and, to a certain extend, the key problem of generating monitors is similar to the generation of automata in model checking. However, there are also important differences to model checking:

- While in model checking, *all executions* of a given system are examined to answer whether they satisfy a given correctness property φ, which corresponds to the language inclusion problem, runtime verification deals with the word problem.
- While model checking typically considers *infinite* traces, runtime verification deals with *finite* executions—as executions have necessarily to be finite.
- While in model checking a complete model is given allowing to consider arbitrary positions of a trace, runtime verification, especially when dealing with online monitoring, considers finite executions of increasing size. For this, a monitor should be designed to consider executions in an *incremental fashion*.

These differences make it necessary to adapt the concepts developed in model checking to be applicable in runtime verification. For example, while checking a property in model checking using a kind of backwards search in the model is sometimes a good choice, it should be avoided in online monitoring as this would require, in the worst case, the whole execution trace to be stored for evaluation.

From an application point of view, there are also important differences between model checking and runtime verification.

Runtime verification deals only with observed executions as they are generated by the real system. Thus runtime verification is applicable to *black box systems* for which no system model is at hand. In model checking, however, a suitable model of the system to be checked must be constructed as—before actually running the system—all possible executions must be checked.

If such a precise model of the underlying system is given, and, if moreover a bound on the size of its state space is known, powerful, so-called *bounded model-checking techniques* can be applied [13] for analyzing the system. The crucial idea, which is equally used in conformance testing [66,19], is that for every finite-state system, an infinite trace must reach at least one state twice. Thus, if a finite trace reaches a state a second time, the trace can be extended to an infinite trace by taking the corresponding loop infinitely often. Likewise, considering all finite traces of length up-to the state-place plus one, one has information on all possible loops of the underlying system, without actually working on the system's state space directly.

Clearly, similar correspondences would be helpful in runtime verification as well. However, in runtime verification, an upper bound on the system's state space if typically not known. More importantly, the states of an observed execution usually do not reflect the system's state completely but do only contain the value of certain variables of interest. Thus, seeing a state twice in an observed execution does not allow to infer that the observed loop can be taken ad infinitum.

Furthermore, model checking suffers from the so-called *state explosion problem*, which terms the fact that analyzing all executions of a system is typically been carried out by generating the whole state space of the underlying system, which is often huge. Considering a single run, on the other hand, does usually not yield any memory problems, provided that when monitoring online only a finite *history* of the execution has to be stored.

Last but not least, in online monitoring, the complexity for *generating* the monitor is typically negligible, as the monitor is often only generated once. However, the *complexity of the monitor*, i.e. its memory and computation time requirements for checking an execution are of important interest, as the monitor is part of the running system and should influence the system as less as possible.

As runtime verification does not consider each possible execution of a system, but just a single or a finite subset, it shares similarities with *testing*, which terms a variety of usually incomplete verification techniques.

Typically, in testing one considers a finite set of finite input-output sequences forming a *test suite* [53]. Test-case execution is then checking whether the output of a system agrees with the predicted one, when giving the input sequence to the system under test.

A different form of testing, however, is closer to runtime verification, which is sometimes termed *oracle-based testing*. Here, a test-suite is only formed by input-sequences. To make sure that the output of the system is as anticipated, a so-called *test oracle* has to be designed and "attached" to the system under test. Thus, in essence, runtime verification can be understood as this form of testing. There are, however, differences in the foci of runtime verification and oracle-based testing:

- In testing, an oracle is typically defined directly, rather than generated from some high-level specification.
- On the other hand, providing a suitable set of input sequences to "exhaustively" test a system, is rarely considered in the domain of runtime verification.

Thus, runtime verification can also considered as a form of *passive testing*.

When monitors are equipped in the final software system, one may also understand runtime verification as "testing forever", which makes it, in a certain sense, complete.

2.4   When to use Runtime Verification?

Let us conclude the description of runtime verification by listing certain application domains, highlighting the distinguishing features of runtime verification:

- The verification verdict, as obtained by model checking or theorem proving, is often referring to a model of the real system under analysis, since applying these techniques directly to the real implementation would be intractable. The model typically reflects most important aspects of the corresponding implementation, and checking the model for correctness gives useful insights to the implementation. Nevertheless, the implementation might be-

have slightly different than predicted by the model. Runtime verification may then be used to easily check the actual execution of the system, to make sure that the implementation really meets its correctness properties. Thus, runtime verification may act as a partner to theorem proving and model checking.

- Often, some information is available only at runtime or is conveniently checked at runtime. For example, whenever library code with no accompanying source code is part of the system to build, only a vague description of the behavior of the code might be available. In such cases, runtime verification is an alternative to theorem proving and model checking.
- The behavior of an application may depend heavily on the environment of the target system, but a precise description of this environment might not exist. Then it is not possible to obtain the information necessary to test the system in an adequate manner. Moreover, formal correctness proofs by model checking or theorem proving may only be achievable by taking certain assumptions on the behavior of the environment—which should be checked at runtime. In this scenario, runtime verification outperforms classical testing and adds on formal correctness proofs by model checking and theorem proving.
- In the case of systems where security is important or in the case of safety-critical systems, it is useful also to monitor behavior or properties that have been statically proved or tested, mainly to have a double check that everything goes well: Here, runtime verification acts as a partner of theorem proving, model checking, and testing.

The above mentioned items can be found in a combined manner especially in highly dynamic systems such as *adaptive*, *self-organizing*, or *self-healing* systems (see [39] for an overview on such approaches towards self-management). The behavior of such systems depends heavily on the environment and changes over time, which makes their behavior hard to predict—and hard to analyze prior to execution. To assure certain correctness properties of especially such systems, we expect runtime verification to become a major verification technique. More specifically, we anticipate a runtime verification based component to be part of the architecture of such dynamic systems, as explained in more detail in Section 5.

3   Runtime Verification of Linear Temporal Logic Specifications

In runtime verification, which is heavily inspired by model checking, a correctness property φ is automatically translated into a monitor. Correctness properties specify the form of individual executions of a system and are usually formulated in some variant of linear temporal logic, such as LTL [51], as seen for example in [35,32,31,37,38,60]. In this section, we recall the ideas of

LTL$_3$ as one example of a linear-time temporal logic specially designed for runtime verification. For a technical presentation, see [10,8].

As especially in the model checking community, Pnueli's LTL [51] is a well-accepted linear-time temporal logic used for specifying properties of infinite traces one usually wants to check the very same properties in runtime verification as well. However, one has to interpret their semantics with respect to finite prefixes as they arise in observing actual systems. This approach to runtime verification is summarized in the following rationale:

> *Pnueli's LTL is a well-accepted linear-time temporal logic used for specifying properties of infinite traces. In runtime verification, our goal is to check LTL properties given* finite prefixes of infinite traces.

Therefore, LTL$_3$'s syntax coincides with LTL, while its semantics is given for finite traces.

To implement the idea that, for a given LTL$_3$ formula, its meaning for a prefix of an infinite trace should correspond to its meaning considered as an LTL formula for the full infinite trace, we use *three* truth values: *true*, *false*, and *inconclusive*, denoted respectively by $\top$, $\bot$, and ?. More precisely, given a finite word u and an LTL$_3$ formula φ, the semantics is defined as follows:

- if there is no continuation of u satisfying φ (considered as an LTL formula), the value of φ is *false*;
- if every continuation of u satisfies φ (considered as an LTL formula), it is *true*;
- otherwise, the value is *inconclusive* since the observations so far are inconclusive, and neither *true* or *false* can be determined.

While there are actually semantics for LTL on finite traces [42,47,29], these use (only) two truth values. We strongly believe that only two truth values lead to misleading results in runtime verification: As a first example, consider a property $G\neg p$, stating that no state satisfying p should occur. Clearly, when p is observed, the corresponding monitor should complain. However, as long as p does not hold, it is misleading to say that the formula is *true*, since the next observation might already violate the formula. On the other hand, the formula Fp, stating that eventually a p is observed, is fulfilled (only) when a first p is observed. As a second example, consider the formula $\neg p\,U\,init$ (read: *not p until init*) stating that nothing bad (p) should happen before the init function is called. If within an execution p becomes true before *init*, the formula is violated and thus *false* (for any continuation of the current execution). If, on the other hand, the init function has been called and no p has been observed before, the formula is *true*, regardless of what will happen in the future. Besides observing faults, for testing and verification, it is equally important to know whether some property is indeed *true* or whether the current observation is

just inconclusive and a violation of the property to check may still occur.

Originally, we proposed this three-valued semantics and its use for runtime verification in [8]. However, some essential concepts were defined by Kupferman and Vardi: In [43] a *bad prefix* (of a Büchi automaton) is defined as a finite prefix which cannot be the prefix of any accepting trace. Dually, a *good prefix* is a finite prefix such that any infinite continuation of the trace will be accepted. It is exactly this classification that forms the basis of our 3-valued semantics: "bad prefixes" (of formulae) are mapped to *false*, "good prefixes" evaluate to *true*, while the remaining prefixes yield *inconclusive*.

For a given LTL_3 formula, we describe in [8] how to construct a (deterministic) finite state machine (FSM) with three output symbols. This automaton reads finite traces and yields their three-valued semantics. Thus, monitors for three-valued formulae classify prefixes as one of $good = \top$, $bad = \bot$, or ? (neither *good* nor *bad*). Standard minimization techniques for FSMs can be applied to obtain a unique FSM that is *optimal* with respect to its number of states. In other words, any smaller FSM must be non-deterministic or check a different property. As an FSM can straightforwardly be deployed, we obtain a practical framework for runtime verification.

The proposed semantics of LTL_3 has a valuable implication for a corresponding monitor. It requires the monitor to report a violation of a given property *as early as possible*: Since any continuation of a bad (good) prefix is bad (respectively good), there exists a *minimal* bad (good) prefix for every bad (good) prefix. In runtime verification, we are interested in getting feedback from the monitor as early as possible, i. e., for minimal prefixes, let them be either good or bad. Since all bad prefixes for a formula φ yield *false* and good prefixes yield *true*, also minimal ones do so. Thus, the correctness of our monitor procedure ensures that already for *minimal* good or bad prefixes either *true* or *false* is obtained. In other words, the corresponding monitor fulfills both maxims *impartiality* and *anticipation*.

In [23], a Büchi automaton was modified to serve as a monitor reporting *false* for minimal bad prefixes. However, no precise semantics in terms of LTL of the resulting monitor was given. As such, LTL_3 can be understood as a logic which complements the constructions carried out in [23] with a formal framework.

It is natural to ask which LTL properties are *monitorable* at all. Pnueli and Zaks [52] define a property as monitorable with respect to a trace whenever a corresponding monitor might still report a violation (or satisfaction). In [10], it has been shown that the popular belief that monitoring is only suitable for safety properties is misleading: The class of monitorable properties is richer than the union of safety and co-safety properties. For example, the property $\varphi \equiv ((p \vee q)Ur) \vee Gp$ is monitorable while it is neither safe or co-safe: $ppp\ldots$

satisfies φ but none of its prefixes u is good—and therefore φ is not safe. Analogously, $qqq\ldots$ violates φ but none of its prefixes is bad, and hence φ is not co-safe as well. But on the other hand, one can show that every prefix is continuable to either satisfaction or violation of φ.

However, there remain many properties which are *non-monitorable:* Consider for example the request/acknowledge property $G(r \to Fa)$ which states that every request is finally acknowledged: any pending request might be acknowledged in the next observation, so a finite trace is never sufficient to prove a violation. Moreover, any finite trace might be continued with a new request that will never be granted, so that the formula cannot be evaluated to *true* either for any finite trace. Since such properties arise often in practice, a solution which evaluates such a property irrespectively to the inconclusive verdict is quite *ugly*—raising the question of whether it is possible to refine the inconclusive verdict into a more telling verdict.

In [9,11], we introduce a four-valued semantics for LTL which refines the inconclusive verdict into a *presumably true* and *presumably false* truth value, and call the resulting logic *Runtime Verification Linear Temporal Logic* (RV-LTL). In other words, RV-LTL's semantics indicates whether a finite trace describes a system behavior which either (1) satisfies the monitored property, (2) violates the property, (3) will presumably violate the property, or (4) will presumably conform to the property in the future, once the system has stabilized. Using these truth values, we resolved the *ugly* situation of facing an invariably inconclusive verdict in verifying a system at runtime: As long as the final verdict depends on future events, an RV-LTL-based monitor displays a presumably true valuation—if no unanswered request is pending—and presumably false otherwise.

Instead of using good/bad-prefixes as the basis for runtime verification, one could rely on Kupferman's and Vardi's notion of *informative prefixes* [43]. Intuitively, a prefix is informative if it "explains" whether a formula holds or not. Consider for example the formula $XXX(p \wedge \neg p)$, saying that after three letters, p should and should not hold. Clearly, the formula is not satisfiable and every finite trace is a bad prefix. Nevertheless, only a trace of four letters is considered *informative*, as each of the first three letters defers checking the remaining subformula on the remaining string, and it is only with the fourth letter to check whether it satisfies $p \wedge \neg p$, which is then appearently not true.

Especially (formula) rewriting-based and alternating automata-based approaches for checking correctness properties at runtime follow a semantics which is based on informative prefixes, as for example the approaches shown in [36,30,59]. For these works, however, the maxim of anticipation is not fulfilled, since sometimes a violation of a property is reported "too late", as shown in the example. See [10] for a more elaborate discussion on this topic.

4 Runtime Verification with Different Levels of Expressiveness

In the previous section, we have considered monitors for LTL specifications. LTL is particularly useful when specifying restricted regular properties of discrete time systems. However, many interesting properties to be monitored at runtime are not expressible in LTL. On the other hand, we learned that request/acknowledge properties are not monitorable in the sense of [52], even though they are expressible in LTL. In this section, we briefly describe extensions, restrictions, and alternatives to the use of LTL in runtime verification. Note that we only list a few logic-based runtime verification approaches. For a more comprehensive list of fault-monitoring tools see [25].

Often, runtime verification is only considered for safety properties, as for example in [37]. In simple words, an infinite word violates a safety property [57] if and only if a finite prefix violates the property. Thus, safety properties are monitorable. In other words, restricting a logical formalism to its safety fragment, avoids to deal with the issue of (non-)monitorability. Similarly, restricting to only past operators in linear temporal logics avoids semantic issues with respect to the unknown future of a trace.

A typical property to check at runtime is that every file that was opened is eventually closed. A straightforward specification in LTL would be $G(open \rightarrow F\,close)$. However, when opening and closing different files, one has to make sure that corresponding files are opened and closed. Thus, a more appropriate specification would be $\bigwedge_{x\in\mathbb{N}} G(open(x) \rightarrow F\,close(x))$ where x ranges over arbitrary file identifiers. An approach for monitoring LTL enriched with such *parameterized* propositions is carried out in [59] within the J-LO project.

Non-regular properties are not expressible in LTL. Thus, expressing a proper nesting of operators like calls and returns cannot be expressed in LTL. In [56], synthesizing monitors for safety properties is described for a logic, that considers nested calls and returns.

LOLA [24] is a monitor generation framework based on the linear μ-calculus with future and past modalities, and with a notion of parallel *streams* of output. It is well suited for synchronous systems such as digital circuits.

In [28], three-valued, anticipatory monitor synthesis procedures for various linear-time temporal logics have been defined, including an LTL version allowing to formulate properties of integer constraints [26].

In practice, often also the exact timing of events has to be checked. So far, however, not many approaches for runtime verification of real-time properties have been introduced. In [34], monitor synthesis based on LTL enriched with a quantifier allowing to freeze the actual time to a variable which may be later

on refered to is studied. In [61] and [14], *fault diagnosis* for timed systems is examined, a problem that shares some similarities with runtime verification yet is more complicated.

In [8,10], we have extended the three-valued approach also to monitoring of real-time properties specified in the temporal logic TLTL [55]. Thus, a three-valued semantics for TLTL is defined yielding one of *true*, *false*, or *inconclusive* for a finite timed trace, i.e., a trace of events enriched with time stamps denoting their time of occurrence. The procedure follows the scheme for LTL_3, yet uses a region automaton-based analysis known from model checking timed automata [5] to decide the verdict in an anticipatory manner. Note that in contrast to the timed LTL version proposed in [34], TLTL also allows the formulation of formulas predicting the time of the occurence of an event.

Monitor synthesis for the the logic Metric Interval Temporal Logic (MITL), which is another popular temporal logic for specifying realtime properties, is studied in [46]. MITL and TLTL both allow to reason on a finite set of properties. In [45,50] monitoring of continuous signals is considered, which is intrinsically different to observing discrete signals in a continuous time domain.

The Eagle temporal logic [6] is a basic, yet, very general specification language suitable for monitoring correctness properties. It is based on recursive rules over three temporal connectives: *next*, *previous*, and *concatenation*. It allows to encode several other logical languages, for example future time temporal logic, past-time logic, extended regular expressions, μ-calculus, or state machines. It allows data-bindings, which caters for checking also real-time properties or for collecting statistics of the observed traces.

Another recent example to strike the right balance between the expressiveness of the employed specification formalism and its capabilities is presented in [21]: LARVA is a tool which generates monitors for specifications formulated in terms of dynamic and communicating automata employing events and timers—and is therefore able to handle timing as well as contextual information.

A serious difficulty for monitoring systems that is especially apparent when considering real time and hybrid systems is that the execution of the monitor on the same platform on which the system is executed might influence the overall system behaviour. Thus, a system with implanted monitors could satisfy different correctness properties than the system without its monitors. In such cases, the only simple way out is to use dedicated monitoring hardware *not* affecting the observed system [64,62,63].

5 Beyond Runtime Verification

One of the distinguishing features of runtime verification compared to other verification techniques is due to the fact that verification, at least in online monitoring, is performed while executing a program. This offers the possibility to react to violations of correctness properties. More specifically, we runtime verification allows to react on faults, before they turn into failures. This allows applications of runtime verification techniques, in which other verification techniques cannot help.

5.1 Applications

In huge systems, certain (hardware) components may fail now and then. Using monitors to check the expected behavior, such a fault may be observed and reported. Additional code for *mitigation* may then react to the fault, for example, by displaying a corresponding error message. In a certain sense, this is similar to the well-known concept of exception handling in programming languages, however, here, exceptions are defined by violations to correctness properties.

However, not only reliability issues may benefit from runtime verification techniques. In some cases it is easy to specify the prohibited behavior whereas it is complicated to specify and verify the allowed behavior—corresponding to the multiplied negation of all proscriptions. Then, it might be even *impractical to implement* all these rules, whereas a go-ahead-check-and-repair strategy might be an easy and effective solution to the problem. For example, one might develop a web server first as an application answering each (atomic) request in the expected manner. To make sure that the server indeed follows a certain protocol, for example, that a user only receives a document from some database after being logged in, may be enforced by monitors and corresponding mitigation code: If the user has not logged in when trying to receive a document (which can easily be checked by a monitor), the mitigation code will display a login screen.

5.2 Approaches to React at Runtime

The idea of monitoring a system and reacting has appeared in different manifestations in computer science. In this section, we briefly introduce three such approaches: First, we describe *fault detection, identification, and recovery (FDIR),* as an example for a traditional methodological framework. Then we

introduce *runtime reflection (RR)* and *monitor-oriented programming (MOP)* as current and runtime verification based approaches.

5.2.1 FDIR

The ideas outlined above, are to a certain extent covered by the popular notion of *FDIR*, which stands for *Fault Detection, Identification, and Recovery* or sometimes for *Fault Diagnosis, Isolation, and Recovery* or various combinations thereof [22]. The general idea of FDIR is that a failure within a system shows up by a fault. A fault, however, does typically not *identify* the failure: for example, there might be different *reasons* why a monitored client does not follow a certain protocol, one of them, e.g., that it uses an old version of a protocol. If this is identified as the failure, reconfiguration may switch the server to work with the old version of the protocol.

Crow and Rushby instantiated the scheme FDIR using Reiter's theory of diagnosis from first principles in [22]. Especially, the detection of errors is carried out using diagnosis techniques. In *runtime reflection* [7], runtime verification is proposed as a tool for fault detection, while a simplified version of Reiter's diagnosis is suggested for identification.

5.2.2 Runtime Reflection

Monitor-based runtime reflection or short *runtime reflection* (RR) is an architecture pattern for the development of reliable systems. The main idea is that a *monitoring layer* is enriched with a *diagnosis layer* and a subsequent *mitigation layer*. We only show the pattern with respect to information flow in a conceptual manner and refer to [7] for a full presentation especially presenting a realization for a distributed system.

The architecture consists of four layers as shown in Figure 1, whose role will be sketched in the subsequent paragraphs.

Logging—Recording of system events. The role of the *logging layer* is to observe system events and to provide them in a suitable format for the monitoring layer. Typically, the logging layer is realized by adding code annotations within the system to build. However, separated stand-alone loggers, logging for example network traffic, can realize this layer as well. While the goal of a
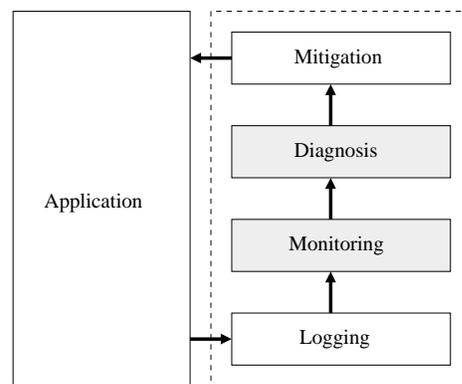

Fig. 1. An application and the layers of the runtime reflection framework.

15

logger is to provide information on the current run to a monitor, it may not assume (much) on the properties to be monitored.

Monitoring—Fault detection. The *monitoring layer* consists of a number of monitors (complying to the logger interface of the logging layer) which observe the stream of system events provided by the logging layer. Its task is to detect the presence of faults in the system without actually affecting its behavior. In runtime reflection, it is assumed to be implemented using runtime verification techniques. If a violation of a correctness property is detected in some part of the system, the generated monitors will respond with an alarm signal for subsequent diagnosis.

Diagnosis—Failure identification. Following FDIR, we separate the detections of faults from the identification of failures. The *diagnosis layer* collects the verdicts of the distributed monitors and deduces an explanation for the current system state. For this purpose, the diagnosis layer may infer a (minimal) set of system components, which must be assumed faulty in order to explain the currently observed system state. The procedure is solely based upon the results of the monitors and general information on the system. Thus, the diagnostic layer is not directly communicating with the application.

Mitigation—Reconfiguration. The results of the system's diagnosis are then used in order to *reconfigure* the system to mitigate the failure, if possible. However, depending on the diagnosis and the occurred failure, it may not always be possible to re-establish a determined system behavior. Hence, in some situations, e. g., occurrence of fatal errors, a recovery system may merely be able to store detailed diagnosis information for off-line treatment.

5.2.3 Monitor-oriented Programming

Monitoring-Oriented Programming (MOP) [18], proposed by Feng and Rosu, is a software development methodology, in which the developer specifies desired properties using a variety of (freely definable) specification formalisms, along with code to execute when properties are violated or validated. The MOP framework automatically generates monitors from the specified properties and then integrates them together with the user-defined code into the original system. Thus, it extends ideas from runtime verification by means for *reacting* on detected violations (or validations) of properties to check. This allows the development of *reflective* software systems: A software system can monitor its own execution such that the subsequent execution is influenced by the code a monitor is executing in reaction to its observations–again influencing the observed behavior and consequently the behavior of the monitor itself.

5.2.4 MOP versus RR

RR differs from monitor-oriented programming in two dimensions. First, MOP aims at a programming methodology, while RR should be understood as an architecture pattern. This implies that MOP support has to be tight to a programming language, for example Java resulting in jMOP, while in RR, a program's structure should highlight that it follows the RR pattern. The second difference of RR in comparison to MOP is that RR introduces a diagnosis layer not found in MOP.[2]

6   Runtime Verification and Contract Enforcement

The notion of *contracts* is getting popular in computer science. For example, there is contract oriented programming [48] which has its origin in Hoare's pre- and post-conditions [40] or contracts describing the behavior of web services and their composition [2,3,17]. Also, the operations of an organization conforming to a body of regulations could be stated by means of contracts to be used for later compliance checking [4,15,33]. Actually, further examples and how to deal with them formally are given within this current special issue in the various articles. For the discussion to come, we follow the definition given in [54].

Definition (Contract [54]): *A* contract *is a document which engages several parties in a transaction and stipulates their obligations, rights, and prohibitions, as well as penalties in case of contract violation.*

It is important to understand this definition in detail. Reading the first part of the definition, we see, in simple words, that a contract defines a mixture of the expected and possible behavior of the involved parties. The last part of this definition, however, caters also for violations to contracted behavior. More specifically, if the behavior *violates* the contract, a *penalty* is due. Then, however, paying the penalty can be understood as a behavior according to the contract. Clearly, we should distinguish such behavior from situations in which one party does neither follow the expected behavior nor does pay an a priory agreed penalty. In that case, we say that one party *breaks* the contract.

Contract enforcement is the problem of monitoring *contract fulfillment* as well as enforcing the penalty, when a contract violation has been observed. Following the discussion above, we define monitoring *contract breakage* as the

---

[2] Clearly, in the MOP framework, a diagnosis can be carried out in the code triggered by a monitor. This yields a program using the MOP methodology and following the RR pattern.

problem of checking whether the contract is fulfilled or whether, in case of contract violation, the necessary penalty is paid.

In [54], a formal language $\mathcal{CL}$ for contracts is introduced which allows to specify obligations, permissions, and prohibitions over actions. Then, a trace semantics for $\mathcal{CL}$ is established and used in conjunction with runtime verification techniques to generate finite-state monitoring procedures for such contracts [44].

Likewise, in [27], checking for contract breakage is studied in the setting of regulatory conformance. To cater for the domain specific way of formulating conformance, linear temporal logic is extended to distinguish between obligations and permissions, and to allow statements to refer to others. Moreover, a synthesis algorithm yielding efficient monitors for the resulting language is introduced.

For contract enforcement, we are not aware of any architectural approach. However, contract enforcement apparently matches runtime reflection: Monitoring contract fulfillment provides suitable properties to verify at runtime. In case of a contract violation, the enforcement of penalties asks for mitigation. However, a detailed study in this direction has to be done.

7   Conclusion

In this paper, we presented the gist of runtime verification. We have identified that runtime verification deals with verification techniques that allow checking whether an execution of a system under scrutiny satisfies or violates a given correctness property. Moreover, we have learned that runtime verification has its roots in model checking and that one of its main technical challenges is the synthesis of efficient monitors from logical specifications.

In a certain sense, runtime verification is an old story: monitoring, software-fault analysis, runtime checking, runtime verification, diagnosis—they all aimed and aim at extracting an execution trace to be analyzed. The name changed with the technology shifts in computer science—but the general idea prevailed and proved to be successful. Currently, the focus on dependable and embedded systems together with the background provided by progress in the broader field of formal methods, especially model checking, are fertilizing the development of runtime verification techniques and propel their application into industry.

Moreover, the emerging paradigms of service oriented architectures, adaptive and self-healing systems, and the use of electronic contracts ask for monitor-

ing executions of the respective systems/contracts and subsequent reaction in case of misbehavior, as these systems hard to analyze statically due to their dynamic nature. Especially for these applications, runtime verification techniques form a powerful basis.

References

[1] IEEE Std 1012 - 2004 IEEE standard for software verificiation and validation. *IEEE Std 1012-2004 (Revision of IEEE Std 1012-1998)*, pages 1–110, 2005.

[2] A. Alves, A. Arkin, S. Askary, C. Barreto, et al. Web services business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, April 2007. OASIS Standard.

[3] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, et al. Web services conversation language (WSCL) 1.0. http://www.w3.org/TR/2002/NOTE-wscl10-20020314, March 2002. W3C.

[4] A. Abrahams. *Developing and Executing Electronic Commerce Applications with Occurrences.* PhD thesis, University of Cambridge, 2002.

[5] R. Alur and D. Dill. Automata for modeling real-time systems. In *Automata, languages and programming*, pages 322–335. Springer, 1990.

[6] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, pages 44–57, 2004.

[7] A. Bauer, M. Leucker, and C. Schallhart. Model-based methods for the runtime analysis of reactive distributed systems. In *Proceedings of the Australian Software Engineering Conference (ASWEC'06)*. IEEE, 2006. to appear.

[8] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of realtime properties. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS'06)*, 2006. 260–272.

[9] A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly—but how ugly is ugly? In *Proceedings of the 7th International Workshop on Runtime Verification (RV'07)*, pages 126–138, 2007.

[10] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. Technical Report TUM-I0724, TU München, 2007.

[11] A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly—but how ugly is ugly? Technical Report TUM-I0803, TU München, 2008.

[12] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions.* An EATCS Series. Springer, 2004.

[13] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. volume 58 of *Advances in Computers*. Academic press, 2003.

[14] P. Bouyer, F. Chevalier, and D. D'Souza. Fault diagnosis using timed automata. In *Proceedings of the 8th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS'05)*, pages 219–233, 2005.

[15] T. D. Breaux, M. W. Vail, and A. I. Anton. Towards regulatory compliance: Extracting rights and obligations to align requirements with regulations. In *Proceedings of the 14th International Requirements Engineering Conference (RE'06)*, 2006.

[16] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[17] G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. In *Proceedings of the 35th Symposium on Principles of programming languages (POPL'08)*, pages 261–272, 2008.

[18] F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07)*, 2007. to appear.

[19] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering (TSE)*, 4(3):178–187, 1978.

[20] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[21] C. Colombo, G. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'08)*, 2008. to appear.

[22] J. Crow and J. Rushby. Model-based reconfiguration: Diagnosis and recovery. NASA Contractor Report 4596, NASA Langley Research Center, 1994.

[23] M. d'Amorim and G. Rosu. Efficient monitoring of omega-languages. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, pages 364–378, 2005.

[24] B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. LOLA: runtime monitoring of synchronous systems. In *Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 166–174, 2005.

[25] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering (TSE)*, 30(12):859–872, 2004.

[26] S. Demri. LTL over integer periodicity constraints. *Theoretical Computer Science (TCS)*, 360(1-3):96–123, 2006.

[27] N. Dinesh, A. Joshi, I. Lee, and O. Sokolsky. Checking traces for regulatory conformance. In *Proceedings of the 8th International Workshop on Runtime Verification (RV'08)*, 2008.

[28] W. Dong, M. Leucker, and C. Schallhart. Impartial anticipation in runtime verification. In *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis (ATVA'08)*, 2008. to appear.

[29] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout. Reasoning with temporal logic on truncated paths. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, pages 27–39, 2003.

[30] M. Geilen. On the construction of monitors for temporal logic properties. *Electronic Notes on Theoretical Computer Science (ENTCS)*, 55(2), 2001.

[31] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE'01)*, pages 412–416, 2001.

[32] D. Giannakopoulou and K. Havelund. Runtime analysis of linear temporal logic specifications. Technical Report 01.21, RIACS/USRA, 2001.

[33] C. Giblin, A. Liu, S. Muller, B. Pfitzmann, and X. Zhou. Regulations expressed as logical models (realm). In M. F. Moens and P. Spyns, editors, *Legal Knowledge and Information Systems*. 2005.

[34] J. Håkansson, B. Jonsson, and O. Lundqvist. Generating online test oracles from temporal logic specifications. *Journal on Software Tools for Technology Transfer (STTT)*, 4(4):456–471, 2003.

[35] K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. *Electronic Notes on Theoretical Computer Science (ENTCS)*, 55(2), 2001.

[36] K. Havelund and G. Rosu. Monitoring programs using rewriting. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE'01)*, page 135, 2001.

[37] K. Havelund and G. Rosu. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, pages 342–356, 2002.

[38] K. Havelund and G. Rosu. Efficient monitoring of safety properties. *Journal on Software Tools for Technology Transfer (STTT)*, 2004.

[39] M. Hinchey and R. Sterritt. Self-managing software. *IEEE Computer*, 39(2):107–109, 2006.

[40] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM (CACM)*, 12(10):576–580, 1969.

[41] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, first edition, 1979.

[42] H. W. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, Los Angeles, 1968.

[43] O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Formal Methods in System Design (FMSD)*, 19(3):291–314, 2001.

[44] M. Kyas, C. Prisacariu, and G. Schneider. Run-time monitoring of electronic contracts. In *6th International Symposium on Automated Technology for Verification and Analysis (ATVA'08)*, 2008. to appear.

[45] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, and Formal Techniques in Real-Time and Fault-Tolerant Systems (FORMATS/FTRTFT'04)*, pages 152–166, 2004.

[46] O. Maler, D. Nickovic, and A. Pnueli. From MITL to Timed Automata. In *Proceedings of the 4th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'06)*, pages 274–289, 2006.

[47] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, New York, 1995.

[48] B. Meyer. Contract-driven development. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE'07)*, page 11, 2007.

[49] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler. *The Art of Software Testing*. John Wiley and Sons, 2 edition, 2004.

[50] D. Nickovic and O. Maler. Amt: A property-based monitoring tool for analog systems. In *Proceedings of the 5th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'07)*, pages 304–319, 2007.

[51] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Symposium on the Foundations of Computer Science (FOCS'77)*, pages 46–57, 1977.

[52] A. Pnueli and A. Zaks. PSL Model Checking and Run-Time Verification Via Testers. In *Proceedings of the 14th International Symposium on Formal Methods (FM'06)*, pages 573–586, 2006.

[53] A. Pretschner and M. Leucker. Model-based testing - a glossary. In Broy et al. [16], pages 607–609.

[54] C. Prisacariu and G. Schneider. A formal language for electronic contracts. In *9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, pages 174–189, 2007.

[55] J.-F. Raskin and P.-Y. Schobbens. The logic of event clocks—decidability, complexity and expressiveness. *Journal of Automata, Languages and Combinatorics*, 4(3):247–286, 1999.

[56] G. Rosu, F. Chen, and T. Ball. Synthesizing monitors for safety properties - this time with calls and returns. In *Proceedings of the 8th Workshop on Runtime Verification (RV'08)*, 2008. to appear.

[57] A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computation*, 6(5):495–512, 1994.

[58] A. P. Sistla and E. M. Clarke. Complexity of propositional temporal logics. *Journal of the ACM (JACM)*, 32:733–749, 1985.

[59] V. Stolz. Temporal assertions with parametrised propositions. In *Proceedings of the 7th International Workshop on Runtime Verification (RV'07)*, pages 176–187, 2007.

[60] V. Stolz and E. Bodden. Temporal Assertions using AspectJ. In *Proceedings of the 5th International Workshop on Runtime Verification (RV'05)*, pages 109–124, 2006.

[61] S. Tripakis. Fault diagnosis for timed automata. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'02)*, pages 205–224, 2002.

[62] J. J. P. Tsai, Y.-D. Bi, and S. Yang. Debugging for timing-constraint violations. *IEEE Software*, 13(2):89–99, 1996.

[63] J. J. P. Tsai and H.-Y. Chen. A noninvasive architecture to monitor real-time distributed systems. *IEEE Computer*, 23(3):11–23, 1990.

[64] J. J. P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering (TSE)*, 16(8):897–916, 1990.

[65] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science (LICS'86)*, pages 332–345, 1986.

[66] M. P. Vasilevski. Failure diagnosis of automata. *Cybernetic*, 9(4):653–665, 1973.