Interest beyond Violation: On Points-of-Interest in Runtime Verification

Christian Colombo[1], Gordon J. Pace[1], and Gerardo Schneider[2]

[1] Department of Computer Science, University of Malta
{christian.colombo | gordon.pace}@um.edu.mt
[2] Department of Computer Science and Engineering,
Chalmers and the University of Gothenburg
gerardo.schneider@gu.se

Abstract. Many formal verification techniques are concerned with comparing system behaviours with formal specifications. Although runtime verification has followed this path (comparing observed traces against formal properties), it has traditionally been burdened with another task — that of raising a flag when a violation is detected. Different approaches can be found in the literature: identifying the earliest such instance, identifying all instances, identifying instances where (potentially future) violations are inevitable, etc. We argue that the lack of a clear distinction between the notion of system correctness and the hard-wired means of identification of points when violation is somehow detected, conflates the notions of points-of-detection and points-of-violation. Frequently, the point at which a point-of-violation may be detected is independent of the point of interest itself, and also independent of the point-of-reaction if a corrective measure is needed. We observe that this distinction becomes more salient in some cases, such as deontic specification languages, which may identify notions such as permission, and in the case of multi-agent systems, where the notion of blame is essential. Using practical and varied examples we motivate why these limitations are significant for the field of runtime verification.

1 Introduction

Traditionally, runtime verification has emerged from verification as a technique which checks a single trace against a specification, rather than all possible behaviours of a system for potential violation of the specification. The question that verification asks is: *"Does a system behaviour satisfy/violate a given specification?"* Transposed to runtime verification, the question changed to: *"Does a particular finite system trace satisfy/violate a given specification?"* (e.g., [12]).

It was not only the meta-level question that the area of runtime verification inherited from model checking, but also the specification language, i.e., LTL, whose semantics focused on infinite traces rather than finite ones. This was fine if the meta-level question was a simply one of yes/no. However, as the field of runtime verification developed, researchers became also interested in answering

the satisfaction/violation question *as early as possible*. This is when various finite prefix and past semantics (and verification algorithms) of LTL started to emerge (e.g., [1]). Given that LTL was not born with finite traces in mind, the new semantics had to deal with awkward situations such as how to deal with formula fragments which refer to future, yet unseen, parts of a trace (resulting in weak and strong versions of the same operators). While there have been several competing proposals in this respect (see [2] for a comprehensive comparison of LTL semantics used for runtime verification), what is common across all of them, is that they try to answer the basic yes/no question as early as possible in the trace, with some interpretations also including the earliest detection of the possibility of a violation.

In parallel, one also saw a number of finite-trace-based specification languages using past (as opposed to future) time modalities. Being trace-based ensured a direct mapping between the specification and the verification code, and being past-oriented enabled more efficient and incremental algorithms which processed events as they came in with no need for taking into account look-ahead or additional memory. Eventually, stream-native specification languages started to emerge (e.g., [7]), avoiding the need to have stream-based interpretations retrofitted onto trace-based logics.

The above came with the added benefit that rather than doing the best effort to answer the yes/no question as early as possible, the point in the trace at which this becomes possible is clearly defined. This eventually made possible a new runtime verification meta-question: *"Which is the point in the trace at which we can answer yes/no?"* Note that this is a major departure from the original question expecting a boolean answer because the answer now is a precise trace position. Another subtle difference is that while the earlier question was handled in terms of a three(or more)-valued logic to handle satisfaction/violation, this new question inherently ignores this distinction and simply raises a flag at a particular point in time. It is then up to the runtime verification engineer to interpret this as a satisfaction or violation depending on the context. This is how monitor-oriented programming (MOP) [4] came to be: having access to precisely identifiable points in the trace enables the programmer to use them to trigger custom routines. Note that this was not possible without precise semantics focusing on the *when* rather than the *yes/no*. These precise points in the trace which answer the *when* meta-question is what we call *points-of-interest*.

Once runtime verification becomes about identifying points-of-interest, the question of what such points represent is pushed to a higher level of abstraction. Traditionally, when talking about runtime verification, points-of-interest has been mainly *points-of-violation* or *points-of-satisfaction*. However, the idea behind MOP is to use *points-of-interest* as *points-of-trigger* whatever that trigger signifies to the system engineer. We believe that this idea has not yet been explored to its full potential for the area of runtime verification.

Note that the *when* question, seen from the new perspective we are arguing here, is not only relevant for the (dynamic) *verification* of the specification, but it also affects the specification itself. Indeed, to handle the above question, we may

need richer specification languages than LTL. Moreover, what is to be specified also changes: we may now be more explicit about what we want to specify, to detect different points-of-interest as the ones we mentioned, and beyond (as explained below).

Why is the notion of points-of-interest beneficial in the context of system specifications and their verification at runtime? First, doing away with the violation/satisfaction question and focusing on points-of-interest opens the door to several other specification languages such as finite state machines and regular expressions which are just as adequate to identify points-of-interest in a trace.[3] Second (and more importantly for this paper), this approach enables us to create another layer on top of points-of-interest to handle specification complexities that would otherwise be difficult to manage. Consider the following specification from a financial transaction system: *"No money transfers can be carried out by a blacklisted user"* and its corrective action: *"During the daily reconciliation process, illegal transfers should be undone"*. Note that the first statement identifies points-of-interest (which we could call the *points-of-violation*) where illegal transfers occur. However, the points-of-interest where this is handled comes later during reconciliation (we could call those points the *points-of-reaction*).

Specifying points-of-interest (whether to signify violation, satisfaction, reaction, or something else) can be done with any specification language of choice as long as it can answer the question of *when* to raise a flag when monitoring a trace. Naturally, this brings up the issue that the point-of-interest might not be detectable exactly at the same time as it occurs (depending on the underlying semantics of the chosen specification language, the monitoring algorithm, etc). In many cases the two coincide as in the previous example where the violation is detected as it occurs, i.e., upon the first transfer carried out after the user is blacklisted (unless the user can be blacklisted retroactively!). However, the point-of-detection may occur after the point-of-violation. For example consider the statements: *"If more than 10 transfers are attempted in 5 minutes, these should be considered fraudulent"* and *"No fraudulent transfers should be allowed."* To detect the first transfer as fraudulent, one would need to observe the following nine. Therefore, the point of a pattern-based fraud detection usually occurs much later than the initial actions of the fraud itself. Note that both points are relevant, particularly if in response to the fraud one would need to undo the transfers, i.e., all the transfers between the point-of-violation and the point-of-detection.[4]

The point-of-detection may also occur before the point-of-violation. Consider the statements: *"A blacklisted user must remain blacklisted for at least one day"* and *"A user must pay transfer fees within an hour."* In this case, the detection of violation can be detected before it happens, i.e., the moment a user is blacklisted (and as a consequence is not allowed to perform payments) with pending transfer fees. This idea has been well studied as *runtime enforcement* (e.g., [8]),

[3] Also note that with this approach, a single trace can have any number of matches.

[4] Reactions which undo—literally or in some abstract sense—previous system actions due to late monitor detection have been studied in terms of compensations [5].

possibly enabling timely intervention (automatic or not) to avert a violation, e.g., processing fee payment automatically on behalf of the user.

Bringing the above together, by detaching runtime verification from thinking simply in terms of satisfaction and violation, we open up possibilities of dealing more easily with real-world complex specifications: While the point-of-reaction must necessarily occur at the same time or after the point-of-detection, the timing of the occurrence of the point-of-violation is independent of these two (it can occur before, at the same time, or after any of the above). All these ingredients can help manage system specifications and their interaction with runtime verification engineering. This shall be further elaborated and exemplified in the rest of the paper.

In Section 2 we discuss how, in various runtime verification contexts, if we need to react to a specification being violated, we may need to separately specify the corresponding point-of-reaction. In Section 3 we discuss how this requirement is particularly salient for certain logics, particularly deontic ones. We provide two case studies to corroborate our points in Section 4; we conclude in Section 5.[5]

2 Point and Temporal Properties

To help illustrate the distinction between different types of points-of-interest, we will show the way the specification language may impact the analysis. Throughout this section, we will model an observed trace of a monitored system over an alphabet as a finite sequence of boolean valuations of the alphabet, representing which states hold at each point in time.

Point Properties. Property languages which exclusively care about a single point in time (the present) do not typically make any distinction between the point-of-violation and the point-of-detection.[6] Such point properties, typically encoded as inline assertions, do not look into the history or the future parts of the trace but simply considers the current snapshot of the system. For example, consider the property: *"If a user is logged in, then they may not be blacklisted"*. Then, access to the valuations of states *logged-in* and *blacklisted* (respectively true if the user: (i) is logged in and, (ii) is blacklisted at that point in time) suffices to check that the latter is never true when the former is false. In this way, a violation is detected at the same point in time it occurs, i.e., the point-of-violation and the point-of-detection coincide.

[5] We will not use any formal (syntax-specific) notation in the rest of the paper with the exception of regular expressions and Boolean logic connectors. So, we will for instance write *globally* and *finally* when referring to LTL formulae; similarly we will write *obligation* and *forbidden* instead of using the deontic logic modalities.

[6] In fairness, one can consider situations in which the detection computation may be deferred for later, e.g., during periods of high load on the system. However, we will only assume online and synchronous runtime verification here.

Regular Expressions. Regular expressions have been used as a specification language for runtime verification [14, 13], for instance by specifying when a violation occurs. Consider the following example specification:[7]

$$(\overline{login}^*; login; \overline{logout}^*; logout)^*; \overline{login}^*; write$$

This property matches any trace which ends with *write* outside of a login-logout cycle. By definition, the semantics of such a regular expression would match any finite trace of past observations, i.e., it does not make references to the future (unseen) parts of the trace. Where the point-of-detection occurs in this case is obviously when the *write* occurs. However, where the point-of-violation occurs is debatable, e.g., one can equally argue that it is the *write* at the end that should not have appeared, as one can argue that the previous event should have been a *login*. The regular expression simply states the inadequacy of the trace as a whole, rather than identifying the point in which the cause of the violation occurred. Even more so, when the point-of-reaction should trigger would depend on how the violation will be addressed, e.g., by suppressing the *write* (in which case the point would be the last event in the trace) or by returning an error the next time a *read* takes place (assuming that the out of order *write* would have corrupted the file).

One can handle these issues by writing separate regular expressions for the different points-of-interest, or by annotating the regular expression in a manner to mark where the different points-of-interest would have occurred. Unless we limit the regular expression specification to match only with the shortest prefix, these solutions would need further information since the different points-of-interest specifications may match multiple times, thus making it unclear which point-of-detection would correspond to which point-of-violation and point-of-reaction.

Linear Temporal Logic (LTL). Over the years, LTL has been studied extensively in the context of runtime verification and for this reason several fragments have been considered along with different runtime verification algorithms. It is worth noting that the standard LTL semantics on infinite traces require engineering finite prefix trace matching against an LTL property. Some properties are not monitorable (i.e., either violation, satisfaction or both are not decidable against any finite prefix of a behavioural trace—see for instance [9, 11, 15] and references therein). However, even limiting oneself to a decidable fragment of the logic, one runs into the need for verification algorithms to monitor and verify an LTL formula.

Given the standard semantics of LTL are over an infinite trace, issues of points-of-violation become even more problematic. For a simple safety formula such as $globally(\neg(login \wedge blacklist))$ many would agree that the points-of-violation are the points in time in which the invariant does not hold. However, even if we only consider safety formulae, a property such as $globally(blacklist \Rightarrow$

[7] We use $\overline{a}$ to match any event other than a.

$\neg next(transfer)$) (*"A blacklist event is never immediately followed by a transfer"*) may have different interpretations, as explained in what follows.

On one hand, one may argue that the point-of-violation occurs when *blacklist* happens, to be followed afterwards with a *transfer*, even if the moment of *transfer* would be the point-of-detection. However, one can argue that this formula is logically equivalent (using past time operators) to $globally(transfer \Rightarrow \neg previous(blacklist))$ (*"A transfer event is never immediately preceded by a blacklisting"*), and a natural interpretation of this formula is that the point-of-violation is the moment when *transfer* occurs, despite the fact that *blacklist* had just happened. With this interpretation, the point-of-detection would coincide with the point-of-violation. This highlights the failure of LTL to clearly delineate where violations occurred. An appropriate point-of-reaction would be equally unclear if we were to base our analysis simply on the correctness specification.

In this example, the first interpretation results in a delay of one time unit before detecting that a violation occurred. However, LTL allows for properties with arbitrarily long delays, including unbounded ones: $globally(delete\text{-}user \Rightarrow \neg finally(transfer))$ (*"After deleting a user, no transfers are possible"*). Under an interpretation that takes the moment of *delete-user* to be the point-of-violation (if there is a transfer some time in the future), the runtime verification algorithm may have to wait arbitrarily long before being able to detect this.

Such formulae make certain violation rectification reactions potentially unfeasible. For instance, the use of compensations [5] to 'undo' actions between the point-of-detection and the point-of-violation may result in a memory leak as the accumulated compensation action grows as the delay increases. This highlights the frequent need for explicit annotation of points-of-interest when runtime verifying LTL formulae.

3 On Norms

After reviewing two traditionally common specification formalisms, in this section we turn our attention to deontic logic. This brings two significant differences: firstly, deontic specifications natively include reactive clauses, i.e., what happens when a specification is violated; secondly, such specifications are frequently multi-party and/or multi-locality, such that the question of *when* the violation happened is enriched to also include *by whom* and possibly even *where*. These additions serve to further highlight the decoupling of points-of-interest, not only temporally but also in terms of responsibility and locality, i.e., the violation might occur in country C by party P at time t, but is detected by party Q in country D at time t' while the reaction is carried out by party R in country E at time t''. Indeed, such scenarios are typical in applications such as smart contracts.

For the intent of this paper, a *normative specification* (or *contract*[8]) will be taken to mean a specification of a system consisting of a set of clauses, rules, norms, regulations, or any kind of statement that could be understood as having a *prescriptive meaning*, that is to stipulate obligations, permissions and prohibitions of the different parties involved as well as penalties to be paid in case of violations. We will use the term *normative system* (or *regulatory system*) to refer to a system formed by a set of parties (or agents) whose behaviour, and interaction, should satisfy (or be compliant with) a normative specification.

One standard way to formalise normative systems is by using deontic logics [10, 16]. Many variants exist, but what is common in all those logics is that they are able to represent obligations, permissions (rights) and prohibitions. Many have argued that a deontic logic should be able to talk about potential violation of a normative specification, and thus be able to represent concepts such as contrary-to-duty clauses (CTDs) and contrary-to-prohibition clauses (CTPs)—what new norms come into force when an obligation or a prohibition is violated.

An important aspect of normative systems is the presence of many *parties* (or *agents*), being directly affected by the underlying normative specification. Those parties are then engaged in an interaction which is regulated by the normative system, meaning that each party has its own obligations, permissions and prohibitions, as well as stipulated penalties in case of violations.

It is worth noting that permissions (rights) cannot be violated by the party who can exercise them, but only by the other party if the latter stops the first party from doing so. Indeed, the right of one of the parties of the contractual agreement almost invariably introduces an implicit obligation on the other party to allow the first one to exercise their right. For instance, a (naïve) clause between a bank and a client might stipulate that *"The client has the right to withdraw money at any time from any ATM in the city"*. The client's right would be violated by the bank in case at a given time the client tries to withdraw money when an ATM is out of cash. As the clause was phrased, the bank would thus be liable for the client's unsuccessful attempt to exercise her right.

At face value, obligations and prohibitions act in a manner similar to normal requirements in a logic, e.g., a prohibition from ever performing a particular action x is similar to the LTL formula $\mathit{globally}(\neg x)$. However, there are a number of caveats to this: (i) given the participation of agents, the notion of blame is important in deontic logics; and (ii) in the presence of CTDs and CTPs, violation of an obligation or prohibition may trigger new norms (despite the fact that not violating the top level norm would still be preferable). In view of these notions, the idea of point-of-violation (or contract breach) and point-of-detection become even more crucial.

Consider the following deontic requirement:

$$\mathit{globally}(\mathit{john{:}transfer} \Rightarrow \mathit{next}(\mathit{forbidden}(\mathit{peter{:}deny\text{-}service})))$$

[8] The term *contract* has been extensively abused to mean different things in computer science, few of which share much with the legal sense of the term. In contrast, here it is being used very much in the sense used in law.

(*"Whenever John performs a transfer, Peter is then prohibited from denying the service"*). Unlike the similar LTL formula we saw in the previous section, it is clear that the point-of-violation now lies at the moment Peter denies the service. The past-time dual would be interpreted differently:

$$\mathit{globally}(\mathit{peter{:}deny\text{-}service} \Rightarrow \neg \mathit{previous}(\mathit{permitted}(\mathit{john{:}transfer})))$$

(*"Whenever Peter denies the use of the service, John would have previously not been permitted to perform a transfer"*). Although the two requirements may appear to be equivalent in terms of allowed/permitted/denied actions, there is a difference concerning blame. Indeed, in the latter, the blame seems to be on John for transferring *before* Peter denied the service, and the point-of-violation and detection are now unclear.

The traditional equivalence (duality) between past and future time logics also breaks down in the context of contracts which may, upon agreement between the parties, be cancelled. Consider the two statements below:[9]

Contract 1: *"If John uses the service, then he is obliged to pay in 7 days":*

$$\mathit{globally}(\mathit{john{:}use\text{-}service} \Rightarrow \mathit{next}_7(\mathit{obligation}(\mathit{john{:}pay})))$$

Contract 2: *"John is obliged to pay if he used the service 7 days ago":*

$$\mathit{globally}(\mathit{previous}_7(\mathit{john{:}use\text{-}service}) \Rightarrow \mathit{obligation}(\mathit{john{:}pay}))$$

The two appear to be identical from a temporal logic perspective. However, consider a situation in which (i) John uses the service on day 1; but (ii) the parties decide to call off the contract on day 2. In the case of contract 1, it can be argued that on day 1, an obligation to pay was enacted for day 8, and since the action which led to this took place while the contract was in force, then the obligation remains active. In contrast, in contract 2, the obligation is never enacted since the predicate: *"John used the service seven days ago"* never held during the lifetime of the contract. Clearly, the two have different points-of-interest, and would require different points-of-reaction (and different actions triggered at such points).

Let us consider a more complete example to highlight points-of-interest during execution traces of contracts. Let us assume a scenario in which participants can associate into pools to download music from a repository, with a global contract governing the repository provider and the participants:

1. Every member of the client pool has the right to download up to 3 songs per week.
2. A client pool may not download more than 7 songs per week.
3. Any individual song can only be downloaded up to twice per week.
4. If any of the above are violated, any other download from any member of the pool would incur a payment which must be performed within 3 days by the individual downloading the song.

[9] We assume a time unit granularity of one day.

If John downloads a particular song on Monday, and again on Tuesday, nobody else can download the song for the rest of the week, and any further download may trigger a point-of-violation (and detection). However, the final clause of the contract identifies a contrary-to-prohibition clause, which effectively allows for such a download, except that for the rest of the week, further downloads would come at a cost. A point-of-reaction could thus be triggered if another download is performed within a week, enacting an obligation to pay within three days. Failing to pay would then trigger a point-of-violation.

The same would happen if John downloads more than three songs in a single week. In this case, the repository owner may decide to limit the penalty solely to John (since he is single-handedly responsible for breaking the first clause), in which case, the point-of-reaction would be activated if John tries to download additional songs within a week's time, but not when Mary (also a member of the pool) does so. As this example illustrates, machine-identified points-of-reaction (and reactions) are not easy (or in some cases even possible) to create, and would require decisions taken by the system designers.

4 Use Cases

In this section, we look at two use cases in which different types of points-of-interest arise, and would require appropriate handling by the monitoring system.

4.1 Underwater Robots

We present here a scenario concerning underwater robots.[10]

Let us consider the property in a mission (for an underwater robot) that indicates that the robot should maintain a safe distance of d meters from another one. In case of violation, the robots should restore the safety property, with conformance being required within x seconds. We consider the following action when the robots approach the safe distance between them: the back robot will stop its propellers for x seconds, after which, if property satisfaction is restored (the distance is deemed to be safe, possibly including a safety margin) then the robots will continue with their expected tasks. However, if property satisfaction is not restored, then the back robot sends a signal to the front robot asking it to accelerate and wait y seconds before checking the situation once more.[11]

Monitoring such property requires getting sensor data and observing whether the safe distance has been violated. A question is how often do we need to sample or read data. We would like to minimise the amount of sampling while not missing any critical point (moments of violations). This is a relevant question given the restricted memory and computation capacity of such robots since reducing

[10] A description of this underwater robot scenario is taken from "Monitoring Safety and Reliability of Underwater Robots: A Case Study", appearing in AISOLA'24 proceedings (through personal communication with the authors).

[11] This is a simplified version of the specification, with the aim of illustrating our point.

the required number of sampling (sensor reads) can have a positive effect on the energy consumption.

In principle, if a violation happens, what we care about is that the robot goes back to conformance within x seconds. So if a violation is detected at time t, the next read can happen at $t + x$ (under certain restricted assumptions concerning the speed of the robot and other environmental conditions).

The frequency of the sampling may obviously result in the point-of-detection happening later than the point-of-violation. However, if we leave issues arising from sampling out and focus on the points-of-violation, points-of-detection, and points-of-reaction, a number of interesting observations are to be made. A clear point of violation is when the distance between the two robots is strictly less than d at a given time t. The point-of-detection, however, may happen a bit later (due to a delay in the sensor or in the sampling), let us say at time $t + \delta$ (for a small time δ) which might mean the distance is thus $d + \epsilon$, for a relatively small ϵ.

A point-of-reaction triggers at time $t + \delta + x$ to check the validity of the property then; if the safety distance is still not restored, a message should be sent to the robot ahead, possibly triggering another point-of-reaction at time $t + \delta + x + y$ (to check the property later once again).

This example highlights the existence of many points of interest, which may trigger different conditions and actions from a runtime monitor (even the need of some kind of hierarchical monitor to "activate" other monitors). We do not, however, further explore here how a solution based on monitors would look like.

The complexity of such situations increases as other agents (e.g., a third robot that may cause the first robot's corrective action to put it in a new violation) and specifying such points-of-reaction and points-of-interest becomes ever more important.

4.2 Fraud detection

Properties describing a violation, typically start off by describing expected behaviour followed by a characterisation of when things go wrong. For example, consider once again the following property:

$$(\overline{login}^*; login; \overline{logout}^*; logout)^*; \overline{login}^*; write$$

This property states that a write cannot occur outside a login-logout pair. Therefore, any such write event would constitute the point-of-violation and also point-of-detection. One could then define the corresponding point-of-reaction to happen later—when the next login occurs (e.g., triggering limitations to the rights of the user):

$$(login; \overline{logout}^*; logout)^*; \overline{login}^*; write; \overline{login}^*; login$$

The above example looks simple enough, but when attempting to characterise a fraudulent pattern of events, this usually involves a significantly more complex combination of events. For example consider a simple fraud check related to

card payments: *"If more than three user accounts are simultaneously logged into from the same IP address, then the associated bank cards should be considered suspicious"*. Note that the focus here is to describe a suspicious pattern of events.

Similarly, taking a real-life example from tax fraud detection in Malta [3]:

> *Load the identity card number of employees over 30 years of age and who, for three sequential years, either declared a total income of less than €3000, or there has been a year-on-year decrease in their declared income.*

Once again, the whole specification describes a pattern which can only emerge after three years of observation. These kind of specifications present an interesting case where as soon as a complete pattern emerges, the point-of-violation would be already way in the past. In the first example, the point-of-detection occurs when a user logs in from the same IP into the fourth account. In the second case, the point-of-detection occurs upon the tax return submission of the third year which falls into the pattern. In both cases, the point-of-violation is the start of the pattern: In the first example, the first login from the IP address in question; in the second case, the first of the three consecutive years of tax return submissions.

The point-of-reaction could also present interesting options. In the first example, one could simply mark the accounts involved as suspicious and trigger an action whenever such accounts attempt to perform payments. This could serve useful for the Money Laundering Reporting Officer to collect more information and possibly incriminating evidence. It could also be useful to consider reversing past successful payments originating from the suspicious accounts. In the case of the tax fraud detection, a reaction could be to trigger a manual investigation and write further specifications to monitor the situation more closely, e.g., trigger an alarm if the person with the same identity card number acquires an asset which costs more the €10,000.

5 Conclusions

Past work in the area of runtime verification indirectly acknowledges that points-of-interest do not necessarily occur at the same time: The body of work on runtime enforcement [8] is based on the premise that points-of-detection can occur before points-of-violation. On the other hand, the work on compensation-based monitoring [5] tackles cases where the point-of-detection occurs after the point-of-violation. In previous work [6], we have classified stream monitoring case studies on whether they are *reactive*, meaning that a violation is never detected late, i.e., the point-of-detection never comes after the point-of-violation.

Notwithstanding these works, to the best of our knowledge, the distinction between various points-of-interest has never been laid out, highlighting in particular the complete decoupling of the timing of the point-of-violation from that of the point-of-detection.

In this paper we have presented the need for richer specifications in the context of runtime verification which not only identify when a full trace is a violation, but also identify the point-of-violation, allows us to reason about the point-of-detection, and identify points-of-reaction to act upon the observations. We do not purport to have solutions to this challenge, but we are currently looking at a number of different dimensions of this problem.

Our observation is that the view that specifications are solely meant for system correctness and verification has been accepted as the norm. However, specifications can play other roles, and in this paper we have argued how specifying points-of-interest can help with triggering additional behaviour, invoking corrective action, etc. The contribution is more along the line of advocating the widening of the view of the role of specifications (in keeping with the theme of SpecifyThis) rather than a technical one.

Depending on the context, we see various ways forward as a continuation of this work. In the area of deontic logics, there are also other points-of-interest which could be interesting to look at, e.g., the *points-of-failure*, referring to the point where the consequences of a property violation have side-effects observable by third parties such as users. Note that points-of-failures can of course coincide with points-of-violations.

From a formal perspective, we see the possibility of developing a model to concretely reason about points-of-interest in different specification languages, enabling more straightforward analysis of specifications for different runtime verification applications.

More generally, we aim at a framework with hierarchical monitors to identify/treat different points-of-interest, increasing modularity of the design following the separation-of-concern principle.

References

1. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: VMCAI'04. LNCS, vol. 2937, pp. 44–57. Springer (2004), https://doi.org/10.1007/978-3-540-24622-0_5
2. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. Comput. 20(3), 651–674 (2010), https://doi.org/10.1093/logcom/exn075
3. Calafato, A., Colombo, C., Pace, G.J.: A controlled natural language for tax fraud detection. In: CNL'16. LNCS, vol. 9767, pp. 1–12. Springer (2016), https://doi.org/10.1007/978-3-319-41498-0_1
4. Chen, F., Rosu, G.: Mop: an efficient and generic runtime verification framework. In: OOPSLA'07. pp. 569–588. ACM (2007), https://doi.org/10.1145/1297027.1297069
5. Colombo, C., Pace, G.J., Abela, P.: Safer asynchronous runtime monitoring using compensations. Formal Methods Syst. Des. 41(3), 269–294 (2012), https://doi.org/10.1007/s10703-012-0142-8
6. Colombo, C., Pace, G.J., Camilleri, L., Dimech, C., Farrugia, R.A., Grech, J., Magro, A., Sammut, A.C., Adami, K.Z.: Runtime verification for stream processing

applications. In: ISoLA'16. LNCS, vol. 9953, pp. 400–406 (2016), https://doi.org/10.1007/978-3-319-47169-3_32
7. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: TIME'05. pp. 166–174. IEEE Computer Society (2005), https://doi.org/10.1109/TIME.2005.26
8. Falcone, Y., Mounier, L., Fernandez, J., Richier, J.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. Formal Methods Syst. Des. 38(3), 223–262 (2011), https://doi.org/10.1007/s10703-011-0114-4
9. Havelund, K., Peled, D.: Runtime verification: From propositional to first-order temporal logic. In: RV'18. LNCS, vol. 11237, pp. 90–112. Springer (2018), https://doi.org/10.1007/978-3-030-03769-7_7
10. Mally, E.: Grundgesetze des Sollens. Elemente fer Logik des Willens. Graz: Leuschner & Lubensky (1926)
11. Pnueli, A., Zaks, A.: PSL model checking and run-time verification via testers. In: FM'06. LNCS, vol. 4085, pp. 573–586. Springer (2006), https://doi.org/10.1007/11813040_38
12. Rosu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Autom. Softw. Eng. 12(2), 151–197 (2005), https://doi.org/10.1007/s10515-005-6205-y
13. Sammapun, U., Easwaran, A., Lee, I., Sokolsky, O.: Simulation of simultaneous events in regular expressions for run-time verification. In: RV'04. ENTCS, vol. 113, pp. 123–143. Elsevier (2004), https://doi.org/10.1016/J.ENTCS.2004.01.030
14. Sen, K., Rosu, G.: Generating optimal monitors for extended regular expressions. In: RV'03. ENTCS, vol. 89, pp. 226–245. Elsevier (2003), https://doi.org/10.1016/S1571-0661(04)81051-X
15. Stucki, S., Sánchez, C., Schneider, G., Bonakdarpour, B.: Gray-box monitoring of hyperproperties. In: FM'19. LNCS, vol. 11800, pp. 406–424. Springer (2019), https://doi.org/10.1007/978-3-030-30942-8_25
16. Wright, G.H.V.: Deontic logic. Mind 60, 1–15 (1951), https://doi.org/10.1093/mind/LX.237.1