# The Concurrency Column

by

Marino Miculan and Nobuko Yoshida

Department of Mathematics, Computer Science and Physics
University of Udine, Italy
marino.miculan@uniud.it

Department of Computer Science
University of Oxford, UK
nobuko.yoshida@cs.ox.ac.uk

The survey "Foundations of Runtime Monitoring through the Lens of Concurrency Theory" provides an overview of the theoretical advancements in runtime monitoring over the last decade. This work is of central importance to the theoretical computer science community as it addresses the fundamental challenge of monitorability, specifically characterizing which properties can be conclusively verified using finite execution traces while navigating the inherent loss of expressiveness compared to exhaustive methods like model checking.

A key aspect of the paper is the rigorous application of concurrency concepts to the monitoring domain. The authors utilize variations of Hennessy-Milner Logic with recursion (recHML) as a touchstone for specifications and employ process-algebraic languages, such as Milner's Calculus of Communicating Systems, to describe monitors. This approach allows for the use of structural operational semantics to formally define monitor behaviour and system interaction, enabling the creation of syntax-directed, correct-by-construction procedures for monitor synthesis. Therefore, the paper demonstrates how established concurrency concepts can address contemporary challenges in verifying complex distributed systems.

# Foundations of Runtime Monitoring through the Lens of Concurrency Theory

Luca Aceto
ICE-TCS, Department of Computer Science,
Reykjavik University, Iceland
Gran Sasso Science Institute, L'Aquila, Italy
luca@ru.is, luca.aceto@gssi.it

Antonis Achilleos
ICE-TCS, Department of Computer Science,
Reykjavik University, Iceland
antonios@ru.is

Elli Anastasiadi
Department of Computer Science, Aalborg University, Denmark
ellia@cs.aau.dk

Duncan Paul Attard
University of Malta, Msida, Malta
duncan.attard@um.edu.mt

Léo Exibard
LIGM, CNRS, Univ. Gustave Eiffel, Marne-la-Vallée, France
leo.exibard@univ-eiffel.fr

Adrian Francalanza
University of Malta, Msida, Malta
adrian.francalanza@um.edu.mt

Daniele Gorla
Department of Computer Science,
'Sapienza' University of Rome, Italy
gorla@di.uniroma1.it

Anna Ingólfsdóttir
ICE-TCS, Department of Computer Science,
Reykjavik University, Iceland
annai@ru.is

Karoliina Lehtinen
CNRS, Aix-Marseille University, LIS, Marseille, France
lehtinen@lis-lab.fr

Jana Wagemaker
Software Science Group, Radboud University, the Netherlands
jana.wagemaker@ru.nl

**Abstract**

This article surveys some of the work on the theoretical foundations of runtime monitoring carried out by various subsets of its authors over the last decade. It focuses on runtime monitoring of classic regular properties, of data-dependent properties and of hyperproperties. The paper also highlights the research philosophy guiding those studies, and the role that classic notions and techniques from concurrency theory have played in them.

# 1 Introduction

Runtime monitoring (also known as runtime verification) is a lightweight verification technique that can be used to determine whether a system satisfies some given requirements as it executes in its actual operating environment. The origins of modern runtime monitoring can be traced back to an influential workshop organised by Klaus Havelund and Grigore Rosu in 2001 [55], which then became the International Conference on Runtime Verification[1] that celebrated its 25th anniversary in 2025. To the best of our knowledge, Havelund and Rosu also coined the term 'runtime verification'.

[1] See https://runtime-verification.github.io/.

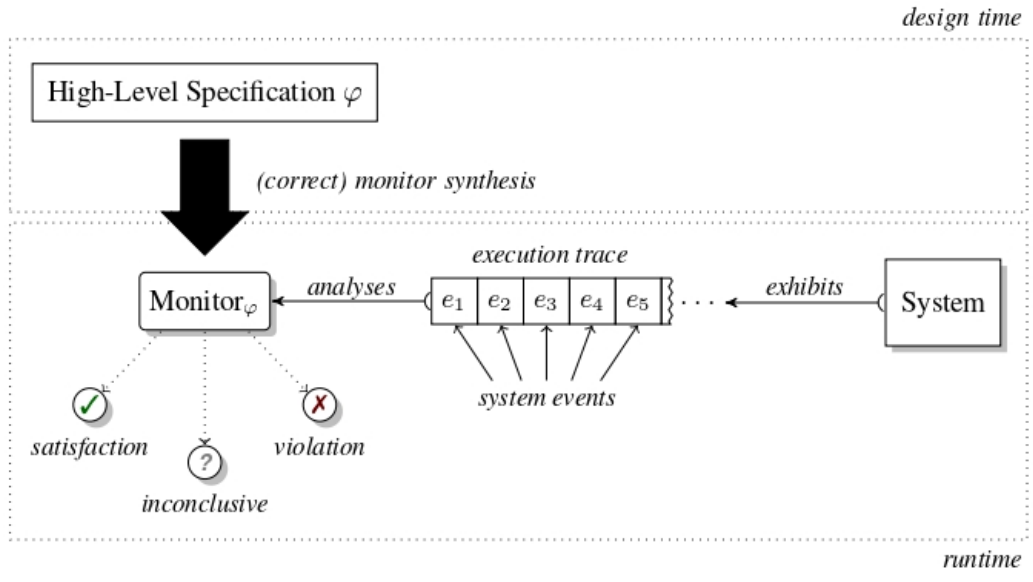


Figure 1: Runtime monitoring.

The general setting for (online) runtime monitoring is shown in Figure 1. Assume that we wish to determine whether a system operates according to some specification $\varphi$ expressed in a formal specification language. In runtime monitoring, a computational entity called *monitor* is used to check whether the system under scrutiny satisfies $\varphi$ or not. The task of a monitor for $\varphi$ is to analyse the observable events in the current execution trace and to use the information it gleans from those observations to determine whether the system satisfies or violates the specification $\varphi$. Since the monitoring set-up should be part of the trusted computing base, it is desirable to synthesise monitors from specifications automatically and in a provably correct way. Indeed, as discussed in [1, 77] amongst other references, writing and maintaining monitors manually is costly and error prone.

Runtime monitoring is an increasingly popular verification technique that can be used in conjunction with classic approaches for the analysis and validation of software-based systems such as model checking [26, 42], deductive verification [22, 29, 64] and testing [69]. Indeed, several characteristics of runtime monitoring make it particularly suitable in the analysis of modern computing systems. For starters, runtime monitoring is a best-effort verification approach that focuses on analysing a *single system execution* (or possibly a few of them, as in offline monitoring or in the multiple-execution, online-monitoring setting studied in [20]). Thus, runtime monitoring does not suffer from the infamous state-explosion problem and is typically more scalable than variations on model checking, which are based on an exhaustive exploration of the state space of the system. (For a counterpoint,

see the complexity results presented in [66, 67].) Moreover, runtime monitoring typically treats the system as a black box and can be applied also when a model of the system under scrutiny or its source code are not available, for instance, because the system is proprietary. The black-box nature of runtime monitoring makes it an excellent fit for the analysis of systems that employ machine learning components and other subsystems based on artificial intelligence artefacts [36, 57, 76, 80]. Moreover, runtime monitoring can exploit the availability of multi-core systems, in that, while some of the processing units in a system carry out tasks pertaining to achieving system objectives, others can check whether a system execution satisfies the correctness requirements identified at design time. Finally, runtime monitoring takes place post-deployment while the system is running in its actual operating environment. This makes this approach very useful in settings, such as autonomous robotics and security, where it is practically impossible to imagine all the possible environments in which a system will operate [45, 75]. We are creative, but Nature is often adversarial and more creative than we are!

However, the aforementioned advantages of runtime monitoring as a verification technique come at the price of a loss of expressiveness. Indeed, as indicated in Figure 1, a monitor issues a verdict on whether the system under scrutiny satisfies the requirement expressed by $\varphi$ after having observed a finite fragment of the system execution. There are properties for which this information is not sufficient to reach a conclusive verdict. As an example, consider the property 'the $a$ action is performed infinitely often'. If a system can perform at least two different actions, then a monitor will never be able to determine conclusively whether that property is satisfied or violated, no matter what finite trace of actions it has observed thus far. Therefore, as Figure 1 makes clear, in general, the logic of monitors is (at least) three valued in that the verdict of a monitor for some property might remain forever inconclusive.

In light of the above discussion, a natural question to ask is whether one can characterise the collection of properties expressible in a given specification language that can be monitored in some given monitoring set-up, by what type of monitors and with what correctness guarantees. That question has been at the heart of the work on the theoretical foundations of monitorability carried out by various subsets of the authors of this article since our team's initial study of that topic in [50, 51]. Perhaps unsurprisingly, in light of our research background, we have attempted to answer it through the lens of concurrency theory. More specifically, our work on runtime monitoring has so far been based on, and guided by, the following design decisions.

- We have used variations on Hennessy-Milner logic with recursion [63] as our touchstone language for writing specifications of system behaviour. That logic is closely related to the modal $\mu$-calculus [61] and can express the

operators in classic temporal logics [46]. Moreover, we use the standard branching- and linear-time semantics for the logics we study and do not change it to fit the runtime-monitoring setting. To our mind, doing so allows one to develop a theory of runtime monitoring for those logics that can be seamlessly combined, as needed, with other approaches to system verification such as model checking.

- We have employed variations on Milner's CCS [68] as our formalism for describing monitors. All our monitor-description languages come equipped with a structural operational semantics [72] that gives the semantics of monitors in terms of a labelled transition system [60]. We also use structural operational semantics to define the interaction between a system under scrutiny and a monitor observing its execution. Moreover, the operators that can be used to construct monitors are carefully chosen, in each case, to facilitate the syntax-directed definition of correct-by-construction, monitor-synthesis procedures from logical specifications. This allows us to reason about monitors, their behaviour and their correctness guarantees using classic proof techniques such as rule induction and structural induction.

- Our study of the specifications in our touchstone logics that can be monitored is relative to a given monitoring set-up. Indeed, the collection of monitorable properties depends on the power of the entities that carry out the monitoring task, viz. the monitors, and on the correctness guarantees one wants the monitoring process to uphold. Our tenet is that monitorability is best viewed as a spectrum of notions, which reflects the trade-off between what properties can be monitored and the correctness guarantees one would like to have—see the article [16], which relates our approach to classic notions of monitorability such as the one given by Pnueli and Zaks in [73] and those studied in [54].

- For each point in the spectrum of notions of monitorable properties, we have striven to identify fragments of our touchstone logic that are *expressively complete*, meaning that every monitorable property is logically equivalent to a formula in that fragment. Apart from their intrinsic theoretical interest, those characterisations may also have practical relevance. Indeed, the definition of automated, syntax-directed, monitor-synthesis procedures need only deal with formulae in the relevant fragment and can be optimised using its syntactic features. Moreover, designers specifying correctness properties can rest assured that if they specify system requirements using a formula in a given fragment, then (1) their requirements can be monitored at runtime with the correctness guarantees that accompany that fragment and (2) correct-by-construction monitors that do so can be generated automatically.

In the rest of this article, we will limit ourselves to presenting three exhibits describing the above-mentioned research approach at work. We will begin by discussing a classic setting in which system executions are expressed as (finite and) infinite traces (Section 2). We will then consider two variations on that setting dealing with monitoring of systems whose behaviour is data dependent (Section 3) and with centralised and decentralised monitoring of hyperproperties (Section 4). In each section, we will keep the presentation relatively informal, focusing on the main ideas and the key results that highlight the research philosophy underlying our work. We hope that readers will be enticed to check the scientific publications we cite for information on the technical details and on the software tools based on the theoretical results. The paper ends with some concluding remarks in Section 5.

**Disclaimer** This article is meant to be an appetiser offering a small taste of some of our work on runtime monitoring and highlighting its guiding principles, couched in classic concurrency theory. It is *not* a survey of the literature on runtime monitoring, which is vast and would deserve a handbook-length treatment. We refer our readers to [28, 65, 74, 78] and the references therein for overviews of the work done in the field of runtime monitoring.

# 2 Exhibit A: Adventures in classic monitorability

We start our journey by studying monitorability in a classic setting in which a system under scrutiny can perform observable actions from a non-empty, finite set Act, ranged over by $a$ and $b$. Following Milner [68], we use $\tau$ as a distinguished unobservable system action that does not occur in Act and let $\mu \in \text{Act} \cup \{\tau\}$.

In this setting, infinite sequences of observable actions $t, u \in \text{Trc} = \text{Act}^{\omega}$, which we usually call *traces*, are a natural model for (the observable content of) system executions. Occasionally, we need to refer to *finite traces*, denoted by $s, r \in \text{Act}^{*}$, to represent objects such as a finite prefix of a system run. A trace and a finite trace with action $a$ at their head are denoted by $at$ and $as$, respectively. We write $st$ for a trace that has prefix $s$ and suffix $t$. A similar notation is used for prefixes and suffixes of finite traces. The empty finite trace $\varepsilon$ is a prefix of every (finite) trace.

## 2.1 Syntax and semantics of recHML

As mentioned in the introduction, we use Hennessy-Milner logic with recursion recHML [19, 63] as our touchstone specification language. This is the collection

**Syntax**

$$
\begin{array}{lllll}
\varphi, \psi \in \textsc{recHML} ::= & \mathsf{tt} & \text{(truth)} & \mid \mathsf{ff} & \text{(falsehood)} \\
\mid & \varphi \vee \psi & \text{(disjunction)} & \mid \varphi \wedge \psi & \text{(conjunction)} \\
\mid & \langle a \rangle \varphi & \text{(possibility)} & \mid [a]\varphi & \text{(necessity)} \\
\mid & \mathsf{min}\, X.\varphi & \text{(min. fixed point)} & \mid \mathsf{max}\, X.\varphi & \text{(max. fixed point)} \\
\mid & X & \text{(rec. variable)} & &
\end{array}
$$

**Linear-Time Semantics**

$$
\begin{array}{rcll}
[\![\mathsf{tt}]\!]\rho & \stackrel{\text{def}}{=} & \textsc{Trc} & [\![\mathsf{ff}]\!]\rho \stackrel{\text{def}}{=} \emptyset \\
[\![\varphi_1 \wedge \varphi_2]\!]\rho & \stackrel{\text{def}}{=} & [\![\varphi_1]\!]\rho \cap [\![\varphi_2]\!]\rho & \\
[\![\varphi_1 \vee \varphi_2]\!]\rho & \stackrel{\text{def}}{=} & [\![\varphi_1]\!]\rho \cup [\![\varphi_2]\!]\rho & \\
[\![[a]\varphi]\!]\rho & \stackrel{\text{def}}{=} & \{t \mid \forall u \cdot t = au \text{ implies } u \in [\![\varphi]\!]\rho\} & \\
[\![\langle a \rangle \varphi]\!]\rho & \stackrel{\text{def}}{=} & \{t \mid \exists u \cdot t = au \text{ and } u \in [\![\varphi]\!]\rho\} & \\
[\![\mathsf{min} X.\varphi]\!]\rho & \stackrel{\text{def}}{=} & \bigcap \{T \mid [\![\varphi]\!]\rho[X \mapsto T] \subseteq T\} & \\
[\![\mathsf{max} X.\varphi]\!]\rho & \stackrel{\text{def}}{=} & \bigcup \{T \mid T \subseteq [\![\varphi]\!]\rho[X \mapsto T]\} & [\![X]\!]\rho \stackrel{\text{def}}{=} \rho(X)
\end{array}
$$

Figure 2: Syntax and linear-time semantics of recHML.

of formulae generated by the grammar in Figure 2. Such formulae are built from a countably infinite set of logical variables $X, Y \in \textsc{LVar}$, which are used to define recursive formulae expressing least or greatest fixed points. Formulae $\mathsf{min}\, X.\varphi$ and $\mathsf{max}\, X.\varphi$ bind free occurrences of the logical variable $X$ in $\varphi$, inducing the usual notions of open/closed formulae and formula equality up to alpha-conversion. In what follows, we tacitly restrict ourselves to considering only closed formulae.

Apart from the standard Boolean constructs and the fixed-point operators, the logic is equipped with action-labelled possibility and necessity modalities from classic Hennessy-Milner logic [56].

The function $[\![-]\!]$ in Figure 2 gives the denotational semantics of recHML. It maps a formula $\varphi$ to the set $[\![\varphi]\!]$ of the traces that satisfy $\varphi$, and gives the *linear-time* semantics of recHML. It uses valuations that map logical variables to sets of traces, $\rho : \textsc{LVar} \to \mathcal{P}(\textsc{Trc})$, to define the semantics by induction on the structure of the formulae. The notation $\rho[X \mapsto T]$ denotes the valuation that maps $X$ to $T$ and agrees with $\rho$ on all the other logical variables. It is well known that the choice of the valuation $\rho$ is immaterial for a closed formula $\varphi$. Therefore, we write $[\![\varphi]\!]$ for the set of traces that satisfy a closed formula $\varphi$.

Intuitively, $\rho(X)$ is the set of traces assumed to satisfy $X$. The cases for the Boolean operators are standard. An existential modal formula $\langle a \rangle \varphi$ denotes all

traces of the form $at$ where $t$ satisfies $\varphi$. A universal modal formula $[a]\varphi$ denotes all traces that do not start with $a$ or have the form $au$ for some $u$ satisfying $\varphi$. The sets of traces satisfying the least and greatest fixed-point formulae, $\mathsf{min}\, X.\varphi$ and $\mathsf{max}\, X.\varphi$, are defined as the intersection (respectively, union) of all the pre-fixed points (respectively, post-fixed points) of the endofunction over the powerset of Trc induced by the formula $\varphi$. Intuitively, greatest fixed-point formulae may be used to describe safety properties—that is, properties that are satisfied by a trace $t$ unless there is some finite prefix of $t$ that provides evidence to the contrary. On the other hand, least fixed-point formulae express liveness properties—that is, properties that are satisfied by a trace $t$ only if there is some finite prefix of $t$ that witnesses that fact [75]. By way of example, as our readers might want to check, $[\![\mathsf{max}\, X.\langle a\rangle X]\!] = \{a^\omega\} = [\![\mathsf{max}\, X.(\langle a\rangle X \wedge X)]\!]$ whereas $[\![\mathsf{min}\, X.\langle a\rangle X]\!] = \emptyset$. Moreover, assuming that $\text{Act} = \{a, b\}$, the set $[\![\mathsf{min}\, X.(\langle a\rangle \mathsf{tt} \vee \langle b\rangle X)]\!]$ contains all traces apart from $b^\omega$; thus, the formula $\mathsf{min}\, X.(\langle a\rangle \mathsf{tt} \vee \langle b\rangle X)$ expresses the liveness property 'the trace eventually contains an $a$'.

Two formulae $\varphi$ and $\psi$ in recHML are *logically equivalent* (over Trc) when $[\![\varphi]\!] = [\![\psi]\!]$.

*Remark.* It is well known that, even though it does not include a negation operator, the logic recHML is semantically closed under negation—that is, for every formula $\varphi$ in recHML there is a formula $\overline{\varphi}$ in recHML such that $[\![\overline{\varphi}]\!] = \text{Trc} \setminus [\![\varphi]\!]$. The use of a logic without negation ensures that all formulae define monotonic endofunctions over $\mathcal{P}(\text{Trc})$, avoiding the need for syntactic restrictions over the collection of formulae.

A formula is *guarded* if every occurrence of each fixed-point variable appears within the scope of a modality within its fixed-point binding. For example, the formulae $\mathsf{max}\, X.\langle a\rangle X$ and $\mathsf{min}\, X.(\langle b\rangle \mathsf{tt} \vee \langle a\rangle X)$ are guarded, but $\mathsf{max}\, X.(\langle a\rangle X \wedge X)$ is not. Without loss of expressiveness, we assume that all formulae are guarded. Indeed, each formula is logically equivalent to a guarded one [35, 62].

*Remark.* The definitions of $[\![-]\!]$ and of logical equivalence apply equally well to other classic trace-based domains such as the collection of finite traces $\text{Act}^*$ and that of finite and infinite traces $\text{Act}^* \cup \text{Act}^\omega$. The latter domain was dubbed the set of *finfinite traces* in [13].

*Remark.* Over the domain of infinite traces built from a finite set of actions Act, the modal operator $[a]$ can be expressed using $\langle a\rangle$ as follows:

$$[a]\varphi = \langle a\rangle\varphi \vee \bigvee_{b \in \text{Act}\setminus\{a\}} \langle b\rangle \mathsf{tt}.$$

However, this fails over finite or finfinite traces. For example, over those domains, there is no formula that does not use the necessity modal operators and is logically

equivalent to $[a]\mathsf{ff}$, which expresses the fact that a trace does not start with the action $a$. Indeed, as our readers can check, every formula that does not use the necessity modal operators and is satisfied by the empty trace is a tautology.

## 2.2 What are the monitorable recHML formulae?

Now that we have introduced our touchstone specification language recHML, we are ready to study which properties expressible in it are monitorable and with which correctness guarantees. However, in order to do so, we first need to define precisely when a formula in recHML is monitorable over Trc.

As we mentioned in the introduction, one of our tenets is that the notion of monitorability should be defined in terms of the computational devices that carry our the monitoring process, viz. the monitors. As depicted in Figure 1, a monitor may reach any one of *three* verdicts after analysing a finite trace: *acceptance*, which we denote by yes, *rejection*, which we write as no, and the *inconclusive* verdict, written end. Following [73], verdicts are *irrevocable*, meaning that a monitor cannot change its mind after it has issued a verdict.

At an abstract level, a monitor $m$ can therefore be fully characterised by the set $A(m)$ of finite traces for which it issues an acceptance verdict yes and the set $R(m)$ of finite traces for which it issues a rejection verdict no. Since verdicts are irrevocable, the sets $A(m)$ and $R(m)$ are *extension closed*, that is, they satisfy the following natural closure properties:

- if $s \in A(m)$ then $sw \in A(m)$ for each $w \in \text{Act}^*$, and
- if $s \in R(m)$ then $sw \in R(m)$ for each $w \in \text{Act}^*$.

Another sanity criterion for monitors is that they be *consistent*, namely that $A(m)$ and $R(m)$ are disjoint. Consistent monitors cannot accept and reject the same finite trace.

For the moment, we assume that each monitor $m$ is described by two sets of finite traces $A(m)$ and $R(m)$ satisfying the aforementioned closure properties. We will then see how those sets can be associated to monitors described using a process-algebraic language using their operational semantics, which details how monitors evolve when observing a trace describing the observable content of a system run.

**Definition 2.1.** A monitor $m$ *accepts* a trace $t \in \text{Trc}$ iff $t = su$ for some $s \in A(m)$ and $u \in \text{Trc}$. Similarly, a monitor $m$ *rejects* a trace $t \in \text{Trc}$ iff $t = su$ for some $s \in R(m)$ and $u \in \text{Trc}$.

*Remark.* The definition of when a monitor accepts or rejects a trace given in [13] is based on a binary operation, called *instrumentation*, whose structural operational

semantics describes how a monitor evolves when it observes a trace. That definition is equivalent to the one given in Definition 2.1. The same applies to the corresponding notions we use in Sections 3 and 4.

Here, we eschewed the introduction of different instrumentation operations to reduce the amount of technical machinery in the presentation of our results.

We are now ready to define the key concepts of monitor soundness and (partial) completeness with respect to a formula in recHML.

**Definition 2.2** (Monitor Soundness and (Partial) Completeness over Traces)**.** Let $\varphi$ be closed formula in recHML.

- A monitor $m$ is *sound* for $\varphi$ if for all $t \in \text{Trc}$:
  - if $m$ accepts $t$ then $t \in [\![\varphi]\!]$;
  - if $m$ rejects $t$ then $t \notin [\![\varphi]\!]$.
- A monitor $m$ is *satisfaction-complete* for $\varphi$ if for all $t \in \text{Trc}$, if $t \in [\![\varphi]\!]$ then $m$ accepts $t$. It is *violation-complete* for $\varphi$ if for all $t \in \text{Trc}$, if $t \notin [\![\varphi]\!]$ then $m$ rejects $t$.
- A monitor $m$ is *complete* for $\varphi$ if it is both violation- and satisfaction-complete for it.

Soundness of a monitor for a formula is a non-negotiable requirement if monitors are to be used in a verification setting. Moreover, as our readers can easily check, it guarantees that a monitor is consistent. However, soundness alone is a very weak correctness guarantee; indeed, a monitor that neither accepts nor rejects any trace is sound for every property. The three flavours of completeness described above provide more monitor-correctness guarantees; they reflect the fact that runtime monitoring is, in general, less expressive than verification techniques such as model checking and, therefore, completeness might be unattainable using it for sufficiently powerful specification languages. Hence, to our mind, it is crucial to characterise the collection of properties that can be monitored in a sound and complete, violation- or satisfaction-complete way. Before presenting such characterisations, we introduce an operational model of monitors that we have employed to obtain those characterisation results.

The language we use to describe monitors, $m, n \in \text{Mon}$, is defined by the grammar and the transition rules in Figure 3. It is a variation on the restriction- and relabelling-free fragment of Milner's CCS [68], where we use verdicts $v \in \{\mathsf{yes}, \mathsf{no}, \mathsf{end}\}$ in lieu of the nil process and we endow monitors with conjunctive parallelism, $\otimes$, and disjunctive parallelism, $\oplus$. Intuitively, the $\otimes$ operator over

**Syntax of monitors**

$$m, n \in \textsc{Mon} ::= v \mid a.m \mid m + n \mid \mathsf{rec}\, x.m \mid x \mid m \otimes n \mid m \oplus n$$

**Structural operational semantics of monitors**

$$\textsc{mAct}\ \frac{}{a.m \xrightarrow{a} m} \qquad \textsc{mRec}\ \frac{}{\mathsf{rec}\, x.m \xrightarrow{\tau} m[\mathsf{rec}\, x.m/x]}$$

$$\textsc{mSelL}\ \frac{m \xrightarrow{\mu} m'}{m + n \xrightarrow{\mu} m'} \qquad \textsc{mSelR}\ \frac{n \xrightarrow{\mu} n'}{m + n \xrightarrow{\mu} n'} \qquad \textsc{mVer}\ \frac{}{v \xrightarrow{a} v}$$

$$\textsc{mPar}\ \frac{m \xrightarrow{a} m' \quad n \xrightarrow{a} n'}{m \odot n \xrightarrow{a} m' \odot n'} \qquad \textsc{mTauL}\ \frac{m \xrightarrow{\tau} m'}{m \odot n \xrightarrow{\tau} m' \odot n} \qquad \textsc{mVrE}\ \frac{}{\mathsf{end} \odot \mathsf{end} \xrightarrow{\tau} \mathsf{end}}$$

$$\textsc{mVrC1}\ \frac{}{\mathsf{yes} \otimes m \xrightarrow{\tau} m} \qquad \textsc{mVrC2}\ \frac{}{\mathsf{no} \otimes m \xrightarrow{\tau} \mathsf{no}}$$

$$\textsc{mVrD1}\ \frac{}{\mathsf{no} \oplus m \xrightarrow{\tau} m} \qquad \textsc{mVrD2}\ \frac{}{\mathsf{yes} \oplus m \xrightarrow{\tau} \mathsf{yes}}$$

Figure 3: Syntax and semantics of monitors.

monitors plays the role of conjunction over formulae in that a monitor of the form $m \otimes n$ can only reach the acceptance verdict yes if both $m$ and $n$ can do so. Similarly, the $\oplus$ operator over monitors is akin to disjunction over formulae in that a monitor of the form $m \oplus n$ can only reach the verdict yes if $m$ or $n$ can do so. Figure 3 presents the structural operational semantics of parallel monitors that formalises their behaviour. (We use the notation $\odot$ to stand for $\otimes$ or $\oplus$, that is, $\odot \in \{\otimes, \oplus\}$.) Rule mPar states that *both* submonitors need to be able to analyse an external action $a$ for their parallel composition to transition with that action. The rules in Figure 3 also allow $\tau$-transitions for the reconfiguration of parallel compositions of monitors. For instance, rules mVrC1 and mVrC2 describe the fact that, whereas yes verdicts are uninfluential in conjunctive parallel compositions, no verdicts supersede the verdicts of other monitors in conjunctive parallel compositions. (Figure 3 omits the symmetric rules.) The dual applies for yes and no verdicts in a disjunctive parallel composition, as described by rules mVrD1 and mVrD2. Rule mVrE applies to both forms of parallel composition and consolidates multiple

inconclusive verdicts. Rules mTauL and its dual mTauR (omitted) are contextual rules for these monitor reconfiguration steps. Finally, we highlight the transition rule for verdicts, describing the fact that, from a verdict state, any action can be analysed by transitioning to the same state; verdicts are thus *irrevocable*. The other rules are standard.

*Remark.* The conclusion of the transition rule for verdicts could be generalised to $v \xrightarrow{\mu} v$ as done in [6]. Doing so would make the algebraic theory of monitors slightly more elegant without changing any of the results we present in this section. For the sake of consistency with previous work, here we decided to stick to the rule given in [13, 50, 51].

Using the operational semantics of monitors, we can now define the sets of finite traces $A(m)$ and $R(m)$ associated with a monitor $m$. In what follows, we write $m \overset{s}{\Rightarrow} m'$ for some monitors $m, m'$ and $s \in \text{Act}^*$ when there is a sequence of transitions leading from $m$ to $m'$ whose observable content is equal to $s$. For instance, $m \overset{\varepsilon}{\Rightarrow} m'$ means that $m$ can transition to $m'$ by performing a sequence of $\tau$-labelled transitions and $m \overset{a}{\Rightarrow} m'$ holds when there are $m_1$ and $m_2$ such that $m \overset{\varepsilon}{\Rightarrow} m_1 \xrightarrow{a} m_2 \overset{\varepsilon}{\Rightarrow} m'$.

**Definition 2.3.** For each $m \in \text{Mon}$, we define

$$A(m) = \{s \mid m \overset{s}{\Rightarrow} \mathsf{yes}\} \quad \text{and} \quad R(m) = \{s \mid m \overset{s}{\Rightarrow} \mathsf{no}\}.$$

**Example 2.1.** As we observed earlier, $[\![\mathsf{max}\, X.\langle a\rangle X]\!] = \{a^\omega\}$ and therefore the formula $\mathsf{max}\, X.\langle a\rangle X$ expresses the fact that a system run only produces action $a$. Assume, for the sake of simplicity, that $\text{Act} = \{a, b\}$. Consider the monitor $m = \mathsf{rec}\, x.(a.x + b.\mathsf{no})$. It is not hard to see that $A(m)$ is empty and $R(m)$ contains all the strings in $\text{Act}^*$ that have at least one occurrence of $b$. This means that $m$ is sound and violation-complete for $\mathsf{max}\, X.\langle a\rangle X$. On the other hand, there is no monitor that is sound and satisfaction-complete for $\mathsf{max}\, X.\langle a\rangle X$. Indeed, any satisfaction-complete monitor $n$ for that formula would have to accept the trace $a^\omega$, since that trace satisfies $\mathsf{max}\, X.\langle a\rangle X$. This means that $a^k \in A(n)$ for some $k \geq 0$. Therefore, $n$ would also accept the trace $a^k b^\omega$ and would be unsound. It follows that there is no sound and complete monitor for $\mathsf{max}\, X.\langle a\rangle X$.

Consider now the formula $\mathsf{min}\, X.(\langle a\rangle \mathsf{tt} \vee \langle b\rangle X)$. As we mentioned earlier in the paper, assuming that $\text{Act} = \{a, b\}$, that formula is satisfied by every trace apart from $b^\omega$. Consider the monitor $m = \mathsf{rec}\, x.(a.\mathsf{yes} + b.X)$. It is not hard to see that $R(m)$ is empty and $A(m)$ contains all the strings in $\text{Act}^*$ that have at least one occurrence of $a$. This means that $m$ is sound and satisfaction-complete for $\mathsf{min}\, X.(\langle a\rangle \mathsf{tt} \vee \langle b\rangle X)$.

On the other hand, reasoning as above, our readers can convince themselves that there is no monitor that is sound and violation-complete for that formula.

As we shall see shortly, the non-existence of complete monitors for the formulae we have used in this example is an instance of a general phenomenon. Indeed, only formulae whose satisfaction can be determined by observing a bounded prefix of a trace can be monitored in a sound and complete way.

We are now ready to present the characterisation of the formulae in recHML that have sound and (partially) complete monitors. Below we write HML for the collection of recHML formulae that do not use fixed-point operators (that is, the formulae in classic Hennessy-Milner logic), maxHML for the set of recHML formulae that do not use the least-fixed-point operator and minHML for the fragment of recHML consisting of the formulae without occurrences of the greatest-fixed-point operator.

**Theorem 2.1** (Aceto et al. [13])**.** *Let* $\varphi \in$ recHML.

1. *There is a sound and complete monitor* $m \in$ Mon *for* $\varphi$ *iff* $\varphi$ *is logically equivalent to a formula in* HML.

2. *There is a sound and violation-complete monitor* $m \in$ Mon *for* $\varphi$ *iff* $\varphi$ *is logically equivalent to a formula in* maxHML.

3. *There is a sound and satisfaction-complete monitor* $m \in$ Mon *for* $\varphi$ *iff* $\varphi$ *is logically equivalent to a formula in* minHML.

The proofs of the above-mentioned results are based on syntax-directed constructions that produce monitors with the stated correctness guarantees from formulae in the relevant fragments of recHML—see [13, Definitions 4.4 and 4.12]. For example, the following function constructs a sound and complete monitor from a formula $\varphi$ in HML:

$$\mathsf{m}(\mathsf{ff}) \stackrel{\text{def}}{=} \mathsf{no} \qquad \mathsf{m}(\varphi_1 \wedge \varphi_2) \stackrel{\text{def}}{=} \mathsf{m}(\varphi_1) \otimes \mathsf{m}(\varphi_2) \qquad \mathsf{m}([a]\varphi) \stackrel{\text{def}}{=} a.\mathsf{m}(\varphi) + \sum_{b \neq a} b.\mathsf{yes}$$

$$\mathsf{m}(\mathsf{tt}) \stackrel{\text{def}}{=} \mathsf{yes} \qquad \mathsf{m}(\varphi_1 \vee \varphi_2) \stackrel{\text{def}}{=} \mathsf{m}(\varphi_1) \oplus \mathsf{m}(\varphi_2) \qquad \mathsf{m}(\langle a \rangle \varphi) \stackrel{\text{def}}{=} a.\mathsf{m}(\varphi) + \sum_{b \neq a} b.\mathsf{no}.$$

In the above-mentioned reference, we also show how to convert monitors into the formulae in a given fragment for which they are sound and (partially) complete [13, Definitions 4.7 and 4.13]. Moreover, we prove that monitors that make no use of the parallel conjunction and parallel disjunction operators suffice to establish Theorem 2.1 and that those monitors can be determinised—see [13, Proposition 3.11]. Intuitively, that result is based on the fact that a parallel monitor $m$ of the type

we construct from formulae describes alternating automata recognising $A(m)$ and $R(m)$, which can then be converted into monitors that express NFAs and equivalent DFAs that accept $A(m)$ and $R(m)$. However, the price to be paid to carry out those conversions is hefty and unavoidable—see [13, Proposition 3.11] and [14].

Using non-constructive means, one can show a stronger version of Theorem 2.1(1) that applies to complete monitoring for any logic defined over traces. (See [13, Theorem 4.8].)

**Theorem 2.2.** *Let $m$ be a monitor from a monitoring system where*

- *verdicts are irrevocable, that is, $A(m)$ and $R(m)$ are extension closed, and*
- *acceptance and rejections are defined as in Definition 2.1.*

*For any property $\varphi$ with a trace interpretation (not necessarily syntactically represented using* recHML*), if $m$ is sound and complete for $\varphi$ then $\varphi$ is logically equivalent to some formula in* HML*.*

*Remark.* The above-mentioned monitorability results hold when formulae are interpreted over *infinite* traces. If we consider finfinite traces instead, the monitorability picture changes considerably. Indeed, it turns out that, up to logical equivalence,

- only the formulae tt and ff have sound and complete monitors (see Lemma 5.4 in [13]);
- the only formulae that have sound and violation-complete monitors are those expressible in the safety fragment of recHML, which contains only the Boolean constants, conjunction, necessity modalities and greatest fixed-point operators (see [13, Proposition 5.8]); and
- the only formulae that have sound and satisfaction-complete monitors are those expressible in the co-safety fragment of recHML, which contains only the Boolean constants, disjunction, possibility modalities and least fixed-point operators (see [13, Proposition 5.8]).

We refer readers interested in reading more about our results dealing with classic monitorability to [10–12, 14–16, 49, 50, 50], which study the branching- and the linear-time settings and their relationships.

The monitoring tool detectEr for Erlang programs[2] is inspired by the theoretical developments presented in some of those references. However, that tool deals with properties of programs whose behaviour is data dependent, which go beyond the classic setting we have described so far. This realisation motivates the next step in our journey, where we study some of the theoretical developments that underpin the practice of runtime monitoring for data-driven systems [8, 27].

[2] See `https://duncanatt.github.io/detecter/` and the references [7, 24, 25, 37–39].

# 3 Exhibit B: Monitoring systems with data

In this section, we apply our research approach to a setting with data. We define recHML$^d$, an extension of recHML tailored to express properties of traces of system executions that contain data values. We also present some results pertaining to the characterisation of monitorable fragments of that logic from [8].

Since we consider linear-time properties as we did in Section 2, we model system executions as data *ω-words*. Typically, those are infinite words whose elements are pairs consisting of a letter from a finite alphabet and a *data value* from an infinite data domain. Since the finite alphabet plays no role in our developments and can be simulated [70], we omit it for simplicity. A data word is thus an infinite sequence of values from an infinite data domain.

For the rest of this section, we fix a countably infinite *data domain* $\mathbb{D}$, whose only predicate is '=' and is decidable. Concrete examples of such data domains include the sets $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\{0,1\}^*$ with equality, amongst others. An infinite (respectively, finite) *trace* is a data word, that is, an infinite (respectively, finite) sequence $t \in \mathbb{D}^\omega$ (respectively, $w \in \mathbb{D}^n$ for some $n \in \mathbb{N}$); the set of all infinite traces is denoted $\text{Trc}^d = \mathbb{D}^\omega$ (respectively, $\text{FTrc}^d = \mathbb{D}^*$ for finite traces).

## 3.1 Syntax and semantics of recHML$^d$

To express properties of traces of data values, we use an extension of recHML, called recHML$^d$. Its syntax and semantics are described in Figure 4. The new ingredients in the syntax of recHML$^d$ are as follows. In addition to logical variables, formulae are built from a countably infinite set of data variables, $x, y \in \text{DVar}$, ranging over an infinite domain of data values, $d \in \mathbb{D}$. To reason about the data carried by process actions, modalities are augmented with decidable, quantifier-free Boolean *constraint expressions*, $b, c \in \text{BExp}$, defined over $\mathbb{D}$ and $\text{DVar} \cup \{\star\}$, where $\star \notin \text{DVar}$ is a placeholder variable for the current action $d \in \mathbb{D}$ produced by a system run. The free data variables $x \in \text{DVar}$ that appear in $b$ are bound by existential and universal quantification constructs $\exists\, \boldsymbol{x}.\varphi$ and $\forall\, \boldsymbol{x}.\varphi$. Intuitively, the formula $\exists\, \boldsymbol{x}.\varphi$ is satisfied by a trace $t$ if there is some $d \in \mathbb{D}$ such that $t$ satisfies $\varphi$ when the value of $x$ is set to $d$. On the other hand, a trace $t$ satisfies $\forall\, \boldsymbol{x}.\varphi$ when, for each $d \in \mathbb{D}$, $t$ satisfies $\varphi$ when the value of $x$ is set to $d$.

The formal definition of the semantics of recHML$^d$ requires several ingredients and is in the spirit of the one presented in [52] for a similar logic. We introduce them next to make the presentation self-contained. Readers who are not interested in the formal details of the semantics can gain an intuitive understanding of the constructs of recHML$^d$ and of their expressiveness by reading the formulae in

**recHML$^d$ Syntax**

$$\varphi, \psi \in \text{recHML}^d ::= \mathsf{tt} \mid \mathsf{ff} \mid \langle b \rangle \varphi \mid [b] \varphi \mid \exists \boldsymbol{x}.\varphi \mid \forall \boldsymbol{x}.\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \mathsf{min}\, X.(\varphi) \mid \mathsf{max}\, X.(\varphi) \mid X$$

**Fragments**

$$\varphi, \psi \in \text{cHML}^d ::= \mathsf{tt} \mid \langle b \rangle \varphi \mid \exists \boldsymbol{x}.\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \mathsf{min}\, X.(\varphi) \mid X$$

$$\varphi, \psi \in \text{sHML}^d ::= \mathsf{ff} \mid [b] \varphi \mid \forall \boldsymbol{x}.\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \mathsf{max}\, X.(\varphi) \mid X$$

$$\varphi, \psi \in \text{disjHML}^d ::= \mathsf{tt} \mid \langle b \rangle \varphi \mid \exists \boldsymbol{x}.\varphi \mid \varphi \vee \psi \mid \mathsf{min}\, X.(\varphi) \mid X$$

$$\varphi, \psi \in \text{HML}^d ::= \mathsf{tt} \mid \mathsf{ff} \mid \langle b \rangle \varphi \mid [b] \varphi \mid \exists \boldsymbol{x}.\varphi \mid \forall \boldsymbol{x}.\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi$$

**Semantics**

$$[\![\, \mathsf{tt} \,]\!]^\rho_\delta \stackrel{\text{def}}{=} \text{Trc}^d \qquad [\![\, \mathsf{ff} \,]\!]^\rho_\delta \stackrel{\text{def}}{=} \varnothing \qquad [\![\, X \,]\!]^\rho_\delta \stackrel{\text{def}}{=} (\rho(X))(\delta)$$

$$[\![\, \langle b \rangle \varphi \,]\!]^\rho_\delta \stackrel{\text{def}}{=} \{t \mid (\exists u, d.t = du \text{ and } b\delta[\star \mapsto d] \Downarrow \mathsf{true} \text{ and } u \in [\![\, \varphi \,]\!]^\rho_\delta)\}$$

$$[\![\, [b] \varphi \,]\!]^\rho_\delta \stackrel{\text{def}}{=} \{t \mid (\forall u, d.(t = du \text{ and } b\delta[\star \mapsto d] \Downarrow \mathsf{true}) \text{ implies } u \in [\![\, \varphi \,]\!]^\rho_\delta)\}$$

$$[\![\, \exists \boldsymbol{x}.\varphi \,]\!]^\rho_\delta \stackrel{\text{def}}{=} \bigcup_{d \in \mathbb{D}} [\![\, \varphi \,]\!]^\rho_{\delta[x \mapsto d]}$$

$$[\![\, \forall \boldsymbol{x}.\varphi \,]\!]^\rho_\delta \stackrel{\text{def}}{=} \bigcap_{d \in \mathbb{D}} [\![\, \varphi \,]\!]^\rho_{\delta[x \mapsto d]}$$

$$[\![\, \varphi \vee \psi \,]\!]^\rho_\delta \stackrel{\text{def}}{=} [\![\, \varphi \,]\!]^\rho_\delta \cup [\![\, \psi \,]\!]^\rho_\delta$$

$$[\![\, \varphi \wedge \psi \,]\!]^\rho_\delta \stackrel{\text{def}}{=} [\![\, \varphi \,]\!]^\rho_\delta \cap [\![\, \psi \,]\!]^\rho_\delta$$

$$[\![\, \mathsf{min}\, X.(\varphi) \,]\!]^\rho_\delta \stackrel{\text{def}}{=} (\bigsqcap \{F \mid \lambda\delta'.[\![\, \varphi \,]\!]^{\rho[X \mapsto F]}_{\delta'} \sqsubseteq F\})(\delta)$$

$$[\![\, \mathsf{max}\, X.(\varphi) \,]\!]^\rho_\delta \stackrel{\text{def}}{=} (\bigsqcup \{F \mid F \sqsubseteq \lambda\delta'.[\![\, \varphi \,]\!]^{\rho[X \mapsto F]}_{\delta'}\})(\delta)$$

**Expressions**

$$b, c \in \text{BExp} ::= \mathsf{true} \mid e = f \mid \neg b \mid b \wedge c \qquad e, f \in \text{Exp} ::= x \in \text{DVar} \mid \star$$

Figure 4: Syntax and linear-time semantics of recHML$^d$.

Example 3.1 and the natural-language descriptions of the properties they describe given there. The intuition provided there suffices to grasp the gist of the main results we present in this section.

We define the domains $\text{DEnv} = \text{DVar} \rightharpoonup \mathbb{D}$ of data environments, $\text{DInt} = \text{DEnv} \rightharpoonup 2^{\text{Trc}^d}$ of data interpretations, and $\text{TEnv} = \text{LVar} \rightharpoonup \text{DInt}$ of trace environments (where $A \rightharpoonup B$ denotes the set of partial functions from set $A$ to set $B$). A *data environment*, $\delta \in \text{DEnv}$, is a partial function with a finite domain mapping data variables to values from $\mathbb{D}$; analogously, a *trace environment*, $\rho \in \text{TEnv}$, maps logical variables to data interpretations $F, G \in \text{DInt}$, that given $\delta$, return a set of traces, whose intended meaning is the interpretation of the logical variable in the data environment $\delta$.

The *linear-time* semantics of recHML$^d$ is given by the denotational semantic function $[\![ - ]\!]$ defined inductively in Figure 4. Formulae are interpreted with respect to a trace environment $\rho$ that gives meaning to logical variables, and a data environment $\delta$ that assigns values to data variables in Boolean constraint expressions. We write $[\![ \varphi ]\!]^{\rho}_{\delta}$ for the set of traces that satisfy $\varphi$ with respect to $\rho$ and $\delta$.

An expression $b$ defines a set of *external* system actions with respect to a data environment $\delta$. An action $d$ is in this set when the data value it carries satisfies $b$ modulo $\delta$, written $b\delta \Downarrow \mathsf{true}$. Possibility formulae $\langle\, b \,\rangle \varphi$ denote all the traces $t = du$ that begin with an action $d$ that is in the action set described by $b\delta$ *and* whose tail $u$ satisfies the continuation formula $\varphi$. Dually, necessity formulae $[\, b \,] \varphi$ describe all the traces that, *whenever* they begin with an action $d$ that is in the action set described by $b\delta$, continue with a trace that satisfies $\varphi$. Note that, in the linear-time setting, necessity can be expressed as possibility: $[\, b \,] \varphi \equiv \langle\, \neg b \,\rangle \mathsf{tt} \vee \langle\, b \,\rangle \varphi$, and dually $\langle\, b \,\rangle \varphi = [\, \neg b \,] \mathsf{ff} \wedge [\, b \,] \varphi$. The existential quantifier $\exists \boldsymbol{x}.\varphi$ is interpreted as the set of traces that satisfy $\varphi$ by assigning *some* $d \in \mathbb{D}$ to $x$; the universal quantifier $\forall\, \boldsymbol{x}.\varphi$ is the set of traces satisfying $\varphi$ under *all* such assignments. Formulae are only interpreted with respect to data environments whose domain includes the set of free data variables occurring in them.

Since the logic does not have an explicit negation operator, for all $\varphi$ the semantic function $[\![ \varphi ]\!]^{\rho}_{\delta}$ is monotonic in $\rho$ over the complete lattice $(\text{DInt}, \sqsubseteq)$, where the partial order $\sqsubseteq$ corresponds to graph inclusion. Formally, it is defined, for all $F, G \in \text{DInt}$, as $F \sqsubseteq G$ whenever $\forall \delta \in \text{DEnv}.F(\delta) \subseteq G(\delta)$. As is standard in the modal $\mu$-calculus, recursion is interpreted through fixed points: by the Knaster-Tarski theorem [79], $\mathsf{min}\ X.(\varphi)$ and $\mathsf{max}\ X.(\varphi)$ respectively correspond to the least and greatest fixed point of the operator that maps a data interpretation $F : \text{DEnv} \to 2^{\text{Trc}^d}$ to the data interpretation $\delta \mapsto [\![ \varphi ]\!]^{\rho[x \mapsto F]}_{\delta}$. This is the analogue of the operator used to define the semantics of recHML over traces in Section 2, lifted to the case of infinite alphabets by parameterising the interpretation by a data environment, in the spirit of [52]. To obtain the sought interpretation for $\mathsf{min}\ X.(\varphi)$

and $\mathsf{max}\, X.(\varphi)$, one then applies the least (respectively, greatest) fixed point of this operator (which is a function from data environments to sets of traces) to the current data environment $\delta$.

**Example 3.1.** To give an intuition of the logic and its expressiveness, we present a few elementary recHML$^d$ properties, along with their respective fragments, and discuss their meaning informally.

- The following formula in HML$^d$ states that the first and the second data values in a trace are equal:

$$\varphi_{\text{f=s}} \stackrel{\text{def}}{=} \exists\, \boldsymbol{x}.\langle\, x = \star \,\rangle \langle\, x = \star \,\rangle \mathsf{tt}. \tag{1}$$

  Indeed, the only way for the first modality $\langle\, x = \star \,\rangle$ to be satisfied is if $x$ takes the value of the first data value. Then, the second modality $\langle\, x = \star \,\rangle$ is satisfied exactly when the second value is equal to $x$, and hence to the first value.

- The following formula in cHML$^d$ expresses the requirement that the first data value appears again in a trace:

$$\varphi_{\text{leak}} \stackrel{\text{def}}{=} \exists\, \boldsymbol{x}.\langle\, x = \star \,\rangle \mathsf{min}\, X.(\langle\, x = \star \,\rangle \mathsf{tt} \vee \langle\, x \neq \star \,\rangle X), \tag{2}$$

  where we use $x \neq \star$ to abbreviate $\neg(x = \star)$. As above, $x$ stores the first data value. Then, we use recursion to look for another occurrence of that value. Intuitively, upon encountering a fixed-point variable $X$ the formula recurses, that is, we unfold the formula by replacing $X$ with the whole $\mathsf{min}\, X.(\varphi)$ that encloses it. Here, the formula recurses while it encounters values satisfying $x \neq \star$, and is satisfied (reaching $\mathsf{tt}$) if it encounters a value satisfying $x = \star$, viz. the first value in the trace. Since this is a least-fixed-point formula (denoted by the use of $\mathsf{min}$), the formula is satisfied only if it recurses finitely many times, which happens exactly when the first value appears again in a trace.

- The following formula in disjHML$^d$ expresses the requirement that *some* data value appears at least twice in a trace:

$$\begin{aligned} \varphi_3 &\stackrel{\text{def}}{=} \exists\, \boldsymbol{x}.\varphi(x) \text{ where} \\ \varphi(x) &\stackrel{\text{def}}{=} \mathsf{min}\, X.(\langle\, x = \star \,\rangle \psi(x) \vee \langle\, x \neq \star \,\rangle X) \\ \psi(x) &\stackrel{\text{def}}{=} \mathsf{min}\, Y.(\langle\, x = \star \,\rangle \mathsf{tt} \vee \langle\, x \neq \star \,\rangle Y) \end{aligned} \tag{3}$$

  For a given value of $x$, the formula is satisfied only if this value is found once (first disjunct of $\varphi(x)$) and then again (first disjunct of $\psi(x)$). Overall, the formula is satisfied whenever there exists such a value, which thus appears twice.

- The following formula states that all data values appearing in a trace are pairwise distinct (negation by dualisation of the DISJHML$^d$ formula above):

$$\begin{aligned} \varphi_4 &\stackrel{\text{def}}{=} \forall\, \boldsymbol{x}.\varphi(x) \text{ where} \\ \varphi(x) &\stackrel{\text{def}}{=} \mathsf{max}\, X.\, ([\, x = \star\,]\, \psi(x) \wedge [\, x \neq \star\,]\, X) \\ \psi(x) &\stackrel{\text{def}}{=} \mathsf{max}\, Y.\, ([\, x = \star\,]\, \mathsf{ff} \wedge [\, x \neq \star\,]\, Y) \end{aligned} \tag{4}$$

  Dually to the one we just discussed, this formula is *not* satisfied whenever some value appears twice in a trace.

- The following formula in RECHML$^d$ states that there exists a data value that never appears:

$$\varphi_5 \stackrel{\text{def}}{=} \exists\, \boldsymbol{x}.\mathsf{max}\, X.\, ([\, x = \star\,]\, \mathsf{ff} \wedge [\, x \neq \star\,]\, X) \tag{5}$$

  As for $\varphi_4$, the greatest-fixed-point operator max allows one to forbid a data value (existentially guessed using the $\exists$ quantifier) from appearing in a trace. Indeed, once we have guessed a value for $x$, the formula

$$\mathsf{max}\, X.\, ([\, x = \star\,]\, \mathsf{ff} \wedge [\, x \neq \star\,]\, X)$$

  is satisfied unless that value appears in the trace, leading to a violation of the subformula $[\, x = \star\,]\, \mathsf{ff}$. Recall that a greatest-fixed-point formula is satisfied unless one of its finite unfoldings is violated by the trace under consideration.

As the formulae in the above example indicate, the logic RECHML$^d$ is very expressive. Indeed, it turns out that even some of its fragments that make a restricted use of fixed-point formulae are undecidable.

**Theorem 3.1** (Theorems 3–4 in [8])**.** *The satisfiability problem for* CHML$^d$ *and the validity problem for* DISJHML$^d$ *are undecidable.*

The above result, which is shown by adapting and sharpening the reduction used in the proof of [70, Theorem 18], paints a grim decidability picture for RECHML$^d$. By comparison, the satisfiability and validity problems for RECHML and the modal $\mu$-calculus are EXP-complete [5, 61]. Fortunately, however, as we will now see, those undecidability results do not prevent us from identifying monitorable fragments of that logic.

## 3.2 What are the monitorable RECHML$^d$ formulae?

Our goal now is to determine which properties over TRC$^d$ = $\mathbb{D}^\omega$ can be monitored and with what guarantees. As in the classic setting we considered in the previous

section, a monitor $m$ can be abstractly characterised by two sets of finite data words $A(m)$ and $R(m)$ included in $\mathbb{D}^*$ that are disjoint and extension closed. The definition of when a monitor accepts or rejects a trace in $\text{Trc}^d$ given in Definition 2.1 applies to our current setting mutatis mutandis, as do those of monitor soundness and of the various flavours of completeness with respect to a formula $\varphi \in \text{recHML}^d$ and to a property of traces $T \subseteq \text{Trc}^d$. We say that the above notions are *effective* when $m$ can be computed by a Turing machine from $\varphi$ or some other finitary description of a property $T$.

In the finite alphabet case, we saw that all completely monitorable properties of traces can be expressed in the fragment HML, which consists of the recHML formulae without fixed-point operators (Theorem 2.2). The proof of that result from [13] (see Theorem 4.8 in that reference) can be adapted to establish a counterpart of that theorem in the setting of properties of data words. We tame the infinity of the data domain by quotienting finite traces by bijections over $\mathbb{D}$.

**Theorem 3.2.** *Let $T \subseteq \text{Trc}^d$ be a set of traces that is stable under renamings (that is, for all bijections $\sigma : \mathbb{D} \to \mathbb{D}$, we have that $\sigma(T) = T$). $T$ is completely monitorable* iff *it can be expressed in* $\text{HML}^d$.

*Remark.* The set $[\![ \varphi ]\!]$ is stable under renamings for each formula $\varphi \in \text{recHML}^d$. Such sets are called *equivariant* in the theory of nominal sets [32, 71].

*Remark.* Theorem 3.2 does not hold if we consider the domain $(\mathbb{N}, <)$. Indeed, there, one can define the set $D = \{d_0 d_1 \dots d_n \# w \mid \forall i < j, d_i > d_j\}$, which is completely monitorable, since $n$ is bounded by $d_0$, but cannot be expressed in $\text{HML}^d$ since $n$ depends on $d_0$.

As Theorem 3.2 indicates, having to detect *all* satisfactions and *all* violations prevents us from monitoring for behaviours that can happen after an unbounded number of steps in a system execution. In what follows, we relax our notion of completeness and focus on satisfaction-completeness, the ability to detect all satisfactions of a property. Results dealing with violation-completeness are obtained by duality.

In Figure 5, we introduce a model of monitors, along with a compositional monitor-synthesis procedure from formulae in $\text{cHML}^d$ (defined in Figure 4). We encourage our readers to compare the syntax and semantics of monitors in the setting with data given in Figure 5 with those we introduced in Figure 3. In particular, note that the prefixing operator we employ in the setting with data corresponds to a conditional choice on a Boolean expression $b$ and matches the modalities in $\text{recHML}^d$. Moreover, the construct guess $x.m$ plays the role of quantification over data values in formulae. Since the behaviour of monitors depends on the current values of its data variables, the structural operational semantics of monitors

**Syntax**

$m, n \in \text{Mon} ::= \mathsf{yes} \mid \mathsf{end} \mid (b).m \mid \mathsf{guess}\ x.m \mid m \oplus n \mid m \otimes n \mid \mathsf{rec}\, x.m \mid x$

**Configurations** $c \in C ::= (m, \delta) \mid c \odot c$, where $m \in \text{Mon}$ is a monitor, $\delta \in \text{DEnv}$ is a data environment and $\odot$ is either $\oplus$ (parallel OR) or $\otimes$ (parallel AND).

**Small-Step Semantics**

$$\frac{v \in \{\mathsf{yes}, \mathsf{end}\}}{v,\ \delta \xrightarrow{d} v,\ \delta} \qquad \frac{b\delta[\star \mapsto d] \Downarrow \mathsf{true}}{(b).m,\ \delta \xrightarrow{d} m,\ \delta}\ d \in \mathbb{D} \qquad \frac{b\delta[\star \mapsto d] \Downarrow \mathsf{false}}{(b).m,\ \delta \xrightarrow{d} \mathsf{end},\ \delta}\ d \in \mathbb{D}$$

$$\frac{}{\mathsf{guess}\ x.m,\ \delta \xrightarrow{\tau} m,\ \delta[x \mapsto d]}\ d \in \mathbb{D} \qquad \frac{}{\mathsf{rec}\, x.m,\ \delta \xrightarrow{\tau} m[\mathsf{rec}\, x.m/x],\ \delta}$$

$$\frac{}{m \odot n,\ \delta \xrightarrow{\tau} m,\ \delta \odot n,\ \delta} \qquad \frac{c_1 \xrightarrow{d} c_1' \quad c_2 \xrightarrow{d} c_2'}{c_1 \odot c_2 \xrightarrow{d} c_1' \odot c_2'}$$

$$\frac{c_1 \xrightarrow{\tau} c_1'}{c_1 \odot c_2 \xrightarrow{\tau} c_1' \odot c_2} \qquad \frac{c_2 \xrightarrow{\tau} c_2'}{c_1 \odot c_2 \xrightarrow{\tau} c_1 \odot c_2'}$$

$$\frac{}{\mathsf{yes},\ \delta \otimes c \xrightarrow{\tau} c} \qquad \frac{}{\mathsf{yes},\ \delta \oplus c \xrightarrow{\tau} \mathsf{yes},\ \delta}$$

**Monitor Synthesis**

$$\mathsf{m}(\mathsf{tt}) = \mathsf{yes} \quad \mathsf{m}(\exists\, \boldsymbol{x}.\varphi) = \mathsf{guess}\ x.\mathsf{m}(\varphi) \quad \mathsf{m}(\langle\, b\,\rangle\, \varphi) = (b).\mathsf{m}(\varphi)$$

$$\mathsf{m}(\varphi \vee \psi) = \mathsf{m}(\varphi) \oplus \mathsf{m}(\psi) \quad \mathsf{m}(\varphi \wedge \psi) = \mathsf{m}(\varphi) \otimes \mathsf{m}(\psi)$$

$$\mathsf{m}(\mathsf{min}\ X.\ (\varphi)) = \mathsf{rec}\, x.\mathsf{m}(\varphi) \qquad \mathsf{m}(X) = x$$

Figure 5: Syntax, small-step semantics and synthesis of monitors.

employs *configurations*, which consist of parallel compositions of monitors, each with its own data environment $\delta$ that assigns values to its data variables.

As we did earlier in the classic setting, using the operational semantics of monitors, we can now define the set of finite traces $A(m) \subseteq \mathbb{D}^*$ associated with a monitor $m$. In what follows, we write $m, \varnothing \stackrel{s}{\Rightarrow} c$ for some monitor $m$, finite trace $s \in \mathbb{D}^*$ and configuration $c$ when there is a sequence of transitions leading from $m, \varnothing$ to $c$ whose observable content is equal to $s$.

**Definition 3.1.** For each $m \in \textsc{Mon}$, we define

$$A(m) = \{s \mid m, \varnothing \stackrel{s}{\Rightarrow} \mathsf{yes}, \delta, \text{ for some } \delta\}.$$

The notions of monitor soundness and satisfaction-completeness are defined using $A(m)$ as we did in the classic setting in Definition 2.2.

We now show that, as the following example indicates in a specific setting, the monitoring system from Figure 5 yields sound and satisfaction-complete monitors for formulae in $\text{cHML}^d$. Note that this fragment of $\textsc{rec}\text{HML}^d$ includes conjunctions, and it can express $\mathsf{ff}$ as, for example, the formula $\langle \bot \rangle \mathsf{tt}$ (where $\bot$ stands for $x \neq x$) and (linear-time) necessity modalities.

**Example 3.2.** Consider a server that issues identifier tokens. Assume that the first token it issues is private and should not be leaked, that is, the server should *not* satisfy the formula $\varphi_{\text{leak}}$ given in (2), which we repeat below for ease of reference:

$$\varphi_{\text{leak}} \stackrel{\text{def}}{=} \exists \boldsymbol{x}.\langle x = \star \rangle \min X.(\langle x = \star \rangle \mathsf{tt} \vee \langle x \neq \star \rangle X).$$

The application of the monitor-synthesis function in Figure 5 to the above formula yields the monitor

$$m_{\text{leak}} \stackrel{\text{def}}{=} \mathsf{m}(\varphi_{\text{leak}}) = \mathsf{guess}\ x.(x = \star).\mathsf{rec}\, x.\, ((x = \star).\mathsf{yes} \oplus (x \neq \star).X).$$

Consider an erroneous execution $110^\omega$ exhibited by the server. The monitor $m_{\text{leak}}$ starts in configuration

$$\mathsf{guess}\ x.(x = \star).\mathsf{rec}\, x.\, ((x = \star).\mathsf{yes} \oplus (x \neq \star).X), \varnothing.$$

In its first step, the monitor $m_{\text{leak}}$ internally selects a concrete value $d \in \mathbb{D}$ for $x$. Note that such a value is selected over a possibly infinite domain, reminiscent of the countable nondeterminism studied in the classic paper [23]. Assume that the monitor chooses the value 0 for $x$. In the next step, the system emits 1 and the monitor checks for the guard $(x = \star)$, which does not hold. Thus, the monitor transitions to the configuration $\mathsf{end}, x \mapsto 0$, where it stays forever. Therefore, that monitor run leads to an inconclusive verdict.

Assume instead that the monitor picks $x = 1$ in its first step. Then, as our readers can check, the execution of the monitor continues and, after observing the prefix 11 of the trace $110^\omega$, the monitor raises a $\mathsf{yes}$ verdict. Thus, the trace is accepted by the monitor, indicating that the system repeats its first action and thus leaks its first data value.

**Theorem 3.3** (Theorem 17 in [8])**.** *The monitor $\mathsf{m}(\varphi)$ is sound and satisfaction-complete for each $\varphi \in \mathrm{cHML}^d$.*

*Remark.* By duality, a similar result holds for formulae in $\mathrm{sHML}^d$, for which one can construct sound and violation-complete monitors.

The monitor model we have used to prove Theorem 3.3 is equivalent to a model of register automata. Register automata were introduced in [59] as *finite-memory automata*. They consist of a finite-state automaton equipped with a finite set of *registers* that can store values from an infinite domain (here, $\mathbb{D}$). A register automaton can compare the value it reads with the content of its registers and move accordingly. We eschew the formal definition of the model that we use to establish the following result and limit ourselves to remarking that it is equivalent to that of [32, Section 1.3], omitting labels, which play no role in our developments. We refer the interested reader to [9, Appendix A.13] for the formal definitions and technical details.

**Theorem 3.4.** *Let $L \subseteq \mathbb{D}^*$ be an extension-closed language. There exists an alternating register automaton with existential guessing that recognises L if and only if there exists a monitor that accepts exactly the traces in L.*

This result echoes the equivalence between the alternation-free $\mu$-calculus and register tree automata presented in Theorems 3 and 7 in [58].

The correspondence also holds between register automata with no universal (respectively, existential) states and monitors with no occurrences of $\otimes$ (respectively, $\oplus$). Moreover, if one defines $\mathsf{match}(r, b) \stackrel{\text{def}}{=} \mathsf{guess}\ r.(b \wedge r = \star)$, all the above correspondences hold for register automata without guessing and monitors whose $\mathsf{guess}\ r$ construct is replaced with the $\mathsf{match}(r, b)$ one.

Since all those classes of register automata are inequivalent [32, Section 1.5], we know that all those variants of monitors correspond to different classes of properties. Thus, in $\mathrm{cHML}^d$ and $\mathrm{sHML}^d$, removing conjunctions or disjunctions reduces expressiveness, and the same holds when replacing existential quantification with a $\mathsf{match}(r, b)$ construct (as defined in [18]). This also shows that deterministic monitors (defined as the counterpart of deterministic register automata) are strictly less expressive than non-deterministic or alternating ones, which invalidates [18, Theorem 18].

A natural question to ask at this point is whether cHML$^d$ suffices to express all the formulae that have sound and satisfaction complete monitors. We now show that, in contrast to the finite alphabet case [13, Proposition 4.18], that fragment is not maximally expressive: there are properties that admit sound and satisfaction-complete monitors that cannot be expressed in cHML$^d$. Indeed, some formulae containing universal quantifiers *are* monitorable. As an example, consider the property that states that the input is divided into blocks separated by dollar and sharp symbols, and that all data values that appear in the second block appear in the first block, which is formalised as follows:

$$L_{\forall\#\exists\$} = \left\{ d_1 \dots d_k \$ e_1 \dots e_l \# \dots \;\middle|\; \forall 1 \leq j \leq l, \exists 1 \leq i \leq k, d_i = e_j \right\}, \tag{6}$$

That property admits a sound and satisfaction-complete monitor: the monitor reads the first two blocks by waiting to see the \$ and then the #; if this never happens, it means that the input violates the property. Otherwise, the monitor can check that all data values in the second block appear in the first one by processing them one by one, going back and forth.

The property $L_{\forall\#\exists\$}$ cannot be expressed in cHML$^d$—see [8, Proposition 21]. Intuitively, the length of the second block is unbounded, and its elements have to be compared with elements that appear *before* in the input, so they cannot be manipulated only using existential quantifiers, even with fixed points. Thus, cHML$^d$ is not a maximally expressive fragment whose formulae are monitorable for satisfaction using 'computable monitors'.

However, the property $L_{\forall\#\exists\$}$ *can* be expressed in recHML$^d$ thus:

$$\begin{aligned}
\varphi_{\forall\#\exists\$} &= \exists\, \boldsymbol{x}.\gamma(x) \wedge \forall\, \boldsymbol{x}.\,(\gamma(x) \vee \psi(x))\,, \text{ where:} \\
\gamma(x) &= \mathsf{min}\, X.\,(\langle \star \neq \$ \rangle X \vee \langle \star = \$ \rangle \mathsf{min}\, Y.\,(\langle \star = \# \rangle \mathsf{tt} \vee \langle \star \neq x \rangle Y)) \\
\psi(x) &= \mathsf{min}\, Z.\,(\langle \star = x \rangle \mathsf{tt} \vee \langle \star \neq \$ \rangle Z).
\end{aligned}$$

The formula $\gamma(x)$ is called the *guard*, and sets a monitorable bound on the maximal position where a candidate $x$ violating the formula $\psi$ can be found. This way, once the bound is found, the monitor knows that the subsequent data values that appear need not be checked. Here, it expresses that the trace starts with two blocks—ended by \$ and #, respectively—and that $x$ does not appear in the second block: first, look for a \$; once it is found, look for a #, and if in the meantime $x$ is encountered, the formula cannot recurse and therefore rejects.

The formula $\psi(x)$ is the universally quantified property, and since we are looking for satisfactions, its universal quantification has to be limited to finitely many values to ensure that it has a finite witness, hence the disjunction with the guard. Here, it expresses that $x$ appears in the first block. Summing up, the conjunct $\exists\, \boldsymbol{x}.\gamma(x)$ ensures that a trace has the form $w_1\$w_2\#u$ for some $w_1, w_2 \in \mathbb{D}^*$

and $u \in \textsc{Trc}^d$, while the conjunct $\forall\, \boldsymbol{x}.\, (\gamma(x) \vee \psi(x))$ yields that every $d \in \mathbb{D}$ occurring in $w_2$ also occurs in $w_1$.

Based on a substantial generalisation of the pattern highlighted by the guarded formula $\varphi_{\forall\#\exists\$}$, in [8, Theorem 26] we offer a characterisation of the collection of formulae in $\textsc{recHML}^d$ without greatest fixed points that can be monitored in a sound and satisfaction-complete fashion by computable monitors. The definition of that fragment and the proofs of the technical results for it are too intricate to be discussed in this survey article; they can be found in [8, Section 4.1] and [9]. It turns out that the fragment we identify does not express all the formulae in $\textsc{recHML}^d$ that have sound and satisfaction-complete monitors. However, in a precise sense, this is the best we can do: there is no maximally monitorable fragment of $\textsc{recHML}^d$ that is effective (see [8, Corollary 29]).

# 4 Exhibit C: Centralised and decentralised monitoring of hyperproperties

The last leg of our journey brings us to the very active study of runtime monitoring for hyperproperties—see, for instance, [2, 21, 33, 34, 40, 41, 48, 53]. *Hyperproperties* were introduced in [44] as sets of *hypertraces*, which are sets of traces that may be seen as describing different system executions or the contributions of different sequential processes to a system execution. Since their introduction, hyperproperties have become a fundamental, trace-based formalism for expressing security and privacy properties, verified using static and dynamic techniques [21,31,33,34,40,48] implemented in a variety of tools [30,47]. Several extensions of temporal logics have been proposed in the literature to describe hyperproperties. Examples of such logics include HyperLTL, HyperCTL$^*$ [43], Hyper$^2$LTL [31], and those based on variations on the $\mu$-calculus [2, 53].

In our work [3, 4], we have used a view of hypertraces that differs from the classic, set-based one from [44]. As in Section 2, we assume a finite set of actions $\textsc{Act}$ that contains at least two actions and write $\textsc{Trc}$ for the collection of infinite traces over $\textsc{Act}$. Given a finite and non-empty set $\mathcal{L}$ of *locations*, ranged over by $\ell$, a *hypertrace* $T$ over $\mathcal{L}$ is a function from $\mathcal{L}$ to $\textsc{Trc}$. The set of hypertraces on $\mathcal{L}$ is denoted by $\textsc{HTrc}_{\mathcal{L}}$.

Intuitively, a hypertrace describes an execution of a (distributed) system with $|\mathcal{L}|$ users; every user is located at a unique location chosen from $\mathcal{L}$ and each user is mapped to the trace they perform.

For $t, t' \in \textsc{Trc}$, we write $t \xrightarrow{a} t'$ whenever $t = at'$. We call a function $A : \mathcal{L} \to \textsc{Act}$ a hyperaction. For $T, T' \in \textsc{HTrc}_{\mathcal{L}}$, we write $T \xrightarrow{A} T'$ whenever $T(\ell) \xrightarrow{A(\ell)} T'(\ell)$,

for every $\ell \in \mathcal{L}$. Notice that, for each $T$, there is a *unique* pair $A$ and $T'$ such that $T \xrightarrow{A} T'$: more precisely, for every $\ell \in \mathcal{L}$, we have that $A(\ell) = a$ and $T'(\ell) = t'$, whenever $T(\ell) = at'$. We denote the $A$ and $T'$ just defined by $\mathrm{hd}(T)$ and $\mathrm{tl}(T)$, respectively.

The notation we just presented reflects the classic synchronous view of hypertraces and hyperproperties, according to which, at every step, a system proceeds by performing an action in Act at each of its locations. In this setting, a *finite* hypertrace $S$ is a function from $\mathcal{L}$ to $\mathrm{Act}^n$, for some $n \geq 0$. Each finite hypertrace $S$ results from the pointwise concatenation of some hyperactions $A_1, \ldots, A_n : \mathcal{L} \to \mathrm{Act}$, for some $n \geq 0$, thus:

$$S(\ell) = A_1 \cdots A_n(\ell) \quad = \quad A_1(\ell) \cdots A_n(\ell).$$

The concatenation of a finite hypertrace and a (finite) hypertrace is defined analogously in pointwise fashion. We say that a finite hypertrace $S$ is a prefix of a hypertrace $T$ if $S(\ell)$ is a prefix of $T(\ell)$ for each $\ell \in \mathcal{L}$—that is, there is some hypertrace $U$ such that $T(\ell) = S(\ell)U(\ell)$, for each $\ell \in \mathcal{L}$. A set of finite hypertraces is extension closed if whenever it contains some finite hypertrace $S$, then it also contains $SW$ for each finite hypertrace $W$.

## 4.1 Syntax and semantics of Hyper-recHML

We adopt a variation on recHML, which we call Hyper-recHML, as the logic to specify *hyperproperties*. We assume two disjoint and countably infinite sets $\Pi$ and $V$ of *location variables* and *logical variables*, ranged over by $\pi$ and $X$, respectively. Formulae of Hyper-recHML are constructed as follows:

$$\begin{aligned} \varphi \quad ::= \quad & \mathsf{tt} \mid \mathsf{ff} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \min X.\varphi \mid \max X.\varphi \mid X \mid \\ & \exists \pi.\varphi \mid \forall \pi.\varphi \mid \pi = \pi \mid \pi \neq \pi \mid [a_\pi]\varphi \mid \langle a_\pi \rangle \varphi. \end{aligned}$$

The new constructs with respect to those used in recHML are existential and universal quantification over locations, equality and inequality tests on location variables, and location-variable-indexed versions of the usual Hennessy-Milner modalities where $[a_\pi]$ stands for 'necessarily after $a$ at the location bound to $\pi$', and $\langle a_\pi \rangle$ denotes 'possibly after $a$ at the location bound to $\pi$'. Using these constructs, one can express properties such as 'the trace observed at each location starts with an $a$' (as $\forall \pi.\langle a_\pi \rangle \mathsf{tt}$) and 'there are two different locations whose trace starts with an $a$' (as $\exists \pi_1.\exists \pi_2.\langle a_{\pi_1} \rangle \mathsf{tt} \wedge \langle a_{\pi_2} \rangle \mathsf{tt} \wedge \pi_1 \neq \pi_2$).

The usual notions of closed and guarded formulae apply as in the previous sections and we tacitly restrict ourselves to those formulae. Moreover, we consider

$$[\![\mathsf{tt}]\!]^{\rho}_{\sigma} = \mathrm{HTRC}_{\mathcal{L}} \qquad [\![\mathsf{ff}]\!]^{\rho}_{\sigma} = \emptyset \qquad [\![X]\!]^{\rho}_{\sigma} = \rho(X)$$

$$[\![\varphi \wedge \varphi']\!]^{\rho}_{\sigma} = [\![\varphi]\!]^{\rho}_{\sigma} \cap [\![\varphi']\!]^{\rho}_{\sigma}$$

$$[\![\varphi \vee \varphi']\!]^{\rho}_{\sigma} = [\![\varphi]\!]^{\rho}_{\sigma} \cup [\![\varphi']\!]^{\rho}_{\sigma}$$

$$[\![\mathsf{max}\, X.\psi]\!]^{\rho}_{\sigma} = \bigcup \{\mathcal{S} \mid \mathcal{S} \subseteq \mathrm{HTRC}_{\mathcal{L}} \text{ and } \mathcal{S} \subseteq [\![\psi]\!]^{\rho[x \mapsto \mathcal{S}]}_{\sigma}\}$$

$$[\![\mathsf{min}\, X.\psi]\!]^{\rho}_{\sigma} = \bigcap \{\mathcal{S} \mid \mathcal{S} \subseteq \mathrm{HTRC}_{\mathcal{L}} \text{ and } \mathcal{S} \supseteq [\![\psi]\!]^{\rho[x \mapsto \mathcal{S}]}_{\sigma}\}$$

$$[\![\exists \pi.\varphi]\!]^{\rho}_{\sigma} = \bigcup_{\ell \in \mathcal{L}} [\![\varphi]\!]^{\rho}_{\sigma[\pi \mapsto \ell]}$$

$$[\![\forall \pi.\varphi]\!]^{\rho}_{\sigma} = \bigcap_{\ell \in \mathcal{L}} [\![\varphi]\!]^{\rho}_{\sigma[\pi \mapsto \ell]}$$

$$[\![\pi = \pi']\!]^{\rho}_{\sigma} = \begin{cases} \mathrm{HTRC}_{\mathcal{L}} & \text{if } \sigma(\pi) = \sigma(\pi') \\ \emptyset & \text{otherwise} \end{cases}$$

$$[\![\pi \neq \pi']\!]^{\rho}_{\sigma} = \begin{cases} \mathrm{HTRC}_{\mathcal{L}} & \text{if } \sigma(\pi) \neq \sigma(\pi') \\ \emptyset & \text{otherwise} \end{cases}$$

$$[\![[a_\pi]\varphi]\!]^{\rho}_{\sigma} = \{T \mid \mathrm{hd}(T)(\sigma(\pi)) = a \text{ implies } \mathrm{tl}(T) \in [\![\varphi]\!]^{\rho}_{\sigma}\}$$

$$[\![\langle a_\pi \rangle \varphi]\!]^{\rho}_{\sigma} = \{T \mid \mathrm{hd}(T)(\sigma(\pi)) = a \ \wedge \ \mathrm{tl}(T) \in [\![\varphi]\!]^{\rho}_{\sigma})\}$$

Figure 6: The semantics of Hyper-recHML.

formulae whose bound location variables are all pairwise distinct (and different from the free variables, when we need to define notions over open fomulae).

The semantics of formulae $\varphi$ in Hyper-recHML is defined over $\mathrm{HT}_{\mathrm{RC}_{\mathcal{L}}}$ through the function $[\![-]\!]^{\rho}_{\sigma}$, as shown in Figure 6, by exploiting two partial functions: $\rho\colon V \rightharpoonup 2^{\mathrm{HT}_{\mathrm{RC}_{\mathcal{L}}}}$, which assigns a set of hypertraces over $\mathcal{L}$ to all free logical variables occurring in $\varphi$, and $\sigma\colon \Pi \rightharpoonup \mathcal{L}$, which assigns a location to all free location variables in $\varphi$. In what follows, we tacitly assume that the free logical and location variables in a formula $\varphi$ are always included in the domains of $\rho$ and $\sigma$, respectively. The function $\rho$ gives the semantics of logical variables and $\sigma$ keeps track of the location associated with each location variable. The function $\sigma$ is extended every time a new variable $\pi$ is introduced (through $\exists\pi$ or $\forall\pi$) to check whether the body of the quantification holds for some or for all locations.

All the constructs have the expected semantics. In particular, a formula $\langle a_\pi\rangle\varphi$ holds true at hypertrace $T$ if the trace in $T$ at the location bound to $\pi$ starts with an $a$ and $\mathrm{tl}(T)$ satisfies $\varphi$; by contrast, a formula $[a_\pi]\varphi$ can also hold true if the trace in $T$ at the location associated to $\pi$ does not start with an $a$.

Whenever $\varphi$ is *closed* (i.e., without any free variable), the semantics is given by $[\![\varphi]\!]^{\emptyset}_{\emptyset}$, where $\emptyset$ denotes the partial function with empty domain. Notationally, we shall simply write $[\![\varphi]\!]$ instead of $[\![\varphi]\!]^{\emptyset}_{\emptyset}$. We say that $T$ satisfies the closed formula $\varphi$ if $T \in [\![\varphi]\!]$.

**Example 4.1.** For example, consider the set of actions $\{a, b\}$; then, the hyperproperty

$$\varphi_a = \forall\pi.\max X.(\langle b_\pi\rangle X \ \vee \ \exists\pi'.(\pi' \neq \pi \ \wedge \ \langle a_{\pi'}\rangle X)) \tag{7}$$

is a consensus-type property stating that, at every position of every trace, whenever there is an $a$ there is another trace that also has $a$. Using the definition of the semantics of the logic, it is not hard to see that the hypertrace $T_1$ over the set of locations $\{\ell_1, \ell_2, \ell_3\}$ that maps $\ell_1$ to $a^\omega$, $\ell_2$ to $ba^\omega$ and $\ell_3$ to $(ba)^\omega$ does not satisfy the property $\varphi_a$: what breaks the property is the first position. On the other hand, the hypertrace $T_2$ that maps $\ell_1$ to $a^\omega$, $\ell_2$ to $(ab)^\omega$ and $\ell_3$ to $(ba)^\omega$ does satisfy $\varphi_a$ because at each position there are two traces that exhibit an $a$. The hypertrace $T_3$ that maps each location to $b^\omega$ also satisfies $\varphi_a$ since no $a$ is ever observed.

As argued in [3, Section 2.2], Hyper-recHML is more expressive than HyperLTL and HyperCTL$^*$. This additional expressiveness is also present in the fragments of that logic for which we can synthesise sound and violation-complete monitors.

## 4.2 What are the monitorable Hyper-recHML formulae?

Following our presentation in the classic setting (Section 2) and in the one with data (Section 3), we can abstractly characterise an irrevocable monitor $m$ for some

$$v \xrightarrow{A} v \qquad \frac{A(\ell) = a}{a_\ell.m \xrightarrow{A} m} \qquad \frac{A(\ell) \neq a}{a_\ell.m \xrightarrow{A} \mathsf{end}} \qquad \frac{m[\mathsf{rec}\, x.m/x] \xrightarrow{A} m'}{\mathsf{rec}\, x.m \xrightarrow{A} m'} \qquad \frac{m \xrightarrow{A} m'}{m + n \xrightarrow{A} m'}$$

$$\frac{n \xrightarrow{A} n'}{m + n \xrightarrow{A} n'} \qquad \frac{m \xrightarrow{A} m' \qquad n \xrightarrow{A} n'}{m \odot n \xrightarrow{A} m' \odot n'}$$

Figure 7: The operational semantics for centralised monitors, where $\odot \in \{\otimes, \oplus\}$.

hyperproperty by means of two disjoint sets of finite hypertraces $A(m)$ and $R(m)$ that are extension closed. Using those sets, we can define when a monitor $m$ is sound and violation- or satisfaction-complete for some property $\varphi \in$ Hyper-recHML by mimicking Definition 2.2. In what follows, we focus solely on monitors that detect violations and therefore we identify a monitor with the set $R(m)$ of finite hypertraces that it rejects.

**Example 4.2.** Assume that $\mathcal{L} = \{\ell_1, \ell_2\}$ and that $\text{Act} = \{a, b\}$. Let $m$ be a monitor such that $R(m)$ consists of all the finite hypertraces $S$ such that $S(\ell_1)$ or $S(\ell_2)$ starts with a $b$. Then $m$ is sound and violation-complete for the formula $\forall\pi.\langle a_\pi\rangle\mathsf{tt}$.

Following the lead of the work we discussed in the previous sections, we now present a monitoring system that is based on centralised, omniscient monitors that can monitor for a collection of hyperproperties expressed in a fragment of Hyper-recHML.

The set of *centralised monitors* CMon, ranged over by $m, n$, is given by the following grammar:

$$m ::= \mathsf{yes} \mid \mathsf{no} \mid \mathsf{end} \mid a_\ell.m \mid m + m \mid m \oplus m \mid m \otimes m \mid \mathsf{rec}\, x.m \mid x.$$

Our readers are already familiar with nearly all the constructs for monitors in CMon. The only new ingredient is the prefixed monitor $a_\ell.m$, which checks if location $\ell$ is currently performing an $a$ and, in that case, continues as $m$.

The operational semantics of centralised monitors is given in Figure 7 and follows the intuitive explanations we gave earlier for the different operators. In particular, a monitor that waits for an action at some location (as prescribed by $a_\ell$) can either see that action at $\ell$ (as stated by an hyperaction $A$) and proceed in its observation, or it does not observe that action and stops its monitoring activity, by reporting $\mathsf{end}$. Recursive monitors should be unfolded to evolve. A

$$v \Rrightarrow v \qquad \frac{m \Rrightarrow \mathsf{end} \quad n \Rrightarrow \mathsf{end}}{m \odot n \Rrightarrow \mathsf{end}} \qquad \frac{m \Rrightarrow \mathsf{yes}}{m \oplus n \Rrightarrow \mathsf{yes}} \qquad \frac{m \Rrightarrow \mathsf{no}}{m \otimes n \Rrightarrow \mathsf{no}}$$

$$\frac{m \Rrightarrow v}{m + n \Rrightarrow v} \qquad \frac{m \Rrightarrow \mathsf{no} \quad n \Rrightarrow v}{m \oplus n \Rrightarrow v} \qquad \frac{m \Rrightarrow \mathsf{yes} \quad n \Rrightarrow v}{m \otimes n \Rrightarrow v} \qquad \frac{m[\mathsf{rec}\, x.m/x] \Rrightarrow v}{\mathsf{rec}\, x.m \Rrightarrow v}$$

Figure 8: Verdict evaluation for centralised monitors (up to commutativity of $+$, $\otimes$, and $\oplus$).

non-deterministic choice $m + n$ can initially behave like $m$ and discard $n$ in doing so or vice versa. Finally, once issued, a verdict never changes; possibly different verdicts obtained in different parallel branches of the monitor can be composed conjunctively or disjunctively. This is represented by the judgement $\Rrightarrow$ defined by the rules in Figure 8. Amongst other things, those rules state that $\mathsf{yes}$ and $\mathsf{no}$ are the absorbing elements for $\oplus$ and $\otimes$, respectively, and the neutral elements for $\otimes$ and $\oplus$, respectively, that verdict evaluation of a non-deterministic monitor is non-deterministic as well, and that recursive monitors must be unfolded before they can be properly evaluated.

As our attentive readers might have noticed already, the aforementioned syntax of centralised monitors is very general and allows one to write monitors that can omit inconsistent verdicts. For example, the monitor $\mathsf{yes} + \mathsf{no}$ can immediately report both a $\mathsf{yes}$ and a $\mathsf{no}$ verdict via the transitions $\mathsf{yes} + \mathsf{no} \Rrightarrow \mathsf{yes}$ and $\mathsf{yes} + \mathsf{no} \Rrightarrow \mathsf{no}$, respectively. In what follows, we tacitly restrict ourselves to consistent monitors. Our monitor-synthesis procedures only generate consistent monitors from formulae in Hyper-recHML.

The transition relation $\rightarrow$ can be extended to finite hypertraces by stipulating that $m \xrightarrow{A_1 \cdots A_n} m'$ holds if and only if the pair of monitors $(m, m')$ is contained in the composition of the relations $\xrightarrow{A_1}, \ldots, \xrightarrow{A_n}$. Using the operational semantics of monitors given in Figures 7-8, we can now associate sets of finite hypertraces $A(m)$ and $R(m)$ to each centralised monitor $m$ thus:

$$\begin{aligned} A(m) &= \{A_1 \cdots A_n \mid m \xrightarrow{A_1 \cdots A_n} m' \Rrightarrow \mathsf{yes}, \text{ for some } m'\} \text{ and} \\ R(m) &= \{A_1 \cdots A_n \mid m \xrightarrow{A_1 \cdots A_n} m' \Rrightarrow \mathsf{no}, \text{ for some } m'\}. \end{aligned}$$

**Example 4.3.** Let us revisit the setting of Example 4.2, where we assumed that

$$
\begin{aligned}
\mathsf{cm}(\mathsf{tt})_\sigma &= \mathsf{yes} \\
\mathsf{cm}(\mathsf{ff})_\sigma &= \mathsf{no} \\
\mathsf{cm}(X)_\sigma &= x \\
\mathsf{cm}(\max X.\varphi)_\sigma &= \mathsf{rec}\, x.\mathsf{cm}(\varphi)_\sigma \\
\mathsf{cm}(\varphi \wedge \varphi')_\sigma &= \mathsf{cm}(\varphi)_\sigma \otimes \mathsf{cm}(\varphi')_\sigma \\
\mathsf{cm}(\varphi \vee \varphi')_\sigma &= \mathsf{cm}(\varphi)_\sigma \oplus \mathsf{cm}(\varphi')_\sigma \\
\mathsf{cm}(\forall \pi.\varphi)_\sigma &= \bigotimes_{\ell \in \mathcal{L}} \mathsf{cm}(\varphi)_{\sigma[\pi \mapsto \ell]} \\
\mathsf{cm}(\exists \pi.\varphi)_\sigma &= \bigoplus_{\ell \in \mathcal{L}} \mathsf{cm}(\varphi)_{\sigma[\pi \mapsto \ell]} \\
\mathsf{cm}(\pi = \pi')_\sigma &= \begin{cases} \mathsf{yes} \text{ if } \sigma(\pi) = \sigma(\pi') \\ \mathsf{no} \text{ otherwise} \end{cases} \\
\mathsf{cm}(\pi \neq \pi')_\sigma &= \begin{cases} \mathsf{yes} \text{ if } \sigma(\pi) \neq \sigma(\pi') \\ \mathsf{no} \text{ otherwise} \end{cases} \\
\mathsf{cm}([a_\pi]\varphi)_\sigma &= a_{\sigma(\pi)}.\mathsf{cm}(\varphi)_\sigma + \sum_{b \neq a} b_{\sigma(\pi)}.\mathsf{yes} \\
\mathsf{cm}(\langle a_\pi \rangle \varphi)_\sigma &= a_{\sigma(\pi)}.\mathsf{cm}(\varphi)_\sigma + \sum_{b \neq a} b_{\sigma(\pi)}.\mathsf{no}
\end{aligned}
$$

Figure 9: Centralised monitor synthesis.

$\mathcal{L} = \{\ell_1, \ell_2\}$ and that $\text{Act} = \{a, b\}$. Consider the monitor

$$m = (a_{\ell_1}.\mathsf{yes} + b_{\ell_1}.\mathsf{no}) \otimes (a_{\ell_2}.\mathsf{yes} + b_{\ell_2}.\mathsf{no}).$$

It is not hard to see that $R(m)$ consists of all the finite hypertraces $S$ such that $S(\ell_1)$ or $S(\ell_2)$ starts with a $b$. Therefore, as we saw in Example 4.2, $m$ is sound and violation-complete for the formula $\forall\pi.\langle a_\pi\rangle\mathsf{tt}$.

We now show how to automatically construct sound and violation-complete monitors from formulae in Hyper-recHML without least fixed-point operators. The definition of the synthetised monitor is given by induction on a formula $\varphi$ and is parametrised by a partial function $\sigma$, assigning a location to all the free location variables of $\varphi$; when $\varphi$ is closed, we consider $\mathsf{cm}(\varphi)_\emptyset$. The formal definition is given in Figure 9. A monitor synthetized from a greatest-fixed-point formula is itself recursive and intuitively checks whether some unfolding of the formula is violated. The monitor for $\varphi \wedge \varphi'$ (respectively, $\varphi \vee \varphi'$) is obtained as the 'parallel and' (respectively, the 'parallel or') of the monitors synthesised from $\varphi$ and $\varphi'$. Universal and existential quantifiers are treated as conjunctions and disjunctions, respectively, and use the function $\sigma$ to assign values to the newly introduced location variable. Finally, the monitors for formulae of the form $[a_\pi]\varphi$ and $\langle a_\pi\rangle\varphi$ look for an occurrence of action $a$ at the location bound to $\pi$ in $\sigma$; if such action is observed at that location, then the monitor proceeds by checking the rest of the formula, otherwise the monitor for $\langle a_\pi\rangle\varphi$ returns $\mathsf{no}$ whereas the monitor for $[a_\pi]\varphi$ returns $\mathsf{yes}$. Notice that we are using the choice operator '+' here to consider only one of the possible observations (corresponding to the action occurring at the location $\sigma(\pi)$) and discarding all the other ones.

The following result states the correctness of the monitor-synthesis function given in Figure 9—see Theorems 3.5 and 3.8 in [4] for its proof.

**Theorem 4.1.** *Let $\varphi$ be a closed formula in Hyper-recHML that does not contain occurrences of least-fixed-point operators and let $T \in \text{HTrc}_{\mathcal{L}}$. Then $\mathsf{cm}(\varphi)_\emptyset$ rejects $T$ iff $T \notin [\![\, \varphi \,]\!]$.*

The above results tells us how to generate sound and violation-complete centralised monitors for a fragment of our touchstone logic Hyper-recHML. However, when verifying a distributed system, having a central authority that performs any type of runtime verification is a strong assumption, as it reduces the appeal of distribution, creates single points of failure during verification and can pose problems in storing all the traces locally, especially in light of the wide availability of multi-core systems. Thus, in [3, 4, Section 4], we studied to what extent hyperproperties can be monitored by *decentralised* monitors; these avoid high contentions (leading to vastly improved scalability [17]) and also offer better privacy guarantees (whenever they are stationed locally at the nodes where the respective traces are generated).

Intuitively, in a decentralised-monitoring setting, we associate monitors defined as in Figure 3, with locations. A monitor $m$ located at $\ell$, denoted by $[m]_\ell$, observes only actions required to happen at $\ell$, thus allowing the processing of events to occur locally. As in [2], located monitors of the form $[m]_\ell$ are nodes in a structure that resembles a Boolean circuit, which allows them to combine their individual verdicts into a global one. However, unlike in the aforementioned reference, in [3,4] we introduce the possibility for monitors to communicate with one another and coordinate their behaviour when checking for properties involving several quantifiers, and therefore the interplay between various traces in a hypertrace. To this end, we extend the syntax of local monitors from Figure 3 with a new prefixing operator of the form $c.m$, where $c$ is a *communication action*. The form of monitor communication we studied in [3,4] was *multicast* and involved communication actions of the form $(!G, \gamma)$, for sending $\gamma$ to all locations in the group $G$, and $(?G, \gamma)$, for receiving $\gamma$ from any monitor from the set of locations $G$.

The details of the operational semantics of decentralised monitors are rather involved, and so is the procedure for synthesising decentralised monitors from formulae in Hyper-recHML without least fixed points that are Boolean combinations of formulae in prenex form. (See [3, Section 4.2] for the precise definition of that fragment of Hyper-recHML.) We therefore refer our readers to the aforementioned reference for details. Here we limit ourselves to mentioning that the proof of correctness in the decentralised case is considerably more technical than the corresponding proof of Theorem 4.1, due to the intricate communication semantics of decentralised monitors. To address the resulting technical challenges, we again rely on classic tools from concurrency theory to develop a proof strategy where we prove the correctness of the decentralised monitor synthesis procedure using the centralised one as a yardstick. More precisely, in [3, Definition 13] we identify six properties of a decentralised monitor synthesis that make it 'principled' and we show that, when a decentralised monitor synthesis is principled, the centralised and decentralised monitors synthesised from a formula are related by a suitable notion of weak bisimulation (see [3, Theorem 14]). Apart from supporting the definition of decentralised monitor synthesis procedures, this result allows us to reduce the correctness of our decentralised monitor synthesis to that of the centralised one, which can in turn drive the definition of further synthesis procedures in future work.

# 5 Concluding remarks

The goal of this article was to provide an introduction to some of our work on runtime monitoring, highlighting the role that classic concurrency theory has

played in our technical developments. It is not up to us to assess the value of what we have achieved thus far for the runtime monitoring community, but we hope that some of the readers of this article will be enticed to study the technical contributions in the papers mentioned in the bibliography in more detail.

Our research journey in the theoretical foundations of runtime monitoring is by no means over and much remains to be done. For example, our work presented in Sections 3–4 falls short of providing theoretical developments that are as general and complete as those in the classic setting studied in Section 2. For example, we still need to develop an elegant (process-algebraic) model of monitors that are sound and satisfaction-complete for the maximally-expressive, guarded fragment of RECHML$^d$ identified in [8]. Moreover, in the setting of Hyper-recHML, we are still missing maximality and expressiveness results. For instance, it is natural to wonder whether the fragment of that logic for which we synthesised sound and violation-complete monitors can express all the hyperproperties for which such monitors exist. Moreover, we do not know yet whether the decentralised monitors with multicast communication we studied in [3, 4] are less powerful than the centralised monitors. And does the expressive power of decentralised monitors depend on the form of communication they employ? We are currently working on these questions and on some others through the lens of concurrency theory, and hope to report on any answers we might obtain in future papers. To quote the title of an ERC project led by Thomas A. Henzinger on the topic of runtime monitoring, 'VAMOS!'[3]

**Acknowledgements** The work reported in this article has been partially supported by grants No. 163406-051, 184940-051 and 217987 of the Icelandic Research Fund. We are most grateful to that funding agency and the Reykjavik University Research Fund for supporting theoretical research in concurrency theory and runtime monitoring over many years.

We thank Marino Miculan and Nobuko Yoshida for their interest in our work and for commissioning this paper.

# References

[1] Yael Abarbanel, Ilan Beer, Leonid Gluhovsky, Sharon Keidar, and Yaron Wolfsthal. FoCs: Automatic generation of simulation checkers from formal specifications. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 538–542. Springer, 2000.

[3] See `https://ista.ac.at/en/vigilant-algorithmic-monitoring-of-software/`.

[2] Luca Aceto, Antonis Achilleos, Elli Anastasiadi, and Adrian Francalanza. Monitoring hyperproperties with circuits. In Mohammad Reza Mousavi and Anna Philippou, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 42nd IFIP WG 6.1 International Conference, FORTE 2022*, volume 13273 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2022.

[3] Luca Aceto, Antonis Achilleos, Elli Anastasiadi, Adrian Francalanza, Daniele Gorla, and Jana Wagemaker. Centralized vs decentralized monitors for hyperproperties. In Rupak Majumdar and Alexandra Silva, editors, *35th International Conference on Concurrency Theory, CONCUR 2024*, volume 311 of *LIPIcs*, pages 4:1–4:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[4] Luca Aceto, Antonis Achilleos, Elli Anastasiadi, Adrian Francalanza, Daniele Gorla, and Jana Wagemaker. Centralized vs decentralized monitors for hyperproperties. *ACM Trans. Comput. Log.*, 27(1):2:1–2:57, 2026.

[5] Luca Aceto, Antonis Achilleos, Elli Anastasiadi, Adrian Francalanza, and Anna Ingólfsdóttir. Complexity results for modal logic with recursion via translations and tableaux. *Log. Methods Comput. Sci.*, 20(3), 2024.

[6] Luca Aceto, Antonis Achilleos, Elli Anastasiadi, and Anna Ingólfsdóttir. Axiomatizing recursion-free, regular monitors. *J. Log. Algebraic Methods Program.*, 127:100778, 2022.

[7] Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Léo Exibard, Adrian Francalanza, and Anna Ingólfsdóttir. A monitoring tool for linear-time $\mu$HML. *Sci. Comput. Program.*, 232:103031, 2024.

[8] Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Léo Exibard, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. Monitorability for the modal mu-calculus over systems with data: From practice to theory. In Patricia Bouyer and Jaco van de Pol, editors, *36th International Conference on Concurrency Theory, CONCUR 2025*, volume 348 of *LIPIcs*, pages 4:1–4:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. Full version available at `https://doi.org/10.48550/arXiv.2506.06172`.

[9] Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Léo Exibard, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. Monitorability for the modal mu-calculus over systems with data: From practice to theory. *CoRR*, abs/2506.06172, 2025.

[10] Luca Aceto, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfsdóttir. Monitoring for silent actions. In Satya V. Lokam and R. Ramanujam, editors, *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017*, volume 93 of *LIPIcs*, pages 7:1–7:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[11] Luca Aceto, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfsdóttir. A framework for parameterized monitorability. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures - 21st International*

*Conference, FOSSACS 2018*, volume 10803 of *Lecture Notes in Computer Science*, pages 203–220. Springer, 2018.

[12] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Sævar Örn Kjartansson. Determinizing monitors for HML with recursion. *J. Log. Algebraic Methods Program.*, 111:100515, 2020.

[13] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karolina Lehtinen. Adventures in monitorability: From branching to linear time and back again. *Proc. ACM Program. Lang.*, 3(POPL):52:1–52:29, 2019. Full version available at `https://arxiv.org/abs/1902.00435`.

[14] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karolina Lehtinen. The cost of monitoring alone. In Ezio Bartocci, Rance Cleaveland, Radu Grosu, and Oleg Sokolsky, editors, *From Reactive Systems to Cyber-Physical Systems - Essays Dedicated to Scott A. Smolka on the Occasion of His 65th Birthday*, volume 11500 of *Lecture Notes in Computer Science*, pages 259–275. Springer, 2019.

[15] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karolina Lehtinen. The best a monitor can do. In Christel Baier and Jean Goubault-Larrecq, editors, *29th EACSL Annual Conference on Computer Science Logic, CSL 2021*, volume 183 of *LIPIcs*, pages 7:1–7:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[16] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karolina Lehtinen. An operational guide to monitorability with applications to regular properties. *Softw. Syst. Model.*, 20(2):335–361, 2021.

[17] Luca Aceto, Duncan Paul Attard, Adrian Francalanza, and Anna Ingólfsdóttir. Runtime instrumentation for reactive components. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming, ECOOP 2024*, volume 313 of *LIPIcs*, pages 2:1–2:33. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[18] Luca Aceto, Ian Cassar, Adrian Francalanza, and Anna Ingólfsdóttir. On runtime enforcement via suppressions. In Sven Schewe and Lijun Zhang, editors, *29th International Conference on Concurrency Theory, CONCUR 2018*, volume 118 of *LIPIcs*, pages 34:1–34:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[19] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge Univ. Press, 2007.

[20] Antonis Achilleos, Adrian Francalanza, and Jasmine Xuereb. If at first you don't succeed: Extended monitorability through multiple executions. In *40th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2025*, pages 636–650. IEEE, 2025.

[21] Shreya Agrawal and Borzoo Bonakdarpour. Runtime verification of k-safety hyperproperties in HyperLTL. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016*, pages 239–252. IEEE Computer Society, 2016.

[22] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, and Mattias Ulbrich, editors. *Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY*, volume 12345 of *Lecture Notes in Computer Science*. Springer, 2020.

[23] Krzysztof R. Apt and Gordon D. Plotkin. Countable nondeterminism and random assignment. *J. ACM*, 33(4):724–767, 1986.

[24] Duncan Paul Attard, Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. Better late than never or: Verifying asynchronous components at runtime. In Kirstin Peters and Tim A. C. Willemse, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 41st IFIP WG 6.1 International Conference, FORTE 2021*, volume 12719 of *Lecture Notes in Computer Science*, pages 207–225. Springer, 2021.

[25] Duncan Paul Attard, Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. A runtime monitoring tool for actor-based systems. In Simon Gay and Antonio Ravara, editors, *BETTY Book*, pages 49–76. River Publishers, 2017. Chapter 3. Available at `http://www.riverpublishers.com/downloadchapter.php?file=RP_9788793519817C3.pdf`.

[26] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[27] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004.

[28] Ezio Bartocci and Yliès Falcone, editors. *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*. Springer, 2018.

[29] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[30] Raven Beutner and Bernd Finkbeiner. Software verification of hyperproperties beyond k-safety. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022*, volume 13371 of *Lecture Notes in Computer Science*, pages 341–362. Springer, 2022.

[31] Raven Beutner, Bernd Finkbeiner, Hadar Frenkel, and Niklas Metzger. Monitoring second-order hyperproperties. In Mehdi Dastani, Jaime Simão Sichman, Natasha Alechina, and Virginia Dignum, editors, *Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2024*, pages 180–188. International Foundation for Autonomous Agents and Multiagent Systems / ACM, 2024.

[32] Mikołaj Bojańczyk. *Slightly Infinite Sets*. Online book draft, 2019. Available at `https://www.mimuw.edu.pl/~bojan/paper/atom-book`.

[33] Borzoo Bonakdarpour and Bernd Finkbeiner. Runtime verification for HyperLTL. In Yliès Falcone and César Sánchez, editors, *Runtime Verification - 16th International Conference, RV 2016*, volume 10012 of *Lecture Notes in Computer Science*, pages 41–45. Springer, 2016.

[34] Borzoo Bonakdarpour, César Sánchez, and Gerardo Schneider. Monitoring hyperproperties by combining static analysis and runtime verification. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018*, volume 11245 of *Lecture Notes in Computer Science*, pages 8–27. Springer, 2018.

[35] Florian Bruse, Oliver Friedmann, and Martin Lange. On guarded transformation in the modal $\mu$-calculus. *Log. J. IGPL*, 23(2):194–216, 2015.

[36] Filip Cano, Thomas A. Henzinger, and Konstantin Kueffner. Algorithmic fairness: A runtime perspective. In Bettina Könighofer and Hazem Torfah, editors, *Runtime Verification - 25th International Conference, RV 2025*, volume 16087 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2025.

[37] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. eAOP: an aspect oriented programming framework for Erlang. In Natalia Chechina and Scott Lystig Fritchie, editors, *Proceedings of the 16th ACM SIGPLAN International Workshop on Erlang*, pages 20–30. ACM, 2017.

[38] Ian Cassar, Adrian Francalanza, Duncan Paul Attard, Luca Aceto, and Anna Ingólfsdóttir. A generic instrumentation tool for Erlang. In Giles Reger and Klaus Havelund, editors, *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*, volume 3 of *Kalpa Publications in Computing*, pages 48–54. EasyChair, 2017.

[39] Ian Cassar, Adrian Francalanza, Duncan Paul Attard, Luca Aceto, and Anna Ingólfsdóttir. A suite of monitoring tools for Erlang. In Giles Reger and Klaus Havelund, editors, *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*, volume 3 of *Kalpa Publications in Computing*, pages 41–47. EasyChair, 2017.

[40] Marek Chalupa and Thomas A. Henzinger. Monitoring hyperproperties with prefix transducers. In Panagiotis Katsaros and Laura Nenzi, editors, *Runtime Verification - 23rd International Conference, RV 2023*, volume 14245 of *Lecture Notes in Computer Science*, pages 168–190. Springer, 2023.

[41] Marek Chalupa, Thomas A. Henzinger, and Ana Oliveira da Costa. Monitoring hypernode logic over infinite domains. In Bettina Könighofer and Hazem Torfah, editors, *Runtime Verification - 25th International Conference, RV 2025*, volume 16087 of *Lecture Notes in Computer Science*, pages 417–437. Springer, 2025.

[42] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[43] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust - Third International Conference, POST 2014*, volume 8414 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2014.

[44] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010.

[45] Ankush Desai, Tommaso Dreossi, and Sanjit A. Seshia. Combining model checking and runtime verification for safe robotics. In Shuvendu K. Lahiri and Giles Reger, editors, *Runtime Verification - 17th International Conference, RV 2017*, volume 10548 of *Lecture Notes in Computer Science*, pages 172–189. Springer, 2017.

[46] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier and MIT Press, 1990.

[47] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. RVHyper: A runtime verification tool for temporal hyperproperties. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018*, volume 10806 of *Lecture Notes in Computer Science*, pages 194–200. Springer, 2018.

[48] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. Monitoring hyperproperties. *Formal Methods Syst. Des.*, 54(3):336–363, 2019.

[49] Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. A foundation for runtime monitoring. In Shuvendu K. Lahiri and Giles Reger, editors, *Runtime Verification - 17th International Conference, RV 2017*, volume 10548 of *Lecture Notes in Computer Science*, pages 8–29. Springer, 2017.

[50] Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. On verifying Hennessy-Milner logic with recursion at runtime. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime Verification - 6th International Conference, RV 2015*, volume 9333 of *Lecture Notes in Computer Science*, pages 71–86. Springer, 2015.

[51] Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. Monitorability for the Hennessy-Milner logic with recursion. *Formal Methods Syst. Des.*, 51(1):87–116, 2017.

[52] Jan Friso Groote and Radu Mateescu. Verification of temporal properties of processes in a setting with data. In Armando Martin Haeberer, editor, *Algebraic Methodology and Software Technology, 7th International Conference, AMAST '98*, volume 1548 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 1998.

[53] Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. Automata and fixpoints for asynchronous hyperproperties. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.

[54] Klaus Havelund and Doron Peled. Runtime verification: From propositional to first-order temporal logic. In Christian Colombo and Martin Leucker, editors, *Runtime Verification - 18th International Conference, RV 2018*, volume 11237 of *Lecture Notes in Computer Science*, pages 90–112. Springer, 2018.

[55] Klaus Havelund and Grigore Rosu, editors. *Workshop on Runtime Verification, RV 2001, in connection with CAV 2001*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

[56] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.

[57] Thomas A. Henzinger, Konstantin Kueffner, and Emily Yu. Formal verification of neural certificates done dynamically. In Bettina Könighofer and Hazem Torfah, editors, *Runtime Verification - 25th International Conference, RV 2025*, volume 16087 of *Lecture Notes in Computer Science*, pages 54–72. Springer, 2025.

[58] Marcin Jurdzinski and Ranko Lazic. Alternation-free modal mu-calculus for data trees. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 131–140. IEEE Computer Society, 2007.

[59] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.

[60] Robert M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, 1976.

[61] Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[62] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *J. ACM*, 47(2):312–360, 2000.

[63] Kim Guldstrand Larsen. Proof systems for satisfiability in Hennessy-Milner logic with recursion. *Theor. Comput. Sci.*, 72(2&3):265–288, 1990.

[64] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[65] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebraic Methods Program.*, 78(5):293–303, 2009.

[66] Nicolas Markey and Philippe Schnoebelen. Model checking a path. In Roberto M. Amadio and Denis Lugiez, editors, *CONCUR 2003 - Concurrency Theory, 14th International Conference*, volume 2761 of *Lecture Notes in Computer Science*, pages 248–262. Springer, 2003.

[67] Nicolas Markey and Philippe Schnoebelen. Mu-calculus path checking. *Inf. Process. Lett.*, 97(6):225–230, 2006.

[68] Robin Milner. *Communication and Concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[69] Kshirasagar Naik and Priyadarshi Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. John Wiley & Sons, 2011.

[70] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435, 2004.

[71] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 37 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2013.

[72] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.*, 60-61:17–139, 2004.

[73] Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 573–586. Springer, 2006.

[74] César Sánchez, Gerardo Schneider, Wolfgang Ahrendt, Ezio Bartocci, Domenico Bianculli, Christian Colombo, Yliès Falcone, Adrian Francalanza, Srdan Krstic, João M. Lourenço, Dejan Nickovic, Gordon J. Pace, José Rufino, Julien Signoles, Dmitriy Traytel, and Alexander Weiss. A survey of challenges for runtime verification from advanced application domains (beyond software). *Formal Methods Syst. Des.*, 54(3):279–335, 2019.

[75] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[76] Sanjit A. Seshia, Dorsa Sadigh, and S. Shankar Sastry. Toward verified artificial intelligence. *Commun. ACM*, 65(7):46–55, 2022.

[77] Deian Tabakov, Kristin Y. Rozier, and Moshe Y. Vardi. Optimized temporal monitors for SystemC. *Formal Methods Syst. Des.*, 41(3):236–268, 2012.

[78] Rania Taleb, Sylvain Hallé, and Raphaël Khoury. Uncertainty in runtime verification: A survey. *Comput. Sci. Rev.*, 50:100594, 2023.

[79] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[80] Hazem Torfah, Sebastian Junges, Daniel J. Fremont, and Sanjit A. Seshia. Formal analysis of AI-based autonomy: From modeling to runtime assurance. In Lu Feng and Dana Fisman, editors, *Runtime Verification - 21st International Conference, RV 2021*, volume 12974 of *Lecture Notes in Computer Science*, pages 311–330. Springer, 2021.