# A Foundation for Runtime Monitoring

# A Foundation for Runtime Monitoring*

Adrian Francalanza[1], Luca Aceto[2], Antonis Achilleos[2], Duncan Paul Attard[1,2], Ian Cassar[1,2], Dario Della Monica[3,4], and Anna Ingólfsdóttir[2]

[1] Department of Computer Science, University of Malta, Msida, Malta
[2] School of Computer Science, Reykjavík University, Iceland
[3] Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Spain
[4] Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione, Università "Federico II" di Napoli, Italy

**Abstract.** Runtime Verification is a lightweight technique that complements other verification methods in an effort to ensure software correctness. The technique poses novel questions to software engineers: it is not easy to identify which specifications are amenable to runtime monitoring, nor is it clear which monitors effect the required runtime analysis correctly. This exposition targets a foundational understanding of these questions. Particularly, it considers an expressive specification logic (a syntactic variant of the modal $\mu$-calculus) that is agnostic of the verification method used, together with an elemental framework providing an operational semantics for the runtime analysis performed by monitors. The correspondence between the property satisfactions in the logic on the one hand, and the verdicts reached by the monitors performing the analysis on the other, is a central theme of the study. Such a correspondence underpins the concept of monitorability, used to identify the subsets of the logic that can be adequately monitored for by RV. Another theme of the study is that of understanding what should be expected of a monitor in order for the verification process to be correct. We show how the monitor framework considered can constitute a basis whereby various notions of monitor correctness may be defined and investigated.

## 0 Introduction

Runtime Verification (RV) [35] is a lightweight verification technique that checks whether the System Under Scrutiny (SUS) satisfies a correctness property by analysing its *current execution*. It has its origins in model checking, as a more scalable (yet still formal) approach to program verification where state explosion problems (which are inherent to model checking) are mitigated [34,35]. RV is often used to *complement* other verification techniques such as theorem proving, model checking and testing, in a multi-pronged approach towards ensuring system correctness [5,4,21,3]. The technique has fostered a number of verification

* This work was supported by the project "TheoFoMon: Theoretical Foundations for Monitorability" (nr.163406-051) of the Icelandic Research Fund and the Marie Curie INDAM-COFUND-2012 Outgoing Fellowship.

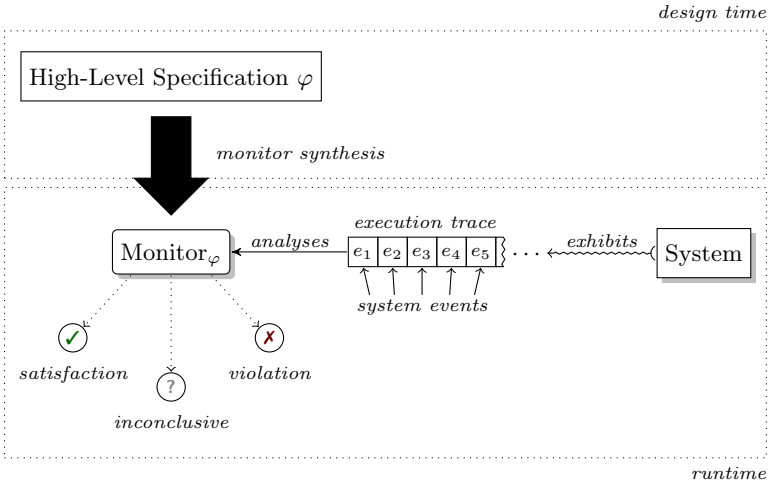


**Fig. 1.** Runtime monitor synthesis and operational set-up.

tools such as [30,12,19,10,16,37,21,18,39,27,20,6,38], to name but a few. It has also been used for a variety of applications, ranging from checking the executions of the NASA Mars Rover software [12] and other autonomous research vehicles [28], to more mundane tasks such as checking for rule violations in financial systems [8], video games [40] and electronic examinations [29].

At its core, RV generally assumes a *logic* describing the correctness specifications that are expected to be satisfied by the SUS. From these specifications, programs called *monitors* are generated and *instrumented* to run with the SUS, so as to analyse its current execution (expressed as a trace of events) and infer any system violations or satisfactions w.r.t. these specifications (see Figure 1). Violation and satisfaction *verdicts* are typically considered to be definite, *i.e.,* cannot be retracted upon observing further system events. Monitors are themselves computational entities that incur runtime costs, and considerable amounts of effort in RV is devoted to the engineering necessary to keep runtime overheads at feasible levels. Yet, a RV set-up such as the one depicted in Figure 1 raises additional questions that warrant equal consideration.

**We need to talk about monitors.** The *expressiveness* of the verification technique w.r.t. the correctness properties that can be described by the logic—an attribute often termed as *monitorability*—should be a central question in RV. In fact, specification logics that are not necessarily wedded to the RV technique may express properties that *cannot* be monitored for at runtime. Particularly, certain aspects of a chosen specification logic can potentially reduce the analytical capabilities of the monitors, such as the ability to express properties describing *multiple or infinite executions* of a system. In such cases, the (finite) trace exhibited by the running system (see Figure 1) may *not* contain sufficient execution

information so as to enable the monitor to determine whether a property under analysis is violated or satisfied by the SUS. Therefore, being able to identify which properties are monitorable is essential for devising an effective strategy in a multi-pronged verification approach that tailors the verification method used to each correctness specification.

Another fundamental question raised by RV concerns monitor *correctness*. Generally, monitors are considered to be part of the *trusted computing base*, and are thus expected to exhibit correct behaviour. When this is not the case, erroneous monitors could invalidate any runtime analysis performed, irrespective of the low overheads a monitor may brandish. On top of that, what is actually expected of these monitors is seldom defined in precise terms, making it hard to ascertain monitor correctness. For instance, in a typical RV set-up such as that of Figure 1, one would expect detection soundness, whereby *any* rejections (resp. acceptances) flagged by the monitor imply that the system indeed violates (resp. satisfies) the property being monitored. Other settings may also stipulate detection *completeness*, by which *all* property violations (resp. satisfactions) are flagged accordingly by the monitor observing the SUS. One may also require attributes that are independent of the specification logic chosen, such as passivity (*i.e.,* the absence of monitor interference with execution of the SUS), and determinism (*i.e.,* whether the monitor consistently yields the same verification outcome for the same observations).

This paper sheds light on one possible approach for addressing these questions. In particular, it showcases the work embarked upon in *Theoretical Foundations for Monitorability* (TheoFoMon), a three-year project funded by the Icelandic Research Fund with the aim of investigating the expressiveness and correctness issues outlined above. To maintain a general approach, the aforementioned questions are studied in the context of the Hennessy-Milner Logic with recursion ($\mu$HML) [33], a reformulation of the expressive and well-studied modal $\mu$-calculus [32]—the logic is agnostic of the verification technique used, and can embed widely used temporal logics such as CTL$^*$ [17,9]. Labelled Transition Systems (LTSs) [2], formalisms that have been used to describe a vast range of computational phenomena, are employed to abstractly model the behaviour of systems and monitors in terms of graphs. This generality serves two goals. On the one hand, the LTS operational formalism abstracts away instance details of specific RV settings, facilitating the distillation of the core concepts at play. On the other hand, the considerable expressiveness of the logic immediately extends any findings and observations to other logics that can be embedded within it.

**Paper Overview.** The preliminaries in Section 1 present the basics for modelling systems and provide an introduction to our touchstone logic. Section 2 presents our monitor and instrumentation models. Section 3 links the properties specified by the logic and the verdicts reached by monitors, whereas Section 4 discusses notions of monitor correctness. Section 5 surveys extensions on the monitorability and correctness results presented, and Section 6 concludes.

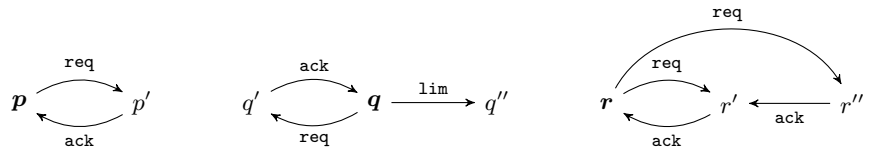


**Fig. 2.** The LTS depicting the behaviour of three server implementations $p$, $q$ and $r$.

## 1 Preliminaries: Groundhog Day

We provide a brief outline of the main technical machinery used by our exposition, namely, the system model and the specification logic. Interested readers are encouraged to consult [33,2] for more details.

**The model.** LTSs are directed graphs with labelled edges modelling the possible behaviours that can be exhibited by an executing system. Formally, a LTS consists of the triple $\langle \textsc{Sys}, (\textsc{Act} \cup \{\tau\}), \longrightarrow \rangle$ comprised of:

- a set of system states $p, q, r \in \textsc{Sys}$ (every system state may also be used to denote a system that starts executing from that state),
- a set of visible actions $\alpha \in \textsc{Act}$ and a distinguished silent action $\tau$, where $\tau \notin \textsc{Act}$ and $\mu$ ranges over $(\textsc{Act} \cup \{\tau\})$, and finally,
- a ternary transition relation between states labelled by actions; we write $p \xrightarrow{\mu} q$ in lieu of $\langle p, \mu, q \rangle \in \longrightarrow$.

The notation $p \not\xrightarrow{\mu}$ is written when $p \xrightarrow{\mu} q$ for *no* system state $q$. Additionally, $p \Longrightarrow q$ denotes a sequence of silent actions $p(\xrightarrow{\tau})^* q$ from state $p$ to $q$, whereas $p \overset{\alpha}{\Longrightarrow} q$, is written in place of $p \Longrightarrow \cdot \xrightarrow{\alpha} \cdot \Longrightarrow q$, to represent a visible action $\alpha$ that may be padded by preceding and succeeding silent actions. We occasionally use the notation $\mathbf{0}$ to refer to a system (state) that is deadlocked and exhibits *no* actions (not even silent ones). We let traces, $t, u \in \textsc{Act}^*$, range over sequences of *visible* actions and write sequences of transitions $p \overset{\alpha_1}{\Longrightarrow} \ldots \overset{\alpha_n}{\Longrightarrow} p_n$ as $p \overset{t}{\Longrightarrow} p_n$, where $t = \alpha_1, \ldots, \alpha_n$. As usual, $\epsilon$ denotes the empty trace.

*Example 1.* The directed graph in Figure 2 depicts a LTS describing the implementation of three servers whose events are represented by the set of visible actions $\textsc{Act} = \{\texttt{req}, \texttt{ack}, \texttt{lim}\}$. State $p$ shows the simplest possible implementation of a server that receives (`req`) and acknowledges (`ack`) client requests repeatedly. State $q$ denotes an extension on this implementation that may reach a termination limit (transition `lim`) after a number or serviced requests (*i.e.,* `req` followed by `ack`). Finally state $r$ represents an unpredictable server implementation that occasionally acknowledges a preceding client request twice. ■

**Syntax**

$$
\begin{array}{rlll}
\varphi, \phi \in \mu\text{HML} ::= & \mathbf{ff} \quad \text{(falsity)} & \mid & \mathbf{tt} \quad \text{(truth)} \\
\mid & \varphi \wedge \phi \quad \text{(conjunction)} & \mid & \varphi \vee \phi \quad \text{(disjunction)} \\
\mid & [\alpha]\varphi \quad \text{(necessity)} & \mid & \langle\alpha\rangle\varphi \quad \text{(possibility)} \\
\mid & \mathbf{max}\, X.\varphi \quad \text{(max. fixpoint)} & \mid & \mathbf{min}\, X.\varphi \quad \text{(min. fixpoint)} \\
\mid & X \quad \text{(recursive variable)} & &
\end{array}
$$

**Semantics**

$$
\begin{array}{lll}
[\![\mathbf{ff}, \rho]\!] & \stackrel{\text{def}}{=} \emptyset & [\![\mathbf{tt}, \rho]\!] \stackrel{\text{def}}{=} \textsc{Sys} \\
[\![\varphi \wedge \phi, \rho]\!] & \stackrel{\text{def}}{=} [\![\varphi, \rho]\!] \cap [\![\phi, \rho]\!] & [\![\varphi \vee \phi, \rho]\!] \stackrel{\text{def}}{=} [\![\varphi, \rho]\!] \cup [\![\phi, \rho]\!] \\
[\![[\alpha]\varphi, \rho]\!] & \stackrel{\text{def}}{=} \left\{ p \mid \forall p'.\, p \stackrel{\alpha}{\Longrightarrow} p' \text{ implies } p' \in [\![\varphi, \rho]\!] \right\} & \\
[\![\langle\alpha\rangle\varphi, \rho]\!] & \stackrel{\text{def}}{=} \left\{ p \mid \exists p'.\, p \stackrel{\alpha}{\Longrightarrow} p' \text{ and } p' \in [\![\varphi, \rho]\!] \right\} & \\
[\![\mathbf{max}\, X.\varphi, \rho]\!] & \stackrel{\text{def}}{=} \bigcup \{ S \mid S \subseteq [\![\varphi, \rho[X \mapsto S]]\!] \} & \\
[\![\mathbf{min}\, X.\varphi, \rho]\!] & \stackrel{\text{def}}{=} \bigcap \{ S \mid [\![\varphi, \rho[X \mapsto S]]\!] \subseteq S \} & \\
[\![X, \rho]\!] & \stackrel{\text{def}}{=} \rho(X) &
\end{array}
$$

**Fig. 3.** The syntax and semantics of $\mu$HML.

**The logic.** $\mu$HML [33,2] is a branching-time logic that can be used to specify correctness properties over systems modelled in terms of LTSs. Its syntax, given in Figure 3, assumes a countable set of logical variables $X, Y \in \textsc{LVar}$, thereby allowing formulae to recursively express largest and least fixpoints using $\mathbf{max}\, X.\varphi$ and $\mathbf{min}\, X.\varphi$ resp.; these constructs bind free instances of the variable $X$ in $\varphi$ and induce the usual notions of free and closed formulae—we work up to alpha-conversion of bound variables. In addition to the standard constructs for truth, falsity, conjunction and disjunction, the $\mu$HML syntax includes the necessity and possibility modalities, one of main distinctive features of the logic.

The semantics of $\mu$HML is defined in terms of the function mapping formulae $\varphi$ to the set of LTS states $S \subseteq \textsc{Sys}$ satisfying them. Figure 3 describes the semantics for open and closed formulae, and uses a map $\rho \in \textsc{LVar} \rightharpoonup 2^{\textsc{Sys}}$ from variables to sets of system states to enable an inductive definition on the structure of the formula $\varphi$. The formula $\mathbf{tt}$ is satisfied by all processes, while $\mathbf{ff}$ is satisfied by none; conjunctions and disjunctions bear the standard set-theoretic meaning of intersection and union. Necessity formulae $[\alpha]\varphi$ state that *for all* system executions producing event $\alpha$ (possibly none), the subsequent system state must then satisfy $\varphi$ (*i.e.,* $\forall p'$, $p \stackrel{\alpha}{\Longrightarrow} p'$ implies $p' \in [\![\varphi, \rho]\!]$ must hold). Possibility formulae $\langle\alpha\rangle\varphi$ require the existence of *at least one* system execution with event $\alpha$ whereby the subsequent state then satisfies $\varphi$ (*i.e.,* $\exists p'$, $p \stackrel{\alpha}{\Longrightarrow} p'$ and $p' \in [\![\varphi, \rho]\!]$ must hold). The recursive formulae $\mathbf{max}\, X.\varphi$ and $\mathbf{min}\, X.\varphi$ are resp. satisfied by the largest and least set of system states satisfying $\varphi$. The semantics of recursive

variables $X$ w.r.t. an environment instance $\rho$ is given by the mapping of $X$ in $\rho$, *i.e.,* the set of processes associated with $X$. *Closed* formulae (*i.e.,* formulae containing no free variables) are interpreted *independently* of the environment $\rho$, and the shorthand $[\![\varphi]\!]$ is used to denote $[\![\varphi, \rho]\!]$, *i.e.,* the set of system states in Sys that satisfy $\varphi$. In view of this, we say that a system (state) $p$ satisfies some closed formula $\varphi$ whenever $p \in [\![\varphi]\!]$, and conversely, that it violates $\varphi$ whenever $p \notin [\![\varphi]\!]$. We highlight two basic formulae that are used pervasively in $\mu$HML: $\langle\alpha\rangle\mathbf{tt}$ describes systems that *can* produce action $\alpha$, while $[\alpha]\mathbf{ff}$ describes systems that *cannot* produce action $\alpha$. Note also that $[\alpha]\mathbf{tt}$ is semantically equivalent to $\mathbf{tt}$ whereas $\langle\alpha\rangle\mathbf{ff}$ equates to $\mathbf{ff}$.

*Example 2.* Recall the server implementations $p$, $q$ and $r$ depicted in Figure 2.

$$\varphi_1 = \langle\mathtt{req}\rangle\langle\mathtt{ack}\rangle\langle\mathtt{req}\rangle\mathbf{tt} \qquad \varphi_2 = [\mathtt{lim}][\mathtt{req}]\mathbf{ff}$$
$$\varphi_3 = [\mathtt{req}][\mathtt{ack}]\langle\mathtt{req}\rangle\mathbf{tt} \qquad \varphi_4 = [\mathtt{req}][\mathtt{ack}]\langle\mathtt{req}\rangle\mathbf{tt} \wedge \langle\mathtt{lim}\rangle\mathbf{tt}$$
$$\varphi_5 = \mathbf{max}\, X.\big([\mathtt{req}]([\mathtt{ack}]X \wedge [\mathtt{ack}][\mathtt{ack}]\mathbf{ff})\big)$$
$$\varphi_6 = \mathbf{min}\, X.(\langle\mathtt{req}\rangle\langle\mathtt{ack}\rangle X \vee \langle\mathtt{lim}\rangle\mathbf{tt})$$

Formula $\varphi_1$ describes systems that *can* produce a `req` after *some* serviced request (*i.e.,* a `req` followed by a `ack`); all server implementations $p, q$ and $r$ satisfy this property. Formula $\varphi_2$ states that *whenever* a system produces the event `lim`, it *cannot* produce any `req` actions. Again all three implementations satisfy this property where, in particular, $p$ and $r$ satisfy this trivially since both never produce a `lim` event. Formula $\varphi_3$ strengthens property $\varphi_1$ by requiring that a system *can* produce a `req` after *any* serviced request. While $p$ and $q$ satisfy this requirement, implementation $r$ violates this property at any time it (non-deterministically) transitions to state $r''$. Formula $\varphi_4$ strengthens this property further, by requiring the implementation to be capable of producing the `lim` event; only $q$ satisfies this property.

Formula $\varphi_5$ specifies a (recursive) safety property that prohibits a system from producing duplicate acknowledgements in answer to client requests after *any number* of serviced requests. System $r$ violates $\varphi_5$ via any trace in the regular language $(\mathtt{req.ack})^+.\mathtt{ack}$. Finally, $\varphi_6$ describes systems that *can* reach a service limit after a number (possibly zero) of serviced requests. System $q$ satisfies $\varphi_6$ immediately, as opposed to $p$ and $r$, which never reach such a state after *any* number of serviced requests. We note that if the minimal fixpoint recursion operator in $\varphi_6$ is substituted with a maximal fixpoint, *i.e.,* $\mathbf{max}\, X.(\langle\mathtt{req}\rangle\langle\mathtt{ack}\rangle X \vee \langle\mathtt{lim}\rangle\mathbf{tt})$, implementations $p$ and $r$ would also satisfy it via the *infinite* sequence of events `req.ack.req.ack`... ■

## 2 Dial M for Monitor

In [25,26], monitors are also perceived as LTSs, expressed through the syntax of Figure 5. It consists of three *verdict* constructs, **yes**, **no** and **end**, resp. denoting acceptance, rejection and termination (*i.e.,* an inconclusive outcome). The

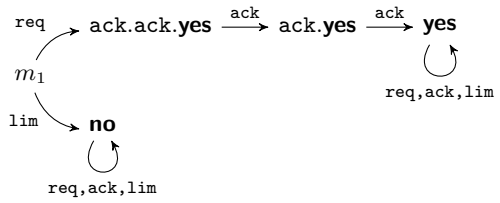


**Fig. 4.** The LTS representation for monitor $m_1 = (\texttt{req}.\texttt{ack}.\texttt{ack}.\textbf{yes} + \texttt{lim}.\textbf{no})$.

syntax also includes action $\alpha$ prefixing, mutually-exclusive (external) choice and recursion, denoted by $\textbf{rec}\, x.m$, acting as a binder for the recursion variables $x$ in $m$. All recursive monitors are assumed to be guarded, meaning that all occurrences of bound variables occur under the context of an action prefix. In the sequel, we assume *closed* monitor terms, where all variable occurrences are bound.

Every monitor term may be given a LTS semantics via the transition rules shown in Figure 5. The rules are fairly standard: for example, $m_1 + m_2 \xrightarrow{\mu} m_3$ (for some $m_3$) if either $m_1 \xrightarrow{\mu} m_3$ or $m_2 \xrightarrow{\mu} m_3$ (by rules SelL or SelR). The only transition rule worth drawing attention to, Ver, specifies that verdicts $v$ may transition with *any* $\alpha \in$ Act to return to the same state, modelling the assumed requirement from Figure 1 that verdicts are *irrevocable*.

*Example 3.* Monitor $m_1 = (\texttt{req}.\texttt{ack}.\texttt{ack}.\textbf{yes} + \texttt{lim}.\textbf{no})$ can be represented diagrammatically via the LTS depicted in Figure 5. This LTS is derived using the monitor dynamics from Figure 5, by applying the rules that match the edge label and the *structure* of the term under consideration. Every term that results from each rule application essentially represents a LTS *state*. For example, the edge labelled `lim` from node $m_1$ to node **no** in Figure 4 follows from the transition

$$\texttt{req}.\texttt{ack}.\texttt{ack}.\textbf{yes} + \texttt{lim}.\textbf{no} \xrightarrow{\texttt{lim}} \textbf{no}$$

derived by applying rule SelR and rule Act. From the LTS in Figure 4, we can see that monitor $m_1$ reaches an acceptance verdict whenever it analyses the sequence of events `req.ack.ack`, and a rejection verdict when the single event `lim` is analysed. ■

In [26], a system $p$ can be *instrumented* with a monitor $m$ (referred to hereafter as a *monitored system* and denoted as $m \triangleleft s$) by composing their respective LTSs. The semantics of $m \triangleleft s$ is defined by the instrumentation rules in Figure 5. We highlight the generality of the instrumentation relation $\triangleleft$. It is parametric w.r.t. the system and monitor abstract LTSs and is largely independent of their specific syntax: it only requires the monitor LTS to contain an inconclusive (persistent) verdict state, **end**. Instrumentation is *asymmetric*, and the monitored system transitions with an observable event only when the system is able to exhibit said event. The suggestive symbol $\triangleleft$ alludes to this unidirectional composition, indicating that trace events flow from the system into the monitor.

**Syntax**

$$m, n \in \textsc{Mon} ::= v \mid \alpha.m \mid m + n \mid \textbf{rec}\, x.m \mid x$$
$$v, u \in \textsc{Verd} ::= \textbf{end} \mid \textbf{no} \mid \textbf{yes}$$

**Dynamics**

$$\textsc{Act}\ \frac{}{\alpha.m \xrightarrow{\alpha} m} \qquad \textsc{Rec}\ \frac{}{\textbf{rec}\, x.m \xrightarrow{\tau} m[\textbf{rec}\, x.m/x]}$$

$$\textsc{SelL}\ \frac{m \xrightarrow{\mu} m'}{m + n \xrightarrow{\mu} m'} \qquad \textsc{SelR}\ \frac{n \xrightarrow{\mu} n'}{m + n \xrightarrow{\mu} n'} \qquad \textsc{Ver}\ \frac{}{v \xrightarrow{\alpha} v}$$

**Instrumentation**

$$\textsc{Mon}\ \frac{p \xrightarrow{\alpha} p' \quad m \xrightarrow{\alpha} m'}{m \triangleleft p \xrightarrow{\alpha} m' \triangleleft p'} \qquad \textsc{Ter}\ \frac{p \xrightarrow{\alpha} p' \quad m \not\xrightarrow{\alpha} \quad m \not\xrightarrow{\tau}}{m \triangleleft p \xrightarrow{\alpha} \textbf{end} \triangleleft p'}$$

$$\textsc{AsS}\ \frac{p \xrightarrow{\tau} p'}{m \triangleleft p \xrightarrow{\tau} m \triangleleft p'} \qquad \textsc{AsM}\ \frac{m \xrightarrow{\tau} m'}{m \triangleleft p \xrightarrow{\tau} m' \triangleleft p}$$

**Fig. 5.** Monitors and instrumentation.

The monitor is in this sense *passive* as it does not interact with the system, but transitions in tandem with it according to the rules in Figure 5. When the system exhibits an observable event that can be analysed by the monitor, the two synchronise and transition in lockstep according to their respective LTSs via the rule Mon. When the monitor *cannot* analyse the aforementioned event,[5] and is it *not* able to transition internally to a state that permits it to do so (*i.e.,* it is already stable, $m \not\xrightarrow{\tau}$), the system is not blocked by the instrumentation. Instead, it is allowed to transition, whereas the monitor is aborted to the inconclusive verdict **end**, as per rule Ter. The system-monitor synchronisation is limited to visible actions, and both system and monitor can transition independently w.r.t. their own internal $\tau$-action (rules AsS and AsM).

*Example 4.* Figure 6 depicts the LTS of the monitored system $m_1 \triangleleft r$ that results from the composition of system $r$ from Figure 2 with monitor $m_1$ from Figure 4. Any verdict that $m_1$ arrives at is subject to the execution path that $r$ decides to follow at runtime. In Figure 6, an acceptance can never be reached since $r$ does not produce event `lim`. Furthermore, when event `req` is exhibited by $r$, the monitored system may non-deterministically transition to either `ack`.`ack`.**yes**$\triangleleft r'$ or `ack`.`ack`.**yes**$\triangleleft r''$ (cases A and B in Figure 6)—this impinges on whether $m_1$ reaches an acceptance verdict or not.

For case A, state $r'$ generates an `ack` event followed by `req`; the monitor at this stage, `ack`.**yes**, is not expecting a `req` event, but `ack` instead. It therefore

[5] This may be due to event knowledge gaps from the instrumentation-side or knowledge disagreements between the monitors and the instrumentation [11].

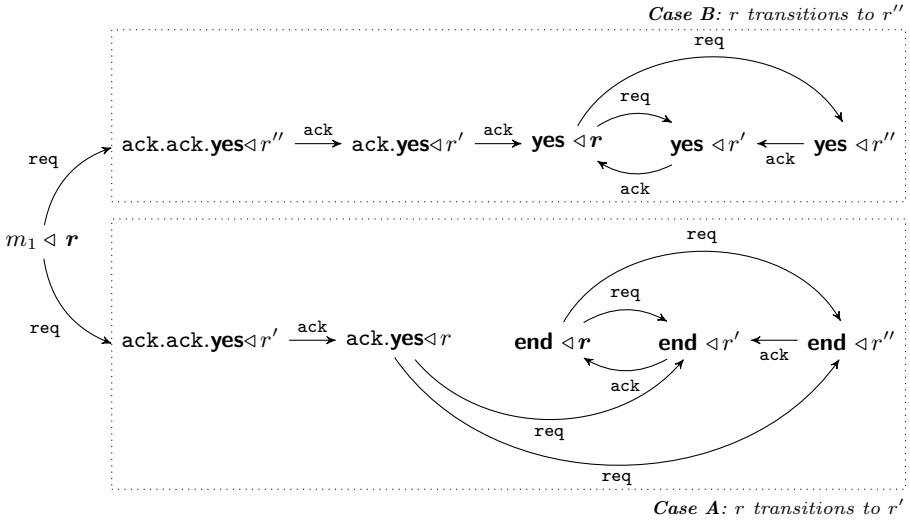


**Fig. 6.** The LTS depicting the behaviour of the monitored system $m \triangleleft r$.

aborts monitoring, reaching the inconclusive verdict **end** using the instrumentation rule Ter (Figure 5). If instead, the monitored system $m_1 \triangleleft r$ transitions to $\mathtt{ack}.\mathtt{ack}.\mathbf{yes}\triangleleft r''$ (case B), it then reaches an acceptance verdict after analysing two consecutive `ack` events. In either outcome, the monitor verdict is preserved once it is reached via rule Ver. ■

## 3 Sense and Monitorability

To understand monitorability in our setting, we need to relate acceptances (**yes**) and rejections (**no**) reached by a monitor $m$ when monitoring a system $p$ with satisfactions ($p \in [\![\varphi]\!]$) and violations ($p \notin [\![\varphi]\!]$) for that system w.r.t. some *closed* $\mu$HML formula $\varphi$. This will, in turn, allow us to determine when a monitor $m$ *represents* (in some precise sense) a property $\varphi$.

*Example 5.* Consider the simple monitor $m_2 = \mathtt{lim}.\mathbf{no}$ which *rejects all* the systems that produce the event `lim`. Since any such system $p$ would necessarily violate the property $\varphi_7 = [\mathtt{lim}]\mathbf{ff}$, *i.e.,* $p \notin [\![[\mathtt{lim}]\mathbf{ff}]\!]$, there exists a tight correspondence between any system rejected by $m_2$ and the systems violating $\varphi_7$. By contrast, the more elaborate monitor $m_3 = \mathbf{rec}\, x.(\mathtt{req}.\mathtt{ack}.x + \mathtt{lim}.\mathbf{yes})$ reaches the *acceptance* verdict for systems that exhibit a trace consisting of a sequence of serviced requests $(\mathtt{req}.\mathtt{ack})^*$ followed by the event `lim`. It turns out that this monitor can *only* reach an acceptance for systems that satisfy the property $\varphi_6 = \mathbf{min}\, X.(\langle\mathtt{req}\rangle\langle\mathtt{ack}\rangle X \vee \langle\mathtt{lim}\rangle\mathbf{tt})$ from Example 2. Stated otherwise, there is a correspondence between the systems satisfying $\varphi_6$, *i.e.,* $p \in [\![\varphi_6]\!]$, and those accepted by monitor $m_3$. ■

However such correspondences are not so clear for certain monitors.

*Example 6.* Monitor $m_1$ from Example 3 reaches an acceptance verdict when composed with a system $p_\bot$ that exhibits the trace `req.ack.ack`, and a rejection verdict for the *same* system $p_\bot$ when it exhibits the trace `lim` (*i.e.,* $m_1 \triangleleft p_\bot \xRightarrow{\texttt{req.ack.ack}} \textbf{yes} \triangleleft \textbf{0}$ and $m_1 \triangleleft p_\bot \xRightarrow{\texttt{lim}} \textbf{no} \triangleleft \textbf{0}$). If we associate $m_1$ with a correctness property such as $\varphi_8 = \langle \texttt{req} \rangle \langle \texttt{ack} \rangle \langle \texttt{ack} \rangle \textbf{tt} \wedge [\texttt{lim}]\textbf{ff}$ and attempt to link acceptances to satisfactions and rejections to violations (as in Example 5) we end up with a logical contradiction, namely that $p_\bot \in [\![\varphi_8]\!]$ and $p_\bot \notin [\![\varphi_8]\!]$. ■

A correspondence between monitor judgements and $\mu$HML properties that relies on the following predicates is established in [26]:

$$\mathsf{acc}(p, m) \stackrel{\text{def}}{=} \exists t, p' \text{ such that } m \triangleleft p \xRightarrow{t} \textbf{yes} \triangleleft p' \qquad \text{(acceptance)}$$

$$\mathsf{rej}(p, m) \stackrel{\text{def}}{=} \exists t, p' \text{ such that } m \triangleleft p \xRightarrow{t} \textbf{no} \triangleleft p' \qquad \text{(rejection)}$$

**Definition 1 (Sound Monitoring).** *A monitor $m$ monitors* soundly *for the property represented by the formula $\varphi$, denoted as $\mathsf{smon}(m, \varphi)$, whenever for* every *system $p \in$ Sys the following conditions hold: (i) $\mathsf{acc}(p, m)$ implies $p \in [\![\varphi]\!]$ , (ii) $\mathsf{rej}(p, m)$ implies $p \notin [\![\varphi]\!]$.* ■

Definition 1 universally quantifies over all system states $p$ satisfying a formula $\varphi$ that is soundly monitored by a monitor $m$ (this may, in general, be an infinite set). It also rules out contradicting monitor verdicts. For whenever the predicate $\mathsf{smon}(m, \varphi)$ holds for some monitor $m$ and formula $\varphi$, and there exists some system $p$ where $\mathsf{acc}(p, m)$, it must be the case that $p \in [\![\varphi]\!]$ by Definition 1. Thus, from the logical satisfaction definition we have $\neg(p \notin [\![\varphi]\!])$, and by the contrapositive of Definition 1, we must also have $\neg\mathsf{rej}(p, m)$. A symmetric argument also applies for any system $p$ where $\mathsf{rej}(p, m)$, from which $\neg\mathsf{acc}(p, m)$ follows. Sound monitoring is arguably the least requirement for relating a monitor with a logical property. Further to this, the obvious additional stipulation would be to ask for the dual of Definition 1, namely *complete monitoring* for $m$ and $\varphi$. Intuitively, this would state that for all $p$, $p \in [\![\varphi]\!]$ implies $\mathsf{acc}(p, m)$, and also that $p \notin [\![\varphi]\!]$ implies $\mathsf{rej}(p, m)$. However, such a demand turns out to be too stringent for a large part of the logic presented in Figure 3.

*Example 7.* Consider the core basic formula $\langle \alpha \rangle \textbf{tt}$, describing processes that can perform action $\alpha$. One could demonstrate that the simple monitor $\alpha.\textbf{yes}$ satisfies the condition that, for all systems $p$, $p \in [\![\varphi]\!]$ implies $\mathsf{acc}(p, m)$. However, for this formula, *no* sound monitor exists satisfying the condition that whenever $p \notin [\![\varphi]\!]$ then $\mathsf{rej}(p, m)$. For assume that one such (sound) monitor $m$ exists satisfying this condition. Since $\textbf{0} \notin [\![\langle \alpha \rangle \textbf{tt}]\!]$, then $\mathsf{rej}(\textbf{0}, m)$ follows by our assumption. This means that this particular monitor can reach a rejection *without* needing to observe any actions (*i.e.,* since $\textbf{0}$ does not produce any actions, we have $m \Longrightarrow \textbf{no}$). Such a statement would, in turn, imply that $\mathsf{rej}(\alpha.\textbf{0}, m)$ also holds (because $m$ is able to reach a rejection verdict for *any* system) although, clearly, $\alpha.\textbf{0} \in$

$[\![\langle\alpha\rangle\mathbf{tt}]\!]$. This makes $m$ unsound, contradicting our initial assumption that $m$ was sound. ■

A dual argument to that of Example 7 can be carried out for another core basic formula in $\mu$HML, namely $[\alpha]\mathbf{ff}$, describing the property of not being able to produce action $\alpha$. Although there are sound monitors $m$ satisfying the condition that for all systems $p$, if $p \notin [\![[\alpha]\mathbf{ff}]\!]$ then $\mathsf{rej}(p, m)$, there are *none* that also satisfy the condition that for any system $p$, if $p \in [\![[\alpha]\mathbf{ff}]\!]$ then $\mathsf{acc}(p, m)$.

The counterarguments posed for the core formulae $\langle\alpha\rangle\mathbf{tt}$ and $[\alpha]\mathbf{ff}$ are enough evidence to convince us that requiring complete monitoring would exclude a large number of useful formulae in $\mu$HML. In fact, the complete monitoring requirement would severely limit correspondence to formulae that are semantically equivalent to $\mathbf{tt}$ and $\mathbf{ff}$ only—admittedly, this would not be very useful. In view of this, we define a weaker form of completeness where we require completeness w.r.t. either logical satisfactions or violations, but not both.

**Definition 2 (Partially-Complete Monitoring).** *A monitor $m$ can monitor for a property $\varphi$ in a satifaction-complete, $\mathsf{scmon}(m, \varphi)$, or a violation-complete, $\mathsf{vcmon}(m, \varphi)$, manner. These are defined as follows:*

$$\mathsf{scmon}(m, \varphi) \stackrel{def}{=} \forall p.\, p \in [\![\varphi]\!] \text{ implies } \mathsf{acc}(p, m) \qquad \text{(satisfaction-complete)}$$

$$\mathsf{vcmon}(m, \varphi) \stackrel{def}{=} \forall p.\, p \notin [\![\varphi]\!] \text{ implies } \mathsf{rej}(p, m) \qquad \text{(violation-complete)}$$

*A monitor $m$ monitors for formula $\varphi$ in a partially-complete manner, denoted as $\mathsf{cmon}(m, \varphi)$, when either $\mathsf{scmon}(m, \varphi)$ or $\mathsf{vcmon}(m, \varphi)$ holds.* ■

By defining the partially-complete monitoring predicate $\mathsf{cmon}(m, \varphi)$ of Definition 2 and the sound monitoring predicate of Definition 1, we are now in a position to formalise our touchstone notion for monitor-formula correspondence.

**Definition 3 (Monitor-Formula Correspondence).** *A monitor $m$ is said to* monitor *for a formula $\varphi$, denoted as $\mathsf{mon}(m, \varphi)$, if it can do it* soundly*, and in a* partially-complete *manner:*

$$\mathsf{mon}(m, \varphi) \stackrel{def}{=} \mathsf{smon}(m, \varphi) \text{ and } \mathsf{cmon}(m, \varphi)$$ ■

*Example 8.* Consider the monitor $\alpha.\mathbf{yes}$ and the basic formula $\langle\alpha\rangle\mathbf{tt}$. One can verify that $\alpha.\mathbf{yes}$ monitors for $\langle\alpha\rangle\mathbf{tt}$, *i.e.,* $\mathsf{mon}(\alpha.\mathbf{yes}, \langle\alpha\rangle\mathbf{tt})$. Intuitively, this is because *every* system that $\alpha.\mathbf{yes}$ accepts must generate the trace $\alpha$, which is precisely the evidence required to show that the system satisfies $\langle\alpha\rangle\mathbf{tt}$. From this information, we can deduce that both requirements of Definition 3, *i.e.,* $\mathsf{smon}(\alpha.\mathbf{yes}, \langle\alpha\rangle\mathbf{tt})$ and $\mathsf{cmon}(\alpha.\mathbf{yes}, \langle\alpha\rangle\mathbf{tt})$, hold.

Consider now the same monitor compared to the formula $\langle\alpha\rangle\langle\beta\rangle\mathbf{tt}$. According to Definition 3, $\alpha.\mathbf{yes}$ does *not* monitor for $\langle\alpha\rangle\langle\beta\rangle\mathbf{tt}$. We can show this via the counterexample system $\alpha.\mathbf{0}$: it is not hard to see that the assertion $\mathsf{acc}(\alpha.\mathbf{0}, \alpha.\mathbf{yes})$ holds, *but* the system being monitored for does not satisfy the formula, *i.e.,* $\alpha.\mathbf{0} \notin$

$[\![\langle\alpha\rangle\langle\beta\rangle\mathbf{tt}]\!]$. This makes the monitor $\alpha.\mathbf{yes}$ unsound, *i.e.,* $\neg\mathsf{smon}(\alpha.\mathbf{yes}, \langle\alpha\rangle\langle\beta\rangle\mathbf{tt})$ from Definition 1.

By Definition 3, the monitor $\alpha.\mathbf{no} + \beta.\mathbf{no}$ monitors for $[\alpha]\mathbf{ff} \wedge [\beta]\mathbf{ff}$ by rejecting *all* the systems that violate the property. This is the case because a system violates $[\alpha]\mathbf{ff} \wedge [\beta]\mathbf{ff}$ if and only if it exhibits either trace $\alpha$ *or* trace $\beta$. Such systems are precisely those that are rejected by the monitor $\alpha.\mathbf{no} + \beta.\mathbf{no}$. ■

Definition 3 describes the relationship between monitors and logical formulae from the monitor's perspective. Dually, *monitorability* is an attribute of a logical formula, describing its ability to be adequately analysed at runtime by a monitor. The concept of monitorability can also be lifted to sets of formulae (*i.e.,* a sublogic). In this sense, the definition of monitorability is dependent on the monitoring set-up assumed (*i.e.,* the semantics of Figure 5) and the conditions that constitute an adequate runtime analysis. In what follows, we discuss the monitorability of our logic $\mu$HML assuming Definition 3 as our base notion for adequate monitoring.

**Definition 4 (Monitorability).** *A formula $\varphi \in \mu$HML is* monitorable *iff there exists a monitor $m$ such that $\mathsf{mon}(m, \varphi)$. A set of formulae $\mathcal{L} \subseteq \mu$HML is monitorable iff* every *formula in the set, $\varphi \in \mathcal{L}$, is monitorable.* ■

Showing that a set of formulae is monitorable is, in general, non-trivial since proving Definition 4 entails two universal quantifications: one for all the formulae in the formula set, and another one for all the systems of the LTS over which the semantics of the formulae is defined. In both cases, the respective sets may be infinite. We immediately note that not all logical formulae in $\mu$HML are monitorable. The following example substantiates this claim.

*Example 9.* Through the witness monitor $\alpha.\mathbf{no} + \beta.\mathbf{no}$ discussed in Example 8, we can show that formula $[\alpha]\mathbf{ff} \wedge [\beta]\mathbf{ff}$ is monitorable with a violation-complete monitor. By contrast, the variant formula $[\alpha]\mathbf{ff} \vee [\beta]\mathbf{ff}$ (swapping the conjunction with a disjunction operator) is *not*. This can be shown by arguing towards a contradiction. Assume that there exists some monitor $m$ such that $\mathsf{mon}(m, [\alpha]\mathbf{ff} \vee [\beta]\mathbf{ff})$. From Definition 3 we know that $\mathsf{smon}(m, [\alpha]\mathbf{ff} \vee [\beta]\mathbf{ff})$ and $\mathsf{cmon}(m, [\alpha]\mathbf{ff} \vee [\beta]\mathbf{ff})$ hold, and by Definition 2, there are two subcases to consider for $\mathsf{cmon}(m, [\alpha]\mathbf{ff} \vee [\beta]\mathbf{ff})$:

- If $m$ is satisfaction-complete, $\mathsf{scmon}(m, [\alpha]\mathbf{ff} \vee [\beta]\mathbf{ff})$, then $\mathsf{acc}(\beta.\mathbf{0}, m)$ for the specific system $\beta.\mathbf{0}$ since $\beta.\mathbf{0} \in [\![[\alpha]\mathbf{ff}]\!]$. By the semantics of the logic in Figure 3 we have $\beta.\mathbf{0} \in [\![[\alpha]\mathbf{ff} \vee [\beta]\mathbf{ff}]\!]$. From the acceptance $\mathsf{acc}(\beta.\mathbf{0}, m)$, we know that $m$ must either be able to reach a satisfaction, either autonomously, $m \Longrightarrow \mathbf{yes}$, or after observing action $\beta$, $m \overset{\beta}{\Longrightarrow} \mathbf{yes}$. For both cases we can argue that $m$ also accepts the system $\alpha.\mathbf{0} + \beta.\mathbf{0}$, since there exists a trace that leads the monitored system $m \triangleleft \alpha.\mathbf{0} + \beta.\mathbf{0}$ to an acceptance verdict. This is unsound (Definition 1) since $\alpha.\mathbf{0} + \beta.\mathbf{0} \notin [\![[\alpha]\mathbf{ff} \vee [\beta]\mathbf{ff}]\!]$, contradicting $\mathsf{smon}(m, [\alpha]\mathbf{ff} \vee [\beta]\mathbf{ff})$.

– If $m$ is violation-complete, $\mathsf{vcmon}(m, [\alpha]\mathbf{ff} \vee [\beta]\mathbf{ff})$, then, for the specific system $\alpha.\mathbf{0} + \beta.\mathbf{0}$, we must have $\mathsf{rej}(\alpha.\mathbf{0} + \beta.\mathbf{0}, m)$ since, by the logic semantics, we know $\alpha.\mathbf{0} + \beta.\mathbf{0} \notin [\![[\alpha]\mathbf{ff} \vee [\beta]\mathbf{ff}]\!]$. Now, from the structure of $\alpha.\mathbf{0} + \beta.\mathbf{0}$, we can deduce that $m$ can reach verdict **no** along one of the traces $\epsilon, \alpha$ or $\beta$:
  - If it is the empty trace $\epsilon$, then $m$ must also reject the system $\mathbf{0}$. However $\mathbf{0} \in [\![[\alpha]\mathbf{ff} \vee [\beta]\mathbf{ff}]\!]$ since $\mathbf{0}$ cannot produce any action; this makes the monitor unsound, contradicting $\mathsf{smon}(m, [\alpha]\mathbf{ff} \vee [\beta]\mathbf{ff})$.
  - If the trace is $\alpha$, $m$ must also reject the system $\alpha.\mathbf{0}$ along the same trace $\alpha$. This also makes the monitor unsound: from $\alpha.\mathbf{0} \in [\![[\beta]\mathbf{ff}]\!]$ and the semantics of Figure 3 we deduce $\alpha.\mathbf{0} \in [\![[\alpha]\mathbf{ff} \vee [\beta]\mathbf{ff}]\!]$. The case for $\beta$ is analogous. ■

**Definition 5 (Monitorable Logic).** *Let* mHML = cHML ∪ sHML*, where*

$$\begin{array}{rlllll} \pi, \varpi \in \text{cHML} ::= & \mathbf{tt} & \mid \mathbf{ff} & \mid \pi \vee \varpi & \mid \langle\alpha\rangle\pi & \mid \mathbf{min}\, X.\pi & \mid X \\ \theta, \vartheta \in \text{sHML} ::= & \mathbf{tt} & \mid \mathbf{ff} & \mid \theta \wedge \vartheta & \mid [\alpha]\theta & \mid \mathbf{max}\, X.\theta & \mid X \end{array}$$ ■

In [25,26], the syntactic subset mHML of Definition 5 is shown to be monitorable. At an intuitive level, mHML consists of the co-safe and safe syntactic subsets of $\mu$HML, resp. labelled as cHML and sHML. The logical subset cHML describes properties whose satisfying systems can provide a witness trace that enables the monitor to conclusively determine that they are included in the property. Dually, sHML captures properties whose systems are *unable to provide* a single witness trace that permits the monitor to conclude that they violate the property. Note that, for both cHML and sHML, any extension of a witness trace used to reach a verdict is a witness trace itself.

*Example 10.* Recall formulae $\varphi_1$, $\varphi_2$, $\varphi_5$ and $\varphi_6$ from Example 2. We can establish that these are indeed monitorable in the sense of Definition 4 by performing a simple syntactic check against the grammar in Definition 5. This avoids complicated semantic reasoning that is usually harder to automate, such as that shown in Example 8 and Example 9. ■

The work in [25,26] goes even further, and shows that the syntactic subset identified in Definition 5 is maximally expressive w.r.t. the monitorability of Definition 4. This means that all the properties that are monitorable can be expressed in terms of the syntactic fragment described in Definition 5. We are unaware of any other maximality results for monitorability in the field of RV.

*Example 11.* The formula $\langle\alpha\rangle\mathbf{tt} \wedge \langle\alpha\rangle\mathbf{tt}$ is not part of the monitorable syntactic fragment of Definition 5, as the cHML syntax prohibits possibility modalities (*i.e.,* $\langle\alpha\rangle$) from being combined using conjunctions. However, it turns out that the property denoted by the formula $\langle\alpha\rangle\mathbf{tt}\wedge\langle\alpha\rangle\mathbf{tt}$ is indeed monitorable because it can be monitored for by the monitor $\alpha.\mathbf{yes}$ (see Definition 3). In fact, $\langle\alpha\rangle\mathbf{tt}\wedge\langle\alpha\rangle\mathbf{tt}$ is semantically equivalent to the formula $\langle\alpha\rangle\mathbf{tt}$ which is, in turn, included in the syntactic fragment of Definition 5. More generally, the apparently restrictive mHML fragment of Definition 5 still allows us to describe *all* the monitorable properties expressible in $\mu$HML. ■

## 4 The Rocky Error Picture Show

A tenet of the basic RV set-up depicted in Figure 1 is that the monitor used in the configuration is itself correct. Yet, it is perilous to assume that monitors are immune to errors, for erroneous monitors pose a number of risks. To begin with, they could invalidate the runtime analysis performed, resulting in wasted computational overhead (irrespective of how low this might be). Even worse, erroneous monitors may jeopardise the execution of the SUS itself, and a system that originally satisfies a correctness specification ends up violating the same specification after it is instrumented with a monitor. Even though these risks could prove to be as detrimental as that of having high monitoring overheads, few monitors come equipped with correctness assurances. In many cases, is it even unclear what these assurances should be, giving rise to discrepancies between the expected monitor behaviour and the actual monitor implementation.

A formal development such as the monitorability formulation in Section 3 may help towards mitigating this. For instance, a synthesis function for the sublogic in Definition 5 was given as a by-product of the monitorability proofs in [25,26]; this synthesis function was shown to generate monitors that satisfy the correspondence requirements of Definition 3. However, these assurances (which mainly focus on the expressiveness of monitors) may not suffice for certain applications. To illustrate, the monitor $\alpha.\mathsf{yes} + \alpha.\mathsf{end}$ adequately monitors for the formula $\langle\alpha\rangle\mathsf{tt}$ according to Definition 3. Yet, it is not hard to see that this does not always yield an acceptance verdict when the SUS produces the witness trace $\alpha$. In this respect, the monitor $\alpha.\mathsf{yes}$ mentioned earlier in Example 8 may be seen as a better, or even, a more correct implementation than $\alpha.\mathsf{yes} + \alpha.\mathsf{end}$ that monitors for $\langle\alpha\rangle\mathsf{tt}$.

The work in [23] studies a possible basis for comparing monitors, that is independent of the specification language used. It develops a number of preorders of the form $m_1 \sqsubseteq m_2$. Intuitively, these denote the fact that, when instrumented with any *arbitrary* system $p$, if the monitored system $m_1 \triangleleft p$ exhibits certain characteristics, then $m_2 \triangleleft p$ exhibits them as well. For different monitoring characteristics, one obtains different preorders. Such preorders may be used in a variety of ways. They may be employed as notions of refinement, where $m_1$ represents a monitor specification whose behaviour is preserved by the concrete implementation $m_2$. Preorders may also be used to determine when one monitor implementation can be safely substituted with another, without affecting the existing characteristics of a monitoring set-up. Importantly, however, they provide a foundation for understanding monitor errors whenever the relation $m_1 \sqsubseteq m_2$ does not hold.

We review the salient aspects of this work in the context of our foundational theory for monitors. To simplify the material that follows, we restrict ourselves to monitors that only reach rejections. This allows us to side-step issues related to soundness (discussed earlier in Section 3) and focus on orthogonal behavioural aspects. Note that our preference for rejections over acceptances is arbitrary. We begin by capturing the (complete) execution of a monitored system, which may

be described in our setting via the notion of a *computation*, defined below. In Definition 6 the trailing $\tau$-transitions permit the monitor to stabilise and reach a verdict after a number of internal computational steps.

**Definition 6.** *The transition sequence with trace $t$, $m \triangleleft p \overset{t}{\Longrightarrow} m_0 \triangleleft p_0 \overset{\tau}{\longrightarrow} m_1 \triangleleft p_1 \overset{\tau}{\longrightarrow} m_2 \triangleleft p_2 \overset{\tau}{\longrightarrow} \ldots$, is called a $t$-*computation *if it is* maximal *(i.e., either it is* infinite *or it is finite and* cannot be extended further *using $\tau$-transitions). A $t$-computation is called* rejected *(or a rejected computation along $t$) iff there exists some $n \in \mathbb{N}$ in the transition sequence where $m_n = \mathbf{no}$.* ■

Following Definition 3, a criterion for comparing monitors considers the *possible verdicts* that may be reached after observing a *specific execution trace* produced by the SUS. In this sense, a monitor is *as good as* another monitor if it can match all of the rejected computations of the other monitor.

**Definition 7.** *A monitor $m$ potentially-rejects system $p$ along trace $t$, denoted as $\mathsf{pr}(m, p, t)$, iff there* exists *a rejecting $t$-computation from $m \triangleleft p$. Monitor $m_2$ is as good as $m_1$ w.r.t. potential rejections, denoted as $m_1 \sqsubseteq_{\mathsf{pr}} m_2$, iff*

$$\textit{for all systems } p\textit{, and all traces } t\textit{, } \mathsf{pr}(m_1, p, t) \textit{ implies } \mathsf{pr}(m_2, p, t).$$

*The preorder induces the expected kernel equivalence $m_1 \cong_{\mathsf{pr}} m_2 \overset{def}{=} (m_1 \sqsubseteq_{\mathsf{pr}} m_2$ and $m_2 \sqsubseteq_{\mathsf{pr}} m_1)$. We write $m_1 \sqsubset_{\mathsf{pr}} m_2$ in lieu of $(m_1 \sqsubseteq_{\mathsf{pr}} m_2$ and $m_2 \not\sqsubseteq_{\mathsf{pr}} m_1)$.* ■

*Example 12.* Consider the following monitor descriptions:

$$m_1 = \alpha.\beta.\mathbf{no} \qquad m_2 = \alpha.\mathbf{no} \qquad m_3 = \alpha.\mathbf{no} + \beta.\mathbf{no} \qquad m_4 = \alpha.\mathbf{no} + \beta.\mathbf{no} + \beta.\mathbf{end}$$

We have $m_1 \sqsubset_{\mathsf{pr}} m_2$ since, for any $p$, any rejected $t$-computation of $m_1$ (which must have prefix $\alpha\beta$) is also rejected by $m_2$, but the inverse is not: for some $p \xrightarrow{\alpha} p'$ we have $\mathsf{pr}(m_2, p, \alpha)$ but *not* $\mathsf{pr}(m_1, p, \alpha)$. For similar reasons, we have

$$m_2 \;\sqsubset_{\mathsf{pr}}\; m_3 \;\cong_{\mathsf{pr}}\; m_4$$

Observe that $m_3$ and $m_4$ are considered to be potential-rejection equivalent. ■

Potential rejections may be too weak for certain mission critical applications. For instance, although $m_4$ can reject traces that start with $\beta$, it does not mean that it will, because it may (non-deterministically) follow the branch $\beta.\mathbf{end}$ that does not lead to a rejection. An alternative criterion would thus be to compare monitors w.r.t. their *deterministic* rejections.

**Definition 8.** *A monitor $m$ deterministically-rejects system $p$ along trace $t$, denoted as $\mathsf{dr}(m, p, t)$, iff* all *$t$-computations from $m \triangleleft p$ are rejecting. Monitor $m_2$ is as good as $m_1$ w.r.t. deterministic rejections, denoted as $m_1 \sqsubseteq_{\mathsf{dr}} m_2$, iff*

$$\textit{for all systems } p\textit{, and all traces } t\textit{, } \mathsf{dr}(m_1, p, t) \textit{ implies } \mathsf{dr}(m_2, p, t).$$

*The respective kernel equivalence, $m_1 \cong_{\mathsf{dr}} m_2$, and irreflexive preorder, $m_1 \sqsubset_{\mathsf{dr}} m_2$, induced by deterministic rejections are as expected.* ■

*Example 13.* We have the following following relationships w.r.t. deterministic rejections for the monitors $m_1$ to $m_4$ from Example 12:

$$m_1 \sqsubset_{\mathsf{dr}} m_2 \cong_{\mathsf{dr}} m_4 \sqsubset_{\mathsf{dr}} m_3$$

We note that, whereas in Example 12, $m_4$ was considered to be strictly better than $m_2$ w.r.t. potential rejections, it is considered equivalent w.r.t. deterministic rejections, since the rejections of $m_4$ for traces commencing with a $\beta$ action are not deterministic and thus ignored. ■

It is worth mentioning that defining the preorders of Definitions 7 and 8 in terms of instrumented system executions (instead of just considering the respective monitor traces in isolation) reveals subtleties that would otherwise be missed. These would nevertheless manifest themselves at runtime.

*Example 14.* Consider the construct $\tau.m$, describing a monitor that performs an internal computation $\tau$ before behaving like $m$. Using the syntax of Figure 5, this may be encoded as $\mathbf{rec}\, x.m$, where $x$ is not a free variable in the continuation $m$ (see rule Rec). We have the following relations between a monitor that immediately rejects (**no**), and another that performs some internal computation before yielding reject ($\tau.\mathbf{no}$):

$$\mathbf{no} \cong_{\mathsf{pr}} \tau.\mathbf{no} \qquad \text{but} \qquad \tau.\mathbf{no} \sqsubset_{\mathsf{dr}} \mathbf{no}$$

It is not hard to see why the monitor **no** is a top element for both preorders $\sqsubseteq_{\mathsf{pr}}$ and $\sqsubseteq_{\mathsf{dr}}$, since it immediately rejects all traces for any given system $p$ (*i.e.,* for any $m$, we have $m \sqsubseteq_{\mathsf{pr}} \mathbf{no}$ and $m \sqsubseteq_{\mathsf{dr}} \mathbf{no}$). The monitor $\tau.\mathbf{no}$ exhibits the exact behaviour w.r.t. potential rejections and is thus a top element for $\sqsubseteq_{\mathsf{pr}}$ as well. Interestingly, $\tau.\mathbf{no}$ is *not* a top element for $\sqsubseteq_{\mathsf{dr}}$ however. Consider, as a counter example, the system $p$ that goes into an infinite (internal) loop $p \xrightarrow{\tau} p \xrightarrow{\tau} \ldots$. When instrumented with the monitor $\tau.\mathbf{no}$, the monitored system can exhibit the $\epsilon$-computation $\tau.\mathbf{no} \triangleleft p \xrightarrow{\tau} \tau.\mathbf{no} \triangleleft p \xrightarrow{\tau} \tau.\mathbf{no} \triangleleft p \xrightarrow{\tau} \ldots$ effectively starving the monitor $\tau.\mathbf{no}$, and preventing it from ever reaching its rejection verdict, **no**. Thus, we do not have $\mathsf{dr}(\tau.\mathbf{no}, p, \epsilon)$ and, as a result, the inequality $\mathbf{no} \sqsubseteq_{\mathsf{dr}} \tau.\mathbf{no}$ does *not* hold. ■

Formulating Definitions 7 and 8 in terms of instrumented system executions also facilitates the classification of anomalous monitor behaviour.

*Example 15.* Consider the unresponsive monitor $m_\omega$, that goes into an infinite loop without exhibiting any other behaviour (*i.e.,* $m_\omega \xrightarrow{\tau} m_\omega \xrightarrow{\tau} \ldots$). Whereas for the potentially-rejecting preorder, this monitor is clearly a bottom element (*i.e.,* for any $m$ we have $m_\omega \sqsubseteq_{\mathsf{pr}} m$) it is, perhaps surprisingly, *not* a bottom element for the deterministic-rejection preorder. In fact, according to Definition 8, we obtain seemingly peculiar orderings such as the one below (using monitor $m_2$ defined earlier in Example 12):

$$m_2 \sqsubset_{\mathsf{dr}} m_\omega \sqsubset_{\mathsf{dr}} \mathbf{no}$$

Note that, according to the instrumentation semantics of Figure 5, $m_\omega$ prevents any system that is composed with it from generating observable events: for arbitrary $p$, whenever $p \xrightarrow{\alpha} p'$ given some $\alpha$, rule Mon cannot be applied (since $m_\omega \not\xrightarrow{\alpha}$) and neither can rule Ter (since $m_\omega \xrightarrow{\tau}$). This means that that $\mathsf{dr}(m_\omega, p, t)$ holds for any system $p$ and trace $t$ that is not empty (*i.e.,* $t \neq \epsilon$). This condition holds trivially because no such trace exists, thus explaining why $m_2 \sqsubset_{\mathsf{dr}} m_\omega$. The only case where $\mathsf{dr}(m_\omega, p, t)$ does *not* hold is precisely when $t = \epsilon$, leaving us with $m_\omega \sqsubset_{\mathsf{dr}}$ **no**. ■

Using the two preorders $\sqsubseteq_{\mathsf{pr}}$ and $\sqsubseteq_{\mathsf{dr}}$, we can define a third (more refined) preorder that is an intersection of the two presented thus far, taking into consideration both of their respective criteria when comparing monitors.

**Definition 9.** *A monitor $m_2$ is as good as a monitor $m_1$ w.r.t. rejections, denoted as $m_1 \sqsubseteq_{rej} m_2$, iff $m_1 \sqsubseteq_{pr} m_2$ and $m_1 \sqsubseteq_{dr} m_2$.* ■

Whenever we have $m_1 \sqsubseteq_{\mathsf{rej}} m_2$, we can substitute $m_1$ with $m_2$ safe in the knowledge that, all potential and deterministic rejections made by $m_1$ are preserved by $m_2$ irrespective of the system these are composed with. Alternatively, should we consider *missed* rejections as our notion of error, then whenever $m_1 \not\sqsubseteq_{\mathsf{rej}} m_2$ we know that when instrumented with a common system, $m_2$ may *not* exhibit all the rejections that $m_1$ produces.

*Example 16.* We obtain the following strict ordering w.r.t. Definition 9:

$$m_1 \sqsubset_{\mathsf{rej}} m_2 \sqsubset_{\mathsf{rej}} m_4 \sqsubset_{\mathsf{rej}} m_3$$ ■

In [23], additional monitor preorders are considered based on other correctness criteria. For instance, monitors are compared w.r.t. to their capability of interfering with the execution of a system, as outlined briefly in Example 15. Apart from devising correctness preorders, the aforementioned work also develops sound compositional techniques that can check for preorder inequalities *without* relying on universal quantifications on systems. Although this is essential for any automated checking of said preorder inequalities, such compositional techniques are outside the scope of this presentation. Interested readers should nevertheless consult [23] for more details.

## 5 Monitorability and Correctness Now Redux

The concepts outlined in Sections 3 and 4 served as a basis for various other work related to RV, monitors and monitorability. We here discuss a subset of this work.

**Tools.** In [6], the monitorability results presented in Section 3 (and their respective proofs reported in [26]) were used for the construction of a monitoring tool called detectEr. In this tool, monitorable correctness properties for reactive Erlang programs can be specified using the monitorable syntactic subset

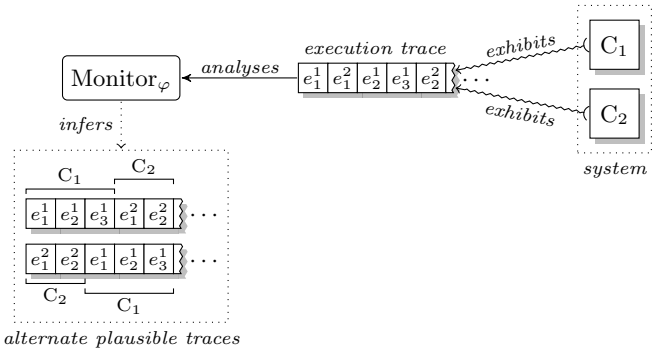


**Fig. 7.** Monitoring components $C_1$ and $C_2$ and inferring their other plausible traces.

of Definition 5. From these logical specifications, detectEr automatically synthesises monitors that are then instrumented to execute alongside an Erlang SUS, reporting satisfaction or violation verdicts depending on the behaviour exhibited. The need for a more comprehensive instrumentation mechanism for the target language (Erlang), resulted in the development of an aspect-oriented utility called eAOP [15]. This tool is incorporated into the detectEr toolchain, and its development and features (*e.g.* its pointcut definitions) are largely guided by the RV requirements for detectEr. Nevertheless, eAOP can also be used as a standalone utility for other purposes such as Erlang code refactoring.

**Monitorability.** The work of [7] considers a slightly different monitoring setup to that presented in Figure 1. Interestingly, it shows how the maximality results for monitorability of Section 3 can be extended when working with the new setup. This work views the SUS as a collection of components (instead of treating it as one monolithic block) whereby certian trace events can be attributed to specific components, as shown in Figure 7. The monitor is equipped with this static information about the SUS to permit it to extend its analytical capabilities. In the case of [7], certain components are known to be execution-independent to one another, meaning that from a particular trace interleaving observed in the current execution trace, the monitor could infer additional (plausible) traces that can also be used for verification purposes.

For instance, in Example 9 we earlier argued why the property $[\alpha]\mathbf{ff} \vee [\beta]\mathbf{ff}$ in the basic setup of Figure 1 is *not* monitorable. The formula states that a SUS satisfies the property if it either cannot perform $\alpha$ or it cannot perform $\beta$. A trace can however only provide us with enough evidence to deduce that only one of the execution subbranches is violated, not both. Yet, if we know that actions $\alpha$ and $\beta$ can be produced exclusively by independently-executing components $C_1$ and $C_2$ resp., from a witness trace of the form $\alpha\beta\ldots$, a monitor in the setup of Figure 7 could infer that the trace $\beta\alpha\ldots$ can also be generated by the system (for a different execution interleaving of components $C_1$ and $C_2$). This allows the

monitor to analyse *two* execution traces instead of one: the witness and inferred traces can then be used together to obtain enough evidence to be able to deduce a violation of the property $[\alpha]\mathbf{ff} \vee [\beta]\mathbf{ff}$, making it monitorable.

In other work [22], the authors show how the monitorable subset of Definition 5 can be used to devise a strategy that apportions the verification process between the pre-deployment (via standard techniques such as Model Checking) and post-deployment (using RV) phases of software development. To illustrate, consider the property $\langle\alpha\rangle([\beta]\mathbf{ff} \vee \langle\alpha\rangle\mathbf{tt})$ which clearly does not belong to the monitorable subset of Definition 5. Reformulating it into its semantic equivalent, $\langle\alpha\rangle[\beta]\mathbf{ff} \vee \langle\alpha\rangle\langle\alpha\rangle\mathbf{tt}$, allows us to isolate the runtime-monitorable subformula $\langle\alpha\rangle\langle\alpha\rangle\mathbf{tt}$. One could then check for subformula $\langle\alpha\rangle[\beta]\mathbf{ff}$ prior to deployment, and in cases where this is not satisfied, monitor for the $\langle\alpha\rangle\langle\alpha\rangle\mathbf{tt}$ at runtime.

**Monitor Correctness.** In [1], the authors consider the use of standard determinisation techniques borrowed from automata theory to determinise monitors and their respective verdicts. The approach would enable a tool to take (non-deterministic) monitor descriptions such as $\alpha.\beta.\mathbf{yes} + \alpha.\gamma.\mathbf{yes}$ (*e.g.* produced by the detectEr tool of [6] when synthesising monitors from the monitorable formula $\langle\alpha\rangle\langle\beta\rangle\mathbf{tt} \vee \langle\alpha\rangle\langle\gamma\rangle\mathbf{tt}$) and obtain the monitor $\alpha.(\beta.\mathbf{yes} + \gamma.\mathbf{yes})$ instead. The key contribution of this work is the establishment of complexity upper bounds (more or less known) and, more interestingly, complexity lower bounds whereby converting a non-deterministic monitor of $n$ states to a deterministic one requires a space complexity of *at least* $2^{2^{\Omega\sqrt{n \log n}}}$.

A fully deterministic monitor behaviour (where every event analysed leads exclusively to one possible monitor state) is too rigid in practice to describe the behaviour of certain monitor implementations. A case in point is monitoring that is conducted via a number of concurrent submonitors [27] that may be subject to different thread interleaving every time they are executed. The monitor framework discussed in Section 4 served as a basis for the formulation of a definition for consistently-detecting monitors [24] that allows degrees of non-deterministic behaviour as long as these are *not externally observable* via inconsistent detections. The work in [24] borrows from the compositionality techniques studied in [23] to define an alternative coinductive definition for consistent monitor behaviour based on the notion of *controllability* [31]. It also develops symbolic techniques that facilitate the construction of analysis tools for determining monitor controllability.

**Language Design.** The work presented in [14] uses the safety fragment sHML from Definition 5 as a basis for defining a scripting language for describing adaptation procedures in a monitor-oriented programming paradigm. It specifically targets systems that are constructed in a layered fashion, with the outer layers adding degrees of functionality in response to the behaviour observed from the inner layers. In this setup, the monitors produced are not passive in the sense that they merely analyse behaviour, but *actively* engage with the observed system via adaptations so as to change its behaviour and mitigate system faults.

Despite of their benefits, runtime adaptations induced by monitors may introduce certain errors themselves. For this reason, in [13] the authors also define language-based methods in the form of a type system to assist monitor construction and to statically detect erroneous adaptations declared in the monitor scripts at design time.

## 6 Conclusion

In this paper we surveyed a body of work that strives to establish a formal foundation for the RV technique. It focusses on addressing two main questions, namely monitorability for system specifications and monitor correctness. Although the account concentrated mainly on unifying the work presented in [25,26] and [23], it also showed how this preliminary work has served as a basis for subsequent developments on the subject, such as [13,14,1,7,24]. We are currently working on extending these results even further, by investigating monitorability for diverse monitoring set-ups along the lines of [7], and by considering monitors with augmented capabilities such as those that perform property enforcement [36].

## References


1. L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, and S. Ö. Kjartansson. On the Complexity of Determinizing Monitors. In *CIAA*, volume 10329 of *LNCS*, pages 1–13, 2017.
2. L. Aceto, A. Ingólfsdóttir, K. G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.
3. W. Ahrendt, J. M. Chimento, G. J. Pace, and G. Schneider. A Specification Language for Static and Runtime Verification of Data and Control Properties. In *FM*, volume 9109 of *LNCS*, pages 108–125, 2015.
4. I. Aktug and K. Naliuka. ConSpec - A formal language for policy specification. *Sci. Comput. Program.*, 74(1-2):2–12, 2008.
5. C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. R. Lowry, C. S. Pasareanu, G. Rosu, K. Sen, W. Visser, and R. Washington. Combining Test Case Generation and Runtime Verification. *Theor. Comput. Sci.*, 336(2-3):209–234, 2005.
6. D. P. Attard and A. Francalanza. A Monitoring Tool for a Branching-Time Logic. In *RV*, volume 10012 of *LNCS*, pages 473–481, 2016.
7. D. P. Attard and A. Francalanza. Trace Partitioning and Local Monitoring for Asynchronous Components. In *SEFM*, LNCS, 2017. (to appear).
8. S. Azzopardi, C. Colombo, G. J. Pace, and B. Vella. Compliance Checking in the Open Payments Ecosystem. In *SEFM*, volume 9763 of *LNCS*, pages 337–343, 2016.
9. C. Baier and J. P. Katoen. *Principles of Model Checking*. MIT Press, New York, 2008.
10. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57. 2004.
11. D. A. Basin, F. Klaedtke, S. Marinovic, and E. Zalinescu. Monitoring Compliance Policies over Incomplete and Disagreeing Logs. In *RV*, volume 7687 of *LNCS*, pages 151–167, 2012.

12. G. P. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. R. Lowry, C. S. Pasareanu, A. Venet, W. Visser, and R. Washington. Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. *Formal Methods in System Design*, 25(2-3):167–198, 2004.
13. I. Cassar and A. Francalanza. Runtime Adaptation for Actor Systems. In *RV*, volume 9333 of *LNCS*, pages 38–54, 2015.
14. I. Cassar and A. Francalanza. On Implementing a Monitor-Oriented Programming Framework for Actor Systems. In *IFM*, volume 9681 of *LNCS*, pages 176–192, 2016.
15. I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfsdóttir. eAOP - An Aspect Oriented Programming Framework for Erlang. In *Erlang Workshop*, 2017. (to appear).
16. F. Chen and G. Rosu. MOP: An Efficient and Generic Runtime Verification Framework. In *OOPSLA*, pages 569–588, 2007.
17. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
18. C. Colombo, A. Francalanza, R. Mizzi, and G. J. Pace. polyLarva: Runtime Verification with Configurable Resource-Aware Monitoring Boundaries. In *SEFM*, volume 7504 of *LNCS*, pages 218–232, 2012.
19. 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 *TIME*, pages 166–174, 2005.
20. S. Debois, T. T. Hildebrandt, and T. Slaats. Safety, Liveness and Run-Time Refinement for Modular Process-Aware Information Systems with Dynamic Sub Processes. In *FM*, volume 9109 of *LNCS*, pages 143–160, 2015.
21. N. Decker, M. Leucker, and D. Thoma. jUnitRV-Adding Runtime Verification to jUnit. In *NFM*, volume 7871 of *LNCS*, pages 459–464, 2013.
22. D. Della Monica and A. Francalanza. Towards a Hybrid Approach to Software Verification. In *NWPT*, number SCS16001 in RUTR, pages 51–54, 2015.
23. A. Francalanza. A Theory of Monitors - (Extended Abstract). In *FoSSaCS*, volume 9634 of *LNCS*, pages 145–161, 2016.
24. A. Francalanza. Consistently-Detecting Monitors. In *CONCUR*, LNCS, 2017. (to appear).
25. A. Francalanza, L. Aceto, and A. Ingólfsdóttir. On Verifying Hennessy-Milner Logic with Recursion at Runtime. In *RV*, volume 9333 of *LNCS*, pages 71–86, 2015.
26. A. Francalanza, L. Aceto, and A. Ingolfsdottir. Monitorability for the Hennessy–Milner Logic with Recursion. *Formal Methods in System Design*, pages 1–30, 2017.
27. A. Francalanza and A. Seychell. Synthesising Correct Concurrent Runtime Monitors. *Formal Methods in System Design*, 46(3):226–261, 2015.
28. A. Kane, O. Chowdhury, A. Datta, and P. Koopman. A Case Study on Runtime Monitoring of an Autonomous Research Vehicle (ARV) System. In *RV*, volume 9333 of *LNCS*, pages 102–117, 2015.
29. A. Kassem, Y. Falcone, and P. Lafourcade. Monitoring Electronic Exams. In *RV*, volume 9333 of *LNCS*, pages 118–135, 2015.
30. M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A Run-Time Assurance Approach for Java Programs. *Formal Methods in System Design*, 24(2):129–155, 2004.
31. J. Klamka. *Control System, Robotics and Automation*, volume 7, chapter System Characteristics: Stability, Controllability, Observability. EOLLS, 2009.
32. D. Kozen. Results on the Propositional mu-Calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

33. K. G. Larsen. Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion. *Theor. Comput. Sci.*, 72(2&3):265–288, 1990.
34. F. Lerda and W. Visser. Addressing Dynamic Issues of Program Model Checking. In *SPIN*, volume 2057 of *LNCS*, pages 80–102, 2001.
35. M. Leucker and C. Schallhart. A Brief Account of Runtime Verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.
36. J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Secur.*, 4(1-2):2–16, 2005.
37. P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Rosu. An Overview of the MOP Runtime Verification Framework. *STTT*, 14(3):249–289, 2012.
38. R. Neykova and N. Yoshida. Let it Recover: Multiparty Protocol-Induced Recovery. In *CC*, pages 98–108, 2017.
39. G. Reger, H. C. Cruz, and D. E. Rydeheard. MarQ: Monitoring at Runtime with QEA. In *TACAS*, volume 9035 of *LNCS*, pages 596–610, 2015.
40. S. Varvaressos, D. Vaillancourt, S. Gaboury, A. B. Massé, and S. Hallé. Runtime Monitoring of Temporal Logic Properties in a Platform Game. In *RV*, volume 8174 of *LNCS*, pages 346–351, 2013.