Adventures in Monitorability

From Branching to Linear Time and Back Again

LUCA ACETO, Gran Sasso Science Institute, Italy and Reykjavik University, Iceland
ANTONIS ACHILLEOS, Reykjavik University, Iceland
ADRIAN FRANCALANZA, University of Malta, Malta
ANNA INGÓLFSDÓTTIR, Reykjavik University, Iceland
KAROLIINA LEHTINEN, Kiel University, Germany and University of Liverpool, United Kingdom

This paper establishes a comprehensive theory of runtime monitorability for Hennessy-Milner logic with recursion, a very expressive variant of the modal μ-calculus. It investigates the monitorability of that logic with a linear-time semantics and then compares the obtained results with ones that were previously presented in the literature for a branching-time setting. Our work establishes an expressiveness hierarchy of monitorable fragments of Hennessy-Milner logic with recursion in a linear-time setting and exactly identifies what kinds of guarantees can be given using runtime monitors for each fragment in the hierarchy. Each fragment is shown to be complete, in the sense that it can express all properties that can be monitored under the corresponding guarantees. The study is carried out using a principled approach to monitoring that connects the semantics of the logic and the operational semantics of monitors. The proposed framework supports the automatic, compositional synthesis of correct monitors from monitorable properties.

CCS Concepts: • **Theory of computation → Logic and verification**; **Modal and temporal logics**; Automata over infinite objects; Complexity theory and logic;

Additional Key Words and Phrases: monitorability, linear-time and branching-time logics, monitor synthesis

ACM Reference Format:
Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. 2019. Adventures in Monitorability: From Branching to Linear Time and Back Again. *Proc. ACM Program. Lang.* 1, POPL, Article 1 (January 2019), 29 pages.

1 INTRODUCTION

The ubiquitous proliferation of software—from high-frequency stock market trading and autonomous vehicles, down to mundane objects such as mobile phones and household appliances—makes a strong case for stringent software correctness requirements. This proliferation has also substantially altered the manner in which software is developed and deployed. Today's software often consists of multiple components (*e.g.,* third-party libraries, mobile apps, microservices, cloud services *etc.*) that are developed and maintained by independent software organisations. In this setting, access to the components' internal workings varies (*e.g.,* open-source versus proprietary code) and different components may be subject to diverse quality controls. Moreover, time-to-market constraints often impose multiple deployment phases where software is rolled out *in stages* and

Authors' addresses: Luca Aceto, Gran Sasso Science Institute, L'Aquila, Italy , School of Computer Science, Reykjavik University, Reykjavik, Iceland, luca@ru.is; Antonis Achilleos, School of Computer Science, Reykjavik University, Reykjavik, Iceland, antonios@ru.is; Adrian Francalanza, Department of Computer Science, ICT, University of Malta, Msida, Malta, adrian.francalanza@um.edu.mt; Anna Ingólfsdóttir, School of Computer Science, Reykjavik University, Reykjavik, Iceland, annai@ru.is; Karoliina Lehtinen, Kiel University, Kiel, Germany , University of Liverpool, Liverpool, United Kingdom, karoliina.lehtinen@liverpool.ac.uk.

2019. 2475-1421/2019/1-ART1 $15.00
https://doi.org/

third-party components change without notice from one deployment phase to the next. Requirements from various stakeholders may also evolve between deployment phases and occasionally become conflicting. These realities suggest that there is no silver bullet for ensuring software correctness. Any adequate solution will most likely need to employ *multiple verification techniques* (*e.g.,* testing, model checking, theorem proving, log analysis, type checking, monitoring *etc.*) in a coherent manner, spanning the various stages of the software development lifecyle.

Runtime Verification (RV) [Bartocci et al. 2018] is a lightweight verification technique that checks for the correctness of the system under scrutiny by analysing the *current execution* exhibited by the system. RV generally assumes a logic (or some other formal language) for describing the correctness specifications of the system. From these specifications, (online) RV generates computational entities called *monitors* that are then instrumented to run with the system so as to *incrementally* analyse its execution (expressed as a trace of captured events) and reach *(irrevocable) judgements* relating to system violations or satisfactions for these specifications. These characteristics make RV an ideal candidate to be used in a multi-pronged approach towards ensuring software correctness: it can verify the correctness of components that are either not available for inspection prior to deployment, or are too expensive to check via more exhaustive and less scalable verification techniques such as model checking [Baier et al. 2008; Clarke et al. 1999]. Importantly, in settings where multiple verification techniques are used, one cannot necessarily expect specifications to be expressed in a language tailored specifically to RV. Indeed, the use of disparate specification logics specific to every verification technique that is used for validating system correctness is expensive. Moreover, an RV-specific property language leads to a poor separation of concerns between the effort required to formulate the specifications and the engineering endeavour needed to determine how to best verify them. Therefore it is natural and important to develop RV foundations that are based on *general-purpose specification languages*, which subsume application-specific verification concerns.

In order for RV to be used effectively in this way, a few foundational questions need to be addressed. Principal among them is the question of *monitorability*: for sufficiently expressive specification logics, it is often the case that some specifications *cannot* be monitored at runtime. For example, the observation of finite executions does not give sufficient information to decide whether the specification "every request is eventually followed by an answer" is satisfied. It is thus important to identify *which* specifiable properties are monitorable and *which are not*, since this directly impinges on whether to use RV or some other verification technique instead. Another fundamental question is that of monitor *correctness*. Monitors are often considered part of the trusted computing base and any errors in their code could either invalidate the runtime analysis they perform or, even worse, compromise the execution of the system itself. In order to ensure monitor correctness, one must first establish what it means for a monitor to *adequately verify* a specification at runtime. In fact, there may be a number of plausible definitions for this notion, each contributing to different monitor implementations. The question of what it means to adequately verify a specification at runtime directly impacts the question of monitorability as well, and guides the design of algorithms for the synthesis of correct monitors from monitorable properties. A third fundamental question concerns the limits of monitor *expressiveness*. After one has established the monitorability of a set of properties from a reasonably general specification logic, it is important to know whether this set contains *all* properties that can be expressed in the logic and can, at the same time, be monitored at runtime. This is the question of *maximality* of the monitorable fragment of the specification language, and its importance lies in the knowledge that one can identify a logical sub-language that syntactically characterises all monitorable properties: syntactic characterisations of monitorable properties provide a core calculus for conducting further studies and facilitate tool construction.

In prior work [Aceto et al. 2017a, 2018a; Francalanza et al. 2017a, 2015, 2017b], these foundational questions have been investigated for a highly expressive logic called Hennessy-Milner Logic with recursion (recHML) [Larsen 1990], a variant of the modal μ-calculus [Kozen 1983], that can embed a variety of widely used logics such as LTL and CTL, thus guaranteeing a good level of generality for the obtained results. A distinctive aspect of this programme of study is the differentiation between the *semantics of the logic* on the one hand, and the *operational semantics of monitors* on the other, which mirrors the separation of concerns required for the multi-pronged verification approach advocated earlier. Within the proposed framework, the definitions of monitorability and correctness emerge naturally as relationships between the two semantics. That is, the relationship between the verdicts reached by a monitor and the satisfaction of a specification by the observed system naturally characterises both the monitor's correctness and the specification's monitorability.

Despite its merits, that body of work remains rather disconnected from the more established classical results on monitorability [Bauer et al. 2010; Chang et al. 1992; Falcone et al. 2012a; Manna and Pnueli 1991; Pnueli and Zaks 2006]. One major complication obstructing a unified understanding of all these monitoring theories is the fact that the former work on recHML is carried out for a *branching-time* semantics, whereas the classical theories target specifications for a *linear-time* semantics. Propitiously, however, the modal μ-calculus also has a well-established linear-time semantics, which can be easily adapted to recHML. This provides us with an opportunity to extend the principled framework developed in Aceto et al. [2017a] and Francalanza et al. [2015, 2017b] to a linear-time setting, offering an ideal basis to better understand the connections between monitorability for branching-time and linear-time specifications. We contend that this framework is general enough to lay the foundations for a potential unified theory of monitorability.

Contributions and Synopsis. This paper sets out to establish a comprehensive theory of monitorability for recHML, by investigating the monitorability of that logic with a linear-time semantics and then comparing the obtained results with those presented in the literature in a branching-time setting. We identify the trade-offs between monitoring guarantees and expressiveness: In general, the more we expect from monitors, the fewer specifications can be monitored. Here we establish an expressiveness hierarchy within linear-time recHML and identify exactly what kind of guarantees can be given for each type of specification.

- We show that, compared to branching time, linear time allows for a much stronger notion of monitorability requiring that a monitor correctly report both the satisfaction and the violation of the property it checks on all system executions. We identify a fragment of recHML that captures exactly linear-time properties with such monitors (Prop. 4.7), and show how to synthesise monitors from them (Def. 4.4).
- For any collection of monitors with irrevocable acceptance and rejection verdicts, which are reported after examining a finite prefix of the observed execution, we show a strong maximality result for the above-mentioned logical fragment (Thm. 4.8), which guarantees that all monitorable properties of traces can be expressed in that fragment of recHML.
- We apply the weaker notion of monitorability called *partial* monitorability from Francalanza et al. [2017b], which guarantees that a monitor does not reach an incorrect verdict and reaches a verdict for either all violations or all satisfactions. Again, we give a syntactic characterisation of linear-time properties that can be monitored with such monitors (Prop. 4.18), we show how to synthesise correct monitors from them (Def. 4.12), and prove maximality results.
- We establish a relationship between specifications that are partially monitorable in branching-time and in linear-time semantics (Sec. 5). To establish this result, we study how considering specifications over *both* finite and infinite executions affects monitorability. Our main observation here is that the syntactic fragment identified as partially monitorable with respect

to branching-time semantics and the one identified as partially monitorable with respect to linear-time semantics are equally expressive under linear-time semantics over a finite set of actions. This bridges the gap in the treatment of monitorability on linear- versus branching-time domains.

Our results establish a unified foundation for an increasingly important verification technique, covering *both* branching-time and linear-time specifications. We establish simple syntactic characterisations for specifications that can be monitored at runtime for various monitor requirements. For each characterisation, we provide a synthesis function that automates the generation of the corresponding monitors, whose correctness proofs depend on delicate arguments about the monitor semantics. This approach facilitates the design and implementation of correct monitors, along the lines of previous work on tool construction [Attard et al. 2017; Attard and Francalanza 2016; Francalanza and Seychell 2015]. Throughout our technical development, we also highlight the subtle aspects of moving between semantics of branching processes, infinite traces, and potentially finite traces, and provide ample discussion on how they affect monitorability. Crucially, our results are not just limited to our line of work. For instance, the syntactic characterisations of monitorable properties set maximality limits to a number of existing RV tools using popular logics such as LTL since these logics can be embedded in our general language recHML.

The proofs of all the results in the paper may be found in the extended version available at http://icetcs.ru.is/theofomon/POPL2019.pdf.

2 PRELIMINARIES

We provide a brief overview of our touchstone logic, recHML [Aceto et al. 2007; Larsen 1990], a reformulation of the highly expressive and extensively studied modal μ-calculus [Kozen 1983].

2.1 The Syntax

The logic described in Fig. 1 is a mild generalisation of recHML [Aceto et al. 2007; Larsen 1990]. It assumes a set of actions, $\alpha, \beta, \ldots \in \textsc{Act}$, together with a distinguished *internal* action τ, where $\tau \notin \textsc{Act}$. We refer to the actions in $\textsc{Act}$ as *external* actions, as opposed to the action τ, and use $\mu \in \textsc{Act} \cup \{\tau\}$ to refer to either. The metavariables $A, B, \ldots \subseteq \textsc{Act}$ range over sets of (external) actions, where the convenient notation $\overline{A}$ is occasionally used to denote $\textsc{Act} \setminus A$; whenever the context allows us to do so unambiguously, singleton sets $\{\alpha\}$ are also occasionally denoted as α, and $\overline{\{\alpha\}}$ is occasionally denoted as $\overline{\alpha}$.

The grammar in Fig. 1 also assumes a countable set of logical variables $X, Y \in \textsc{LVar}$. Apart from the standard constructs for truth, falsehood, conjunction and disjunction, the logic is equipped with existential and universal modal operators that use sets of actions, A. A hallmark of the logic is the use of *two* recursion operators that express least or greatest fixpoints: formulae $\min X.\varphi$ and $\max X.\varphi$ bind free instances of the logical variable X in φ, inducing the usual notions of open/closed formulae and formula equality up to alpha-conversion. A formula is said to be guarded if every fixpoint variable appears within the scope of a modality within its fixpoint binding. All formulae are assumed to be guarded (without loss of expressiveness [Kupferman et al. 2000]). For a formula φ, we use $l(\varphi)$ to denote the length of φ as a string of symbols.

2.2 The models

We provide linear- and branching-time interpretations for the logic. The metavariables $t, u \in \textsc{Trc} = \textsc{Act}^{\omega}$ range over *infinite* sequences of external actions, abstractly representing complete system

Syntax

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

Linear-Time Semantics

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

Branching-Time Semantics

$$\begin{array}{rcl rcl} [\![\mathsf{tt},\rho]\!]_{\mathrm{B}} & \stackrel{\text{def}}{=} & \textsc{Prc} & [\![\mathsf{ff},\rho]\!]_{\mathrm{B}} & \stackrel{\text{def}}{=} & \emptyset \\ [\![\varphi_1{\vee}\varphi_2,\rho]\!]_{\mathrm{B}} & \stackrel{\text{def}}{=} & [\![\varphi_1,\rho]\!]_{\mathrm{B}} \cup [\![\varphi_2,\rho]\!]_{\mathrm{B}} & [\![\varphi_1{\wedge}\varphi_2,\rho]\!]_{\mathrm{B}} & \stackrel{\text{def}}{=} & [\![\varphi_1,\rho]\!]_{\mathrm{B}} \cap [\![\varphi_2,\rho]\!]_{\mathrm{B}} \\ [\![\langle A\rangle\varphi,\rho]\!]_{\mathrm{B}} & \stackrel{\text{def}}{=} & \left\{p \mid \exists q\cdot\exists\alpha\in A\cdot p \overset{\alpha}{\Longrightarrow} q \ \text{ and } \ q\in[\![\varphi,\rho]\!]_{\mathrm{B}}\right\} & & & \\ [\![[A]\varphi,\rho]\!]_{\mathrm{B}} & \stackrel{\text{def}}{=} & \left\{p \mid \forall q\cdot\forall\alpha\in A\cdot p \overset{\alpha}{\Longrightarrow} q \ \text{ implies } \ q\in[\![\varphi,\rho]\!]_{\mathrm{B}}\right\} & & & \\ [\![\mathsf{min}X.\varphi,\rho]\!]_{\mathrm{B}} & \stackrel{\text{def}}{=} & \bigcap\{P \mid [\![\varphi,\rho[X\mapsto P]]\!]_{\mathrm{B}} \subseteq P\} & & & \\ [\![\mathsf{max}X.\varphi,\rho]\!]_{\mathrm{B}} & \stackrel{\text{def}}{=} & \bigcup\{P \mid P \subseteq [\![\varphi,\rho[X\mapsto P]]\!]_{\mathrm{B}}\} & [\![X,\rho]\!]_{\mathrm{B}} & \stackrel{\text{def}}{=} & \rho(X) \end{array}$$

Fig. 1. recHML Syntax, Linear-Time and Branching-Time Semantics

runs; the metavariable $T \subseteq \textsc{Trc}$ ranges over *sets* of traces. *Finite traces*, denoted as $s, r \in \textsc{Act}^*$, represent *finite* prefixes of a system run or finite executions. *Explicit traces*, denoted as $e, f \in (\textsc{Act} \cup \{\tau\})^*$, represent *detailed finite* prefixes of a system run that also include its internal transitions; the function $\lceil e \rceil$ returns the finite trace s that is left after dropping all the τ-actions from e. We say that two explicit traces *agree on the external actions*, denoted as $e_1 \equiv_{\textsc{Act}} e_2$, whenever $\lceil e_1 \rceil = \lceil e_2 \rceil$. A trace (*resp.,* finite trace) with action α at its head is denoted as αt (*resp., αs*). An explicit trace with action μ at its head is denoted as μe. Similarly, a trace with a prefix s and continuation t is denoted as st.

The denotational semantic function $[\![-]\!]_{\mathrm{L}}$ in Fig. 1 maps a formula to a set of traces, and is referred to as the *linear-time* semantics of recHML. It uses valuations that map logical variables to sets of traces, $\sigma : \textsc{LVar} \rightarrow \mathcal{P}(\textsc{Trc})$, to define the semantics by induction on the structure of the formulae. Intuitively, $\sigma(X)$ is the set of traces assumed to satisfy X. The cases for truth, falsehood, disjunction and conjunction are straightforward. An existential modal formula $\langle A\rangle\varphi$ denotes all traces with a prefix action α from the action set A and a continuation that satisfies φ. A universal modal formula $[A]\varphi$ denotes all traces that are either *not* prefixed by any α in A, or have a continuation u satisfying φ. The sets of traces satisfying the least and greatest fixpoint formulae, $\mathsf{min}\,X.\varphi$ and $\mathsf{max}\,X.\varphi$, are defined as intersection (*resp.,* union) of all the pre-fixpoints (*resp.,* post-fixpoints) of the function induced by the formula φ.

The second interpretation of $\textsc{rec}$HML, denoted by $[\![-]\!]_{\mathrm{B}}$, is defined in terms of *processes*, $\textsc{Prc}$, and is referred to as the branching-time semantics. It assumes a set of process states, $p, q, \ldots \in \textsc{Prc}$ where $P \subseteq \textsc{Prc}$, and a transition relation, $\longrightarrow \subseteq (\textsc{Prc} \times (\textsc{Act} \cup \{\tau\}) \times \textsc{Prc})$. The triple $\langle \textsc{Prc}, (\textsc{Act} \cup \{\tau\}), \longrightarrow \rangle$ forms a Labelled Transition System (LTS) [Keller 1976]. The suggestive notation $p \xrightarrow{\mu} p'$ denotes $(p, \mu, p') \in \longrightarrow$; we also write $p \stackrel{\mu}{\not\longrightarrow}$ to denote $\neg(\exists p' \cdot\ p \xrightarrow{\mu} p')$. We employ the usual notation for weak transitions and write $p \Longrightarrow p'$ in lieu of $p(\xrightarrow{\tau})^{*}p'$ and $p \xRightarrow{\mu} p'$ for $p \Longrightarrow \cdot \xrightarrow{\mu} \cdot \Longrightarrow p'$, referring to p' as a μ-derivative of p. As we have done for strong transitions, for weak transitions we use $p \xRightarrow{\mu}$ to denote $\exists p' \cdot\ p \xRightarrow{\mu} p'$ and $p \stackrel{\mu}{\not\Longrightarrow}$ to denote $\neg(\exists p' \cdot\ p \xRightarrow{\mu} p')$. Sequences of weak transitions $p \xRightarrow{\alpha_1} \cdots \xRightarrow{\alpha_n} p'$ are written as $p \xRightarrow{s} p'$, where $s = \alpha_1 \cdots \alpha_n$. Similarly, for strong transitions, $p \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_n} p'$ is written as $p \xrightarrow{e}\!\!\!\!\rightarrow p'$, where $e = \mu_1 \cdots \mu_n$. We say that p produces a trace $t = \alpha_1\alpha_2 \cdots$ if there are processes $p_0, p_1, p_2, \ldots$ such that $p = p_0$ and $p_0 \xRightarrow{\alpha_1} p_1 \xRightarrow{\alpha_2} p_2 \cdots$. While an LTS can be used to model a single system, it can also model all possible system behaviours.

The branching-time semantics in Fig. 1 follows the linear-time semantics for most cases, using a valuation from variables to *sets of processes*, $\rho : \textsc{LVar} \to \mathcal{P}(\textsc{Prc})$, instead. The main differences are with respect to the modal formulae. A universal modal formula $[A]\varphi$ requires *all* α-derivatives of a process, where $\alpha \in A$, to satisfy φ. By contrast, an existential modal formula $\langle A \rangle\varphi$ requires the *existence* of at least one α-derivative, for some $\alpha \in A$, that satisfies φ.

For closed formulae, we use $[\![\varphi]\!]_{\mathrm{L}}$ and $[\![\varphi]\!]_{\mathrm{B}}$ in lieu of $[\![\varphi, \sigma]\!]_{\mathrm{L}}$ and $[\![\varphi, \rho]\!]_{\mathrm{L}}$ (for some σ and ρ) *resp.*, since the semantics is independent of the valuation. We also write $[\![\varphi]\!]$ instead of $[\![\varphi]\!]_{\mathrm{L}}$ or $[\![\varphi]\!]_{\mathrm{B}}$, whenever the correct interpretation can be discerned from the context or the specific interpretation is unimportant. Unless otherwise stated, we assume that the formulae we consider are all *closed.*

Example 2.1 (Expressiveness)**.** For arbitrary formulae $\varphi, \psi \in \textsc{rec}$HML, we can encode the following characteristic LTL operators [Clarke et al. 1999] as:

$$\mathsf{X}\,\varphi \stackrel{\text{def}}{=} \langle \textsc{Act} \rangle \varphi \qquad \varphi\,\mathsf{U}\,\psi \stackrel{\text{def}}{=} \min Y.\big(\psi \vee (\varphi \wedge \langle \textsc{Act} \rangle Y)\big) \qquad \varphi\,\mathsf{R}\,\psi \stackrel{\text{def}}{=} \max Y.\Big((\psi \wedge \varphi) \vee (\psi \wedge \langle \textsc{Act} \rangle Y)\Big) \qquad \square$$

Example 2.2 (Comparison)**.** Assume $\textsc{Act} = \{a, b, c\}$. Consider the two formulae

$$\varphi_1 = [a][a]\mathsf{ff} \qquad\qquad \varphi_2 = [a](\langle a \rangle \mathsf{tt} \vee \langle\, \{b, c\}\, \rangle \mathsf{tt})$$

together with the trace (denoted by the ω-regular expression) $t = (a.b)^{\omega}$, and the (non-deterministic) process (described by the regular CCS syntax [Milner 1989]) $p = \mathsf{rec}\, x.(a.b.x + a.a.x + a.\mathsf{nil})$. In particular, we note that p can produce the infinite trace t.

Whereas $t \in [\![\varphi_1]\!]_{\mathrm{L}}$, we have $p \notin [\![\varphi_1]\!]_{\mathrm{B}}$ because along one branch we have $p \xRightarrow{a} a.p$ and $a.p \notin [\![[a]\mathsf{ff}]\!]_{\mathrm{B}}$. In linear-time semantics, the equality $[\![\langle A \rangle \mathsf{tt} \vee \langle \overline{A} \rangle \mathsf{tt}]\!]_{\mathrm{L}} = [\![\mathsf{tt}]\!]_{\mathrm{L}}$ holds for *each* A. One can also easily deduce that $[\![[A]\mathsf{tt}]\!] = [\![\mathsf{tt}]\!]$ for both linear- and branching-time semantics from the semantics of Fig. 1. Hence, in our case (where $\textsc{Act} = \{a, b, c\}$), we obtain $[\![\langle a \rangle \mathsf{tt} \vee \langle\, \{b, c\}\, \rangle \mathsf{tt}]\!]_{\mathrm{L}} = [\![\mathsf{tt}]\!]_{\mathrm{L}}$ by instantiating $[\![\langle A \rangle \mathsf{tt} \vee \langle \overline{A} \rangle \mathsf{tt}]\!]_{\mathrm{L}} = [\![\mathsf{tt}]\!]_{\mathrm{L}}$ with $A = \{a\}$. As a result, φ_2 is equivalent to tt under linear-time semantics and we have $t \in [\![\varphi_2]\!]_{\mathrm{L}}$ for *every* trace t. However, under branching-time semantics $[\![\langle a \rangle \mathsf{tt} \vee \langle\, \{b, c\}\, \rangle \mathsf{tt}]\!]_{\mathrm{B}} \neq [\![\mathsf{tt}]\!]_{\mathrm{B}}$ (one witness for the inequality is the deadlocked process nil, $[\![\mathsf{tt}]\!]_{\mathrm{B}} \ni \mathsf{nil} \notin [\![\langle a \rangle \mathsf{tt} \vee \langle\, \{b, c\}\, \rangle \mathsf{tt}]\!]_{\mathrm{B}}$). In fact, $p \xrightarrow{a} \mathsf{nil}$ and thus $p \notin [\![\varphi_2]\!]_{\mathrm{B}}$. $\square$

Remark. Action sets A in $[A]\varphi$ and $\langle A \rangle\varphi$ are typically expressed using predicates in tools such as those described in Attard and Francalanza [2016], Attard et al. [2017] and Aceto et al. [2018b]. For example, modalities can be labelled by an output action on port x carrying payload $\langle 8, y \rangle$ where the data variables x and y are constrained by conditions, as in $[\mathbf{out}(x, \langle 8, y \rangle), (192.188.34.42 \geq x \geq 192.188.34.1) \wedge \mathbf{mod}(y) = 1]\varphi$. In the sequel, we shall assume that $\textsc{Act}$ (and thus any action set A) is a *finite* set of actions. This helps to simplify our technical development and enables us to focus on

Syntax

$$
\begin{array}{rlllll}
m, n \in \textsc{REMon} ::= & v & \mid \alpha.m & \mid m + n & \mid \mathsf{rec}\, x.m & \mid x \\
v, u \in \textsc{Verd} ::= & \mathsf{end} & \mid \mathsf{no} & \mid \mathsf{yes} & &
\end{array}
$$

Dynamics

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

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

Instrumentation

$$\textsc{iMon}\ \frac{p \xrightarrow{\alpha} p' \quad m \xrightarrow{\alpha} m'}{m \triangleleft p \xrightarrow{\alpha} m' \triangleleft p'} \qquad \textsc{iTer}\ \frac{p \xrightarrow{\alpha} p' \quad m \not\xrightarrow{\alpha} \quad m \not\xrightarrow{\tau}}{m \triangleleft p \xrightarrow{\alpha} \mathsf{end} \triangleleft p'}$$

$$\textsc{iAsyP}\ \frac{p \xrightarrow{\tau} p'}{m \triangleleft p \xrightarrow{\tau} m \triangleleft p'} \qquad \textsc{iAsyM}\ \frac{m \xrightarrow{\tau} m'}{m \triangleleft p \xrightarrow{\tau} m' \triangleleft p}$$

Fig. 2. Monitors and Instrumentation

the core issues being studied. However, finite action sets are not necessarily a limitation since, in most cases, infinite data sets can be treated in a finite manner using standard symbolic techniques (*e.g.*, see Francalanza [2017] for a recent treatment of the subject in the context of monitors). □

Remark. For a finite set I of indices, the (standard) notation $\bigwedge_{i\in I} \varphi_i$ denotes tt when $I = \emptyset$, and a conjunction of the formulae in $\{\varphi_i \mid i \in I\}$ when $I \neq \emptyset$. Similarly $\bigvee_{i\in I} \varphi_i$ denotes ff when $I = \emptyset$, and a disjunction of the formulae in $\{\varphi_i \mid i \in I\}$ when $I \neq \emptyset$. These notations are justified by the fact that $\vee$ and $\wedge$ are commutative and associative with respect to all the semantics considered in the paper. We also observe that, for both semantics, $[A]\varphi$ is equivalent to $\bigwedge_{\alpha\in A} [\alpha]\varphi$, and $\langle A\rangle\varphi$ is equivalent to $\bigvee_{\alpha\in A} \langle\alpha\rangle\varphi$ for finite A, so we use these equivalent notations interchangeably. □

3 A MONITORING FRAMEWORK

A distinctive feature of the work in Aceto et al. [2017a, 2018a] and Francalanza et al. [2017b] is the full description of the monitoring setup used, which incorporates the monitor definition together with the system instrumentation mechanism—monitor compositionality results have shown that the semantics of monitors in an instrumented setup differs substantially from that given for monitors in isolation [Francalanza 2016, 2017]. Here we follow this comprehensive approach.

3.1 Regular Monitors

Regular monitors are LTSs defined by the grammar and transition rules in Fig. 2, used already in Aceto et al. [2017a] and Francalanza et al. [2017b]. A transition $m \xrightarrow{\alpha} n$ denotes that the monitor in state m can *analyse* the (external) action α and transition to state n. Monitors may reach any one of *three* verdicts after analysing a finite trace: *acceptance*, yes, *rejection*, no, and the *inconclusive* verdict end. We highlight the transition rule for verdicts in Fig. 2, describing the fact that from a verdict state any action can be analysed by transitioning to the same state; verdicts are thus *irrevocable.* The remaining constructs and transitions are standard. If *at most one* of the verdicts yes, no appears in m, then m is called a *single-verdict monitor.* Otherwise, m is called a dual-verdict monitor. Just like for formulae, we use $l(m)$ to denote the length of m as a string of symbols. In the sequel, for a finite nonempty set of indices I, we use notation $\sum_{i\in I} m_i$ to denote a combination of the monitors in $\{m_i \mid i \in I\}$ using the operator $+$. The notation is justified, because $+$ is commutative and associative with respect to the transitions that a resulting monitor can exhibit. We also use the shorthand notation $A.m$ to denote $\sum_{\alpha\in A} \alpha.m$ (for finite non-empty A). The regular monitors in Fig. 2 have an important property, namely that their state space, *i.e.,* the set of reachable states, is finite. This is a valuable property for ensuring reasonable overheads in terms of the amount of memory the monitor will use at runtime (see Prop. 3.2).

Lemma 3.1 (Verdict Persistence). *$v \overset{e}{\longrightarrow\!\!\!\!\!\rightarrow} m$ implies $m = v$.*

Definition 3.1 (Monitor Reachable States). $\mathsf{reach}(m) \overset{\text{def}}{=} \{\, n \mid \exists e \cdot m \overset{e}{\longrightarrow\!\!\!\!\!\rightarrow} n \,\}$. □

Proposition 3.2. *Regular monitors are finite state i.e., for all $m \in$ REMON, reach(m) is finite.* □

We define the following behavioural predicate on monitors, which relates to their correctness.

Definition 3.2 (Monitor Consistency). A monitor m is *consistent* when there is *no* finite trace s such that $m \overset{s}{\Longrightarrow} \mathsf{yes}$ and $m \overset{s}{\Longrightarrow} \mathsf{no}$. □

Monitors are intended to run in conjunction with the system (*i.e.,* process) they are analysing. Following Francalanza [2016, 2017] and Francalanza et al. [2017b], Fig. 2 defines a transition relation for a process p instrumented with a monitor m, denoted as $m \triangleleft p$. The relation is parametric with respect to the transition semantics of the process p and the monitor, as long as the latter includes the inconclusive verdict end (*e.g.,* the monitor transition semantics given in Fig. 2 does). The semantics relegates the monitor m to a *passive role* in an instrumented system $m \triangleleft p$, meaning that $m \triangleleft p$ transitions with an external action α only when p transitions with that action. For instance, when p transitions with action α to some p', and m can analyse this action and transition to state m', the instrumented pair transitions in lockstep to $m' \triangleleft p'$; see rule iMON. Conversely, if p wants to transition with an action α that the instrumented monitor is *not* able to analyse (perhaps due to underspecification), the instrumented system is *still* allowed to transition with α, but the monitor analysis is prematurely aborted to the inconclusive state; see rule iTER. The other rules allow monitors and processes to execute independently of one another with respect to internal (τ-)moves.

Example 3.1. When the monitor $\mathsf{rec}\, x.(a.x + b.\mathsf{yes})$ is instrumented with the process $a.\mathsf{rec}\, x.b.x$, it can reach an acceptance verdict thus:

$$\mathsf{rec}\, x.(a.x + b.\mathsf{yes}) \triangleleft a.\mathsf{rec}\, x.b.x \xrightarrow{\tau} (a.(\mathsf{rec}\, x.(a.x + b.\mathsf{yes})) + b.\mathsf{yes}) \triangleleft a.\mathsf{rec}\, x.b.x \xrightarrow{a}$$

$$\mathsf{rec}\, x.(a.x + b.\mathsf{yes}) \triangleleft \mathsf{rec}\, x.b.x \overset{\tau\tau}{\longrightarrow\!\!\!\!\!\rightarrow} a.\mathsf{rec}\, x.(a.x + b.\mathsf{yes}) + b.\mathsf{yes} \triangleleft b.\mathsf{rec}\, x.b.x \xrightarrow{b} \mathsf{yes} \triangleleft \mathsf{rec}\, x.b.x.$$

However, if the same process is instrumented with a slightly different monitor $\mathsf{rec}\, x.(a.a.x + b.\mathsf{yes})$ we obtain a different verdict.

$$\mathsf{rec}\, x.(a.a.x + b.\mathsf{yes}) \triangleleft a.\mathsf{rec}\, x.b.x \xrightarrow{\tau} (a.a.(\mathsf{rec}\, x.(a.a.x + b.\mathsf{yes})) + b.\mathsf{yes}) \triangleleft a.\mathsf{rec}\, x.b.x \xrightarrow{a}$$

$$a.\mathsf{rec}\, x.(a.x + b.\mathsf{yes}) \triangleleft \mathsf{rec}\, x.b.x \xrightarrow{\tau} a.\mathsf{rec}\, x.(a.x + b.\mathsf{yes}) \triangleleft b.\mathsf{rec}\, x.b.x \xrightarrow{b} \mathsf{end} \triangleleft \mathsf{rec}\, x.b.x$$

The last transition is obtained via rule iTER, whereby the process exhibited an action that the current monitor state was unable to analyse (*i.e.,* it could only analyse action a, not b). □

The following lemmata describe how the respective monitor and system LTSs can be composed and decomposed according to instrumentation [Francalanza 2016; Francalanza et al. 2017b].

Syntax

$$\begin{array}{lllllll} m,n \in \textsc{Mon} ::= v & \mid \alpha.m & \mid m+n & \mid \mathsf{rec}\,x.m & \mid x \\ & \mid m\otimes n & \text{(conj. para.)} & \mid m\oplus n & \text{(disj. para.)} \end{array}$$

Dynamics

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

$$\textsc{mVrC1}\ \frac{}{\mathsf{yes}\otimes m \xrightarrow{\tau} m} \qquad \textsc{mVrC2}\ \frac{}{\mathsf{no}\otimes m \xrightarrow{\tau} \mathsf{no}} \qquad \textsc{mVrD1}\ \frac{}{\mathsf{no}\oplus m \xrightarrow{\tau} m} \qquad \textsc{mVrD2}\ \frac{}{\mathsf{yes}\oplus m \xrightarrow{\tau} \mathsf{yes}}$$

Fig. 3. Parallel Monitors. The syntax and dynamics of parallel monitors are extensions of the ones for regular monitors, as presented in Fig. 2. Parallel monitors use the same instrumentation as regular monitors.

Lemma 3.3 (General Unzipping). *$m \triangleleft p \overset{s}{\Longrightarrow} n \triangleleft q$ implies*

- *$p \overset{s}{\Longrightarrow} q$ and*
- *$m \overset{s}{\Longrightarrow} n$ or ($\exists s_1, s_2, \alpha, m' \cdot s = s_1\alpha s_2$ and $m \overset{s_1}{\Longrightarrow} m' \not\xrightarrow{\tau}$ and $m' \not\xrightarrow{\alpha}$ and $n = \mathsf{end}$).* □

Lemma 3.4 (Zipping). *($p \overset{s}{\Longrightarrow} q$ and $m \overset{s}{\Longrightarrow} n$) implies $m \triangleleft p \overset{s}{\Longrightarrow} n \triangleleft q$.* □

Within this framework, we can formalise our understanding of process and trace acceptance and rejection by a monitor. Acceptances and rejections will constitute the monitoring counterpart to formula satisfactions and violations from Sec. 2 when we consider our definitions of monitorability.

Definition 3.3 (Process and Trace Acceptance and Rejection)**.** A monitor m *rejects p along s*, denoted as $\mathbf{rej}(m,p,s)$, if $m \triangleleft p \overset{s}{\Longrightarrow} \mathsf{no} \triangleleft p'$ for some p'. Similarly, m *accepts p along s*, denoted as $\mathbf{acc}(m,p,s)$, if $m \triangleleft p \overset{s}{\Longrightarrow} \mathsf{yes} \triangleleft p'$ for some p'.

- A monitor m rejects (*resp.*, accepts) t, using the abuse of notation $\mathbf{rej}(m,t)$ (*resp.*, $\mathbf{acc}(m,t)$), if $\exists p, s, u$ such that $t = su$ and $\mathbf{rej}(m,p,s)$ (*resp.*, $\mathbf{acc}(m,p,s)$).
- A monitor m rejects (*resp.*, accepts) p, using the abuse of notation $\mathbf{rej}(m,p)$ (*resp.*, $\mathbf{acc}(m,p)$), if $\exists s$ such that $\mathbf{rej}(m,p,s)$ (*resp.*, $\mathbf{acc}(m,p,s)$).

We also say that *m rejects s* as a shorthand for $\exists p \cdot \mathbf{rej}(m,p,s)$, and similarly, *$m$ accepts s* is a shorthand for $\exists p \cdot \mathbf{acc}(m,p,s)$. □

As Def. 3.3 and Lems. 3.3 and 3.4 make clear, a monitor accepts or rejects a finite trace s iff it can transition to the appropriate verdict by reading s. This hints at the fact that each monitor might be "equivalent to a deterministic one". As we will see in Prop. 3.11, this is indeed the case.

3.2 Parallel Composition of Monitors

When relating monitors to formulae, it may be convenient *not* to view monitors as one monolithic entity but rather as a *system of sub-monitors* where the constituent submonitors are concerned with checking specific subformulae. For instance, the use of sub-monitors executing in parallel facilitates the synthesis of monitors from formulae in a *compositional* fashion. Monitors with parallel composition, $m, n \in \textsc{Mon}$, are defined by the grammar and transition rules in Fig. 3. In particular, we endow monitors with conjunctive parallelism, $\otimes$, and disjunctive parallelism, $\oplus$. We use the notation $\odot$ to range over either $\otimes$ or $\oplus$ (*i.e.,* $\odot \in \{\otimes, \oplus\}$).

Fig. 3 also outlines the behaviour of parallel monitors. Rule mPar states that *both* submonitors need to be able to analyse an external action α for their parallel composition to transition with that action. The rules in Fig. 3 also allow τ-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 a conjunctive parallel compositions (Fig. 3 omits obvious 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. Finally, rules mTauL and its dual mTauR (omitted) are contextual rules for these monitor reconfiguration steps.

We identify a useful monitor predicate that obviates the need for the rule iTer of Fig. 2 that prematurely terminates monitors; see Example 3.1. In the case of parallel monitors, it also allows us to neatly decompose the monitor behaviour in terms of the respective sub-monitors.

Definition 3.4 (Monitor Reactivity). We call a monitor m *reactive* when for every $n \in \mathrm{reach}(m)$ and $\alpha \in \mathrm{Act}$, there is some n' such that $n \overset{\alpha}{\Longrightarrow} n'$. □

Example 3.2 below indicates why the assumption that m_1 and m_2 are reactive is needed in Lem. 3.5, which states that parallel monitors behave as expected with respect to the acceptance and rejection of traces as long as the consitituent submonitors are reactive.

Example 3.2. Assume that $\mathrm{Act} = \{a, b\}$. The monitors $a.\mathrm{yes} + b.\mathrm{no}$ and $\mathrm{rec}\, x.(a.x + b.\mathrm{yes})$ are both reactive. The monitor $m = a.\mathrm{yes} \otimes b.\mathrm{no}$, however, is *not* reactive. Since the submonitor $a.\mathrm{yes}$ can only transition with a, according to the rules of Fig. 3, m cannot transition with any action that is not a. Similarly, as the submonitor $b.\mathrm{no}$ can only transition with b, m cannot transition with any action that is not b. Thus, m cannot transition to any monitor, and therefore it cannot reject or accept any trace. By contrast, the monitor $n = (a.\mathrm{yes} + b.\mathrm{end}) \otimes (b.\mathrm{yes} + a.\mathrm{end})$ is reactive, because its constituent submonitors are reactive as well. □

Lemma 3.5 (Monitor Composition and Decomposition). *For reactive m_1 and m_2:*

- *$m_1 \otimes m_2$ rejects t if and only if either m_1 or m_2 rejects t.*
- *$m_1 \otimes m_2$ accepts t if and only if both m_1 and m_2 accept t.*
- *$m_1 \oplus m_2$ rejects t if and only if both m_1 and m_2 reject t.*
- *$m_1 \oplus m_2$ accepts t if and only if either m_1 or m_2 accepts t.* □

Parallel monitors are a convenient formalism for constructing monitors in a compositional fashion and facilitate the definition of monitor synthesis functions from a specification logic. However, these monitors are only as expressive as regular monitors, as Prop. 3.8 demonstrates. Sec. 3.3 is devoted to the proof of this result.

3.3 Monitor Transformations: Parallel to Regular

We describe how one can transform a parallel monitor to a verdict-equivalent regular one. For this, we use known results about alternating finite automata, restated here for completeness.

Definition 3.5 (Alternating Automata). An alternating finite automaton is a quintuple $A = (Q, \Sigma, q_0, \delta, F)$, where Q is a finite set of states, Σ is a finite alphabet, q_0 is the starting state, $F \subseteq Q$ is the set of accepting/final states, and $\delta : (Q \times \Sigma) \to (2^Q \to \{0, 1\})$ is the transition function. An alternating finite automaton is non-deterministic (NFA) if for each $\alpha \in \Sigma$ and $q \in Q$, there is some $S_{q,\alpha} \subseteq Q$, such that for all $S \subseteq Q$, $\delta(q, \alpha)(S) = 1$ if and only if $S \cap S_{q,a} \neq \emptyset$. □

Intuitively, given a state $q \in Q$ and a symbol $\alpha \in \Sigma$, δ returns a boolean function on 2^Q that evaluates, given a truth-assignment on the states of Q (represented by a subset of Q), an assigned truth-value for q. We can extend the transition function to $\delta^* : (Q \times \Sigma^*) \to (2^Q \to \{0, 1\})$, so that $\delta^*(q, \varepsilon)(R) = 1$ iff $q \in R$, and $\delta^*(q, \alpha w)(R) = \delta(q, \alpha)(\{q' \in Q \mid \delta^*(q', w)(R) = 1\})$. We say that the automaton *accepts* $w \in \Sigma^*$ when $\delta^*(q_0, w)(F) = 1$, and that it recognizes $L \subseteq \Sigma^*$ when L is the set of strings accepted by the automaton.

Definition 3.6 (Monitor Language Acceptance and Rejection)**.** A monitor m accepts (*resp.*, rejects) a set of finite traces (*i.e.*, a language) $L \subseteq \textsc{Act}^*$ when for every $s \in \textsc{Act}^*$, $s \in L$ if and only if m accepts (*resp.*, rejects) s. We call the set that m accepts (*resp.*, rejects) $L_a(m)$ (*resp.*, $L_r(m)$). □

Proposition 3.6. *For every reactive parallel monitor m, there is an alternating automaton that accepts $L_a(m)$ and one that accepts $L_r(m)$.*

Proof. We describe the process of constructing an alternating automaton that accepts $L_a(m)$ — the case for $L_r(m)$ is similar. We assume that for every variable x that appears in m, there is a unique submonitor of m of the form $\mathrm{rec}\, x.n$, such that x appears in n. The automaton for m is $A_m = (Q, \textsc{Act}, m, \delta, F)$, where

- Q is the set of submonitors of m;
- $F = \{n \in Q \mid n \text{ accepts } \varepsilon\}$;
- Let for every $S \subseteq Q$, $\delta_0(q, \alpha)(S) = 1$ iff $q \in F$; δ is the closure of δ_0 under the following conditions. For every $S \subseteq Q$:
  - if $n \in S$, then $\delta(\alpha.n, \alpha)(S) = 1$;
  - if $\delta(n, \alpha)(S) = 1$ or $\delta(n', \alpha)(S) = 1$, then $\delta(n + n', \alpha)(S) = 1$;
  - if $\delta(n, \alpha)(S) = 1$ or $\delta(n', \alpha)(S) = 1$, and $n \overset{\alpha}{\Longrightarrow}$ and $n' \overset{\alpha}{\Longrightarrow}$, then $\delta(n \oplus n', \alpha)(S) = 1$;
  - if $\delta(n, \alpha)(S) = 1$ and $\delta(n', \alpha)(S) = 1$, then $\delta(n \otimes n', \alpha)(S) = 1$;
  - if $\delta(n, \alpha)(S) = 1$ and $\mathrm{rec}\, x.n \in Q$, then $\delta(\mathrm{rec}\, x.n, \alpha)(S) = \delta(x, \alpha)(S) = 1$.

To complete the proof, we show that m accepts s if and only if $\delta^*(m, s)(F) = 1$. □

Remark. The assumption that the monitor is reactive is necessary for the construction in the proof of Prop. 3.6 to be correct. Consider, for example, the monitor $m_1 = a.a.\mathrm{yes} \oplus a.b.\mathrm{yes}$. Although $a.a.\mathrm{yes} \xrightarrow{a} a.\mathrm{yes} \xrightarrow{a} \mathrm{yes}$, the monitor does not accept any trace since $b.\mathrm{yes} \overset{a}{\not\longrightarrow}$. By the construction, in the resulting alternating automaton, $F = \{\mathrm{yes}\}$, and therefore $\delta(\mathrm{yes}, a)(F) = 1$, implying that $\delta(a.\mathrm{yes}, a)(F) = 1$, in turn implying that $\delta^*(m_1, aa)(F) = 1$, according to the closure conditions for δ. Therefore, aa is a finite trace that the automaton accepts and the monitor does not.

In light of our assumption that monitor m in Prop. 3.6 is reactive, the third condition for δ in the construction in the proof of the proposition may seem superfluous. However, reactivity does not transfer to submonitors. For example, let $m_2 = (a.\mathrm{yes} \oplus b.\mathrm{yes}) + a.\mathrm{end} + b.\mathrm{end}$. Reasoning similarly to the above argument for m_1, m_2 is a reactive parallel monitor, which accepts no traces. On the other hand, a more naive construction that ensures that $\delta(n \oplus n', \alpha)(S) = 1$ whenever $\delta(n, \alpha)(S) = 1$ or $\delta(n', \alpha)(S) = 1$, would result in an automaton that accepts the finite trace a.

As we see in the remainder of this section, Prop. 3.6 implies that potentially *infinite-state* parallel monitors are equivalent to *finite-state* regular monitors. The subtleties that we pointed out are the trade-off for keeping the construction of the alternating automaton straightforward. □

Corollary 3.7. *For every reactive parallel monitor m, there is an NFA that accepts $L_a(m)$ and an NFA that accepts $L_r(m)$, and each has at most $2^{l(m)}$ states.*

Proof. The alternating automaton that is constructed in the proof of Prop. 3.6 has at most as many states as there are submonitors in m which, in turn, are not more than $l(m)$. Furthermore, it is a known result that every alternating automaton with k states can be converted into an NFA with at most 2^k states that accepts the same language [Chandra et al. 1981; Fellah et al. 1990]. □

We now have all the ingredients to complete the proof of Prop. 3.8. This relies on a notion of monitor equivalence from Aceto et al. [2017b] that focusses on how monitors can reach verdicts.

Definition 3.7 (Verdict Equivalence). Monitors m and n are *acceptance equivalent* (*resp.*, rejection equivalent), denoted as $m \simeq_{\mathrm{acc}} n$ (*resp.*, $m \simeq_{\mathrm{rej}} n$), if for every finite trace s, $m \overset{s}{\Rightarrow}$ yes iff $n \overset{s}{\Rightarrow}$ yes (*resp.*, $m \overset{s}{\Rightarrow}$ no iff $n \overset{s}{\Rightarrow}$ no). They are *verdict equivalent*, denoted as $m \simeq_{\mathrm{ver}} n$, if they are both acceptance- and rejection-equivalent. □

Proposition 3.8. *For all reactive parallel monitors m, there exist regular monitors n_1, n_2, and n such that n_1 and n_2 are single-verdict monitors that are respectively acceptance-equivalent and rejection-equivalent to m, and m and n are verdict equivalent, and $l(n_1), l(n_2), l(n) = 2^{O\left(l(m)\cdot 2^{l(m)}\right)}$.*

Proof. Let A^a_m be an NFA for $L_a(m)$ with at most $2^{l(m)}$ states, and let A^r_m be an NFA for $L_r(m)$ with at most $2^{l(m)}$ states, which exist by Cor. 3.7. From these NFAs, we can construct regular monitors m^a_R and m^r_R, such that m^a_R accepts $L_a(m)$ and m^r_R rejects $L_r(m)$, and $l(m^a_R), l(m^r_R) = 2^{O\left(l(m)\cdot 2^{l(m)}\right)}$ [Aceto et al. 2016]. Therefore, $m^a_R \simeq_{\mathrm{acc}} m$ and $m^r_R \simeq_{\mathrm{rej}} m$, and $m^a_R + m^r_R$ is regular and verdict-equivalent to m, and $l(m^a_R + m^r_R) = 2^{O\left(l(m)\cdot 2^{l(m)}\right)}$. □

The techniques of Aceto et al. [2016] can also be used to produce deterministic monitors.

Definition 3.8 ([Aceto et al. 2016]). A regular monitor m is *syntactically deterministic* iff every sum of at least two summands which appears in m is of the form $\sum_{\alpha\in A} \alpha.m_\alpha$, where $A \subseteq$ Act. □

Example 3.3. The monitor $a.b.$yes $+ a.a.$no is not syntactically deterministic while the verdict-equivalent monitor $a.(b.$yes $+ a.$no$)$ is syntactically deterministic. □

One can also consider non-syntactic notions of determinism, such as if $m \overset{s}{\Longrightarrow} n$ and $m \overset{s}{\Longrightarrow} n'$, then $n \simeq_{\mathrm{ver}} n'$. Lem. 3.9 shows that syntactic determinism implies this semantic notion. Henceforth we will simply say deterministic to mean syntactically deterministic.

Lemma 3.9 ([Aceto et al. 2016]). *If m is deterministic, $m \overset{s}{\Longrightarrow} n$, and $m \overset{s}{\Longrightarrow} n'$, then $n \simeq_{ver} n'$.* □

Theorem 3.10 ([Aceto et al. 2016]). *For every consistent regular monitor m, there is a verdict-equivalent deterministic regular monitor n such that $l(n) = 2^{2^{O(l(m))}}$.* □

Proposition 3.11. *For every consistent reactive parallel monitor m, there is a verdict-equivalent deterministic regular monitor n such that $l(n) = 2^{2^{2^{O\left(l(m)\cdot 2^{l(m)}\right)}}}$.*

Proof. Using Prop. 3.8, m can be translated into a (possibly nondeterministic) verdict-equivalent (hence consistent) regular monitor n_r, such that $l(n_r) = 2^{O\left(l(m)\cdot 2^{l(m)}\right)}$. Thm. 3.10 can then be used to convert n_r into a verdict-equivalent deterministic regular monitor n, such that $l(n) = 2^{2^{O(l(n_r))}}$. Therefore, $l(n) = 2^{2^{2^{O\left(l(m)\cdot 2^{l(m)}\right)}}}$. □

4 MONITORABILITY FOR recHML

Monitorability is the study of the relationship between the semantics of a logic on the one hand (*i.e.,* satisfactions and violations), and the verdicts that can be discerned by the monitoring setup on the other (*i.e.,* acceptances and rejections). The concept relies on what a *correct* monitor for a particular formula is, which, in turn, defines what it means for a formula to be *monitorable.* In this section we focus on the monitorability of recHML. Based on the definition of trace acceptance and rejection of Def. 3.3, we adapt the concepts of monitor soundness and completeness (with respect to a formula) from Francalanza et al. [2017b] to the linear-time setting.

Definition 4.1 (Linear-time Monitor Soundness and Completeness)**.**

- A monitor m is *sound* for a (closed) formula φ of recHML over traces if, *for all* $t \in \text{Trc}$:
  - $\mathbf{rej}(m,t)$ implies $t \notin [\![\varphi]\!]_{\text{L}}$;
  - $\mathbf{acc}(m,t)$ implies $t \in [\![\varphi]\!]_{\text{L}}$.
- A monitor m is *violation-complete* for a (closed) formula φ of recHML over traces if *for all* $t \in \text{Trc}$, $t \notin [\![\varphi]\!]_{\text{L}}$ implies $\mathbf{rej}(m,t)$. It is *satisfaction-complete* if $t \in [\![\varphi]\!]_{\text{L}}$ implies $\mathbf{acc}(m,t)$.
- A monitor m is *complete* for a (closed) formula φ of recHML if it is *both* violation- and satisfaction-complete for it. □

The definition of soundness and completeness for monitors depends on the semantics given to the formulae. Since we focus on linear-time semantics in this section, instead of saying that a monitor is sound or violation- or satisfaction-complete, or complete for a formula over traces, we respectively simply say that it is sound or violation- or satisfaction-complete, or complete for the formula. In Sec. 5, we will introduce variations of Def. 4.1 that depend on different semantics for recHML. Observe that a monitor that is *sound* for some formula must be *consistent*.

Following Francalanza et al. [2017b], we assume that the minimum requirement for a monitor to correctly correlate to a formula is for it to be *sound*. It can be however argued that, depending on the circumstance of the application requirements, different notions of completeness may be deemed adequate enough. It turns out that *not* all formulae can be monitored adequately at runtime. Moreover, the more stringent the requirement for adequate monitoring, the more are the formulae that *cannot* be monitored. In the remainder of the section, we consider different definitions for adequate monitoring and establish recHML monitorability results in each case.

In Sec. 4.1, we present monitorability results with respect to complete monitors. In Sec. 4.2, we introduce the additional requirement of *tightness* for a monitor, under which the monitor reaches a verdict as soon as it has read sufficient information from the input trace and not later. We explain what one needs to do to construct a tight monitor. In Sec. 4.3, we establish monitorability results for *partially complete* monitors, which are satisfaction-complete or violation-complete for their respective formulae, but are not required to be both. This relaxation allows us to monitor for more formulae. Finally, in Sec. 4.4, we examine what one must do to construct tight partially complete monitors, and we explain why the methods of Sec. 4.2 are not likely to apply for this case.

4.1 Complete Monitorability

We first consider *(sound and) complete* monitors as our notion of adequate monitoring for a particular formula. This induces the following definition of monitorable formula and (sub)logic.

Definition 4.2 (Complete Monitorability)**.** A formula $\varphi \in$ recHML is *complete-monitorable* over traces iff there *exists* a monitor m that is sound and complete for it. A (sub)logic $\mathcal{L} \subseteq$ recHML is *complete-monitorable* over traces iff each formula $\varphi \in \mathcal{L}$ is complete-monitorable. □

Remark. In this section we only use Def. 4.2 for the linear-time interpretation of recHML. However, its general form allows it to be used for other interpretations of the logic, with the appropriate adaptation of complete monitors (*e.g.*, along the lines of Francalanza et al. [2017b]). □

As the following results highlight, soundness and completeness for monitors are invariant under verdict equivalence.

Proposition 4.1. *If m is sound and complete for φ then*

(1) $m \simeq_{ver} n$ implies n is sound and complete for φ;
(2) m is a sound and complete monitor for φ' implies $[\![\varphi]\!]_L = [\![\varphi']\!]_L$. □

In line with other works on monitorability [Bauer et al. 2010; Chang et al. 1992; Cini and Francalanza 2015; Falcone et al. 2012a; Francalanza et al. 2017b; Manna and Pnueli 1991; Pnueli and Zaks 2006], *not* all properties in recHML are complete monitorable.

Example 4.1. The formula $\varphi_1 = \langle a\rangle \mathsf{tt} \,\mathsf{U}\, \langle b\rangle\mathsf{tt}$ is *not* complete-monitorable. For if, by contradiction, we assume that it was then there must *exist* some sound and complete monitor m for φ_1. Since the trace $a^\omega \notin [\![\varphi_1]\!]_{\mathrm{L}}$, this monitor m rejects a^ω which, by Def. 3.3, means that it must reach a violation after observing a *finite* prefix a^k (for $k \geq 0$). But this would also mean that m rejects *all* traces of the form $a^k bt$, which clearly satisfy φ_1, thereby contradicting the assumption that m is sound. Similarly, it can be argued that the formula $\varphi_2 = \langle a\rangle\langle b\rangle\langle b\rangle\mathsf{tt}\,\mathsf{R}\,\langle a\rangle\mathsf{tt}$ is *not* complete-monitorable either. For if it was, a sound and complete monitor m_2 would accept the trace a^ω after analysing some prefix a^n of it; this would also mean that this monitor would also accept any trace of the form $a^n bat$, which clearly violates the property. Thus, no such monitor exists. □

Example 4.1 raises the question as to which recHML properties can be monitored according to Def. 4.2. To answer this question, we first identify a fragment of recHML that is guaranteed to be complete-monitorable and then show its maximality.

Definition 4.3 (The complete-monitorable fragment of recHML)**.** The recursion-free syntactic fragment of recHML (a syntactic variant of HML [Hennessy and Milner 1985]) is defined as:

$$\varphi, \psi \in \mathrm{HML} ::= \mathsf{tt} \quad | \; \mathsf{ff} \quad | \; \varphi \vee \psi \quad | \; \varphi \wedge \psi \quad | \; \langle A\rangle\varphi \quad | \; [A]\varphi. \qquad □$$

For every formula $\varphi \in \mathrm{HML}$, we can define a monitor synthesis function as follows.

Definition 4.4 (Complete Monitor Synthesis)**.** The function $\mathsf{m}(-) : \mathrm{HML} \to \textsc{Mon}$ is defined inductively as follows:

$$\begin{array}{lll} \mathsf{m}(\mathsf{ff}) \stackrel{\text{def}}{=} \mathsf{no} & \mathsf{m}(\varphi_1 \wedge \varphi_2) \stackrel{\text{def}}{=} \mathsf{m}(\varphi_1) \otimes \mathsf{m}(\varphi_2) & \mathsf{m}([A]\varphi) \stackrel{\text{def}}{=} A.\mathsf{m}(\varphi) + \overline{A}.\mathsf{yes} \\ \mathsf{m}(\mathsf{tt}) \stackrel{\text{def}}{=} \mathsf{yes} & \mathsf{m}(\varphi_1 \vee \varphi_2) \stackrel{\text{def}}{=} \mathsf{m}(\varphi_1) \oplus \mathsf{m}(\varphi_2) & \mathsf{m}(\langle A\rangle\varphi) \stackrel{\text{def}}{=} A.\mathsf{m}(\varphi) + \overline{A}.\mathsf{no}. \end{array} \qquad □$$

Lemma 4.2. *For all $\varphi \in \mathrm{HML}$, $\mathsf{m}(\varphi)$ is reactive.* □

Example 4.2. Assuming $\textsc{Act} = \{a, b, c\}$, the synthesised monitor for $\varphi = [a]\langle b\rangle\mathsf{tt} \wedge \langle a\rangle[c]\mathsf{ff}$, where $[\![\varphi]\!]_{\mathrm{L}} = \{\, abt \mid t \in \textsc{Act}^\omega \,\}$, is

$$\mathsf{m}(\varphi) = m = \Big(a.(b.\mathsf{yes} + \{a,c\}\,.\mathsf{no}) + \{b,c\}\,.\mathsf{yes}\Big) \otimes \Big(a.(c.\mathsf{no} + \{a,b\}\,.\mathsf{yes}) + \{b,c\}\,.\mathsf{no}\Big).$$

When we compose m with $p = \mathrm{rec}\,x.a.b.x$, we observe the following monitored behaviour:

$$m \triangleleft p \xrightarrow{\tau\, a}\!\!\!\!\rightarrow \Big((b.\mathsf{yes} + \{a,c\}\,.\mathsf{no}) \otimes (c.\mathsf{no} + \{a,b\}\,.\mathsf{yes})\Big) \triangleleft b.p \xrightarrow{b} \mathsf{yes} \otimes \mathsf{yes} \triangleleft p \xrightarrow{\tau} \mathsf{yes} \triangleleft p. \qquad □$$

We show that, for each formula $\varphi \in \mathrm{HML}$, the monitor $\mathsf{m}(\varphi)$ is the witness sound and complete monitor for it. This, in turn, shows that HML is complete-monitorable, in the sense of Def. 4.2.

Proposition 4.3. *For all $\varphi \in \mathrm{HML}$, $\mathsf{m}(\varphi)$ is a sound and complete monitor for φ.* □

Corollary 4.4. *HML is complete monitorable.* □

Following Francalanza et al. [2017b], we go one step further and show that the fragment HML of Def. 4.3 is *maximally expressive* with respect to sound and complete monitors. By this we mean that *every* formula $\varphi \in$ recHML that is complete-monitorable, in the sense of Def. 4.2, is semantically equivalent to a formula from HML. Thus, we can limit ourselves to the syntactic fragment HML without sacrificing any expressiveness in terms of complete-monitorable properties.

We show this claim in two steps. First, we tighten expressiveness results from Sec. 3 for the specific case of complete monitoring. Concretely, we argue that every complete-monitorable formula

(Def. 4.2) can be monitored adequately by a *recursion-free* syntactically deterministic monitor (see Def. 3.8). This is shown via Lem. 4.5, which relies on Def. 4.5. In the second step, we devise an inverse synthesis function to obtain complete-monitorable HML formulae from *recursion-free* deterministic monitors, Lem. 4.6. This formula synthesis function is then used for Prop. 4.7, the last main result of Sec. 4.1.

Definition 4.5 (Removing Monitor Recursion). For each monitor m, we define $\mathsf{noR}(m)$ thus:

$$\mathsf{noR}(x) \overset{\text{def}}{=} \mathsf{end} \qquad \mathsf{noR}(v) \overset{\text{def}}{=} v \qquad \mathsf{noR}(\mathsf{rec}\, x.n) \overset{\text{def}}{=} \mathsf{noR}(n)$$
$$\mathsf{noR}(n_1 + n_2) \overset{\text{def}}{=} \mathsf{noR}(n_1) + \mathsf{noR}(n_2) \qquad \mathsf{noR}(\alpha.n) \overset{\text{def}}{=} \alpha.\mathsf{noR}(n). \qquad \square$$

Lemma 4.5. *If m is a syntactically deterministic monitor that is sound and complete for φ, then $noR(m)$ is also a sound and complete monitor for φ.* □

The next step towards proving Prop. 4.7 is that of synthesising formulae from any recursion-free syntactically deterministic monitor, which can be described by the following grammar.

Definition 4.6 (Recursion-free Deterministic Monitors).

$$m, n \in \textsc{FMon} \ ::= \ \mathsf{no} \ \ | \ \mathsf{yes} \ \ | \ \textstyle\sum_{\alpha \in A} \alpha.m_\alpha . \qquad \square$$

We now show how to convert any recursion-free monitor m into an HML formula $\mathsf{f}(m)$. We then argue that a *reactive* m monitors soundly and completely for $\mathsf{f}(m)$.

Definition 4.7. The synthesis function $\mathsf{f}(-) : \textsc{FMon} \to \text{HML}$ is defined as follows:

$$\mathsf{f}(\mathsf{yes}) = \mathsf{tt} \qquad \mathsf{f}(\mathsf{no}) = \mathsf{ff} \qquad \mathsf{f}(\textstyle\sum_{\alpha \in A} \alpha.m_\alpha) = \bigwedge_{\alpha \in A} [\alpha]\mathsf{f}(m_\alpha) \qquad \square$$

Lemma 4.6. *Every* reactive *monitor $m \in$ FMon is a sound and complete monitor for $f(m)$.* □

We are now in a position to prove the expressive maximality of HML from Def. 4.3.

Proposition 4.7 (Maximality for HML). *For each $\varphi \in$ recHML, if φ is complete-monitorable, then there exists some $\psi \in$ HML such that $[\![\varphi]\!]_L = [\![\psi]\!]_L$.*

Proof. From the results in Sec. 3 and Lem. 4.5, each complete-monitorable $\varphi \in$ recHML has a recursion-free deterministic monitor m that is sound and complete for it. By Lem. 4.6, m is sound and complete for $\mathsf{f}(m)$ as well which is in HML. Prop. 4.1 thus yields $[\![\varphi]\!]_{\mathrm{L}} = [\![\mathsf{f}(m)]\!]_{\mathrm{L}}$ as required. □

The proof of Prop. 4.7 is constructive. We are also able to prove (albeit in a non-constructive manner) an even stronger result (Thm. 4.8) with respect to complete monitoring for any *arbitrary* logic defined over traces. This increases the importance of the fragment identified in Def. 4.3 for the linear-time interpretation.

Theorem 4.8. *Let m be a monitor from a monitoring system with the following two properties:*

(1) verdicts are irrevocable, that is, if m accepts (respectively, rejects) a finite trace s, then it accepts (respectively, rejects) all its extensions, and
(2) m accepts (respectively, rejects) a trace t if, and only if, it accepts (respectively, rejects) some finite prefix s of t.

For any property φ with a trace interpretation (not necessarily syntactically represented using recHML), if m is sound and complete for φ then φ can be expressed via the syntactic fragment HML of Def. 4.3. □

4.2 Tightly-Complete Monitors

The sound and complete monitoring studied in Sec. 4.1 does not specify *when* a monitor should reach a verdict while it analyses a trace, as illustrated by the following example.

Example 4.3. Assume $\text{Act} = \{a, b\}$ and consider the formula $\varphi = \langle a\rangle\langle a\rangle\mathsf{ff}$, which is equivalent to ff. Following Def. 4.4, the synthesised monitor for φ is $m = a.(a.\mathsf{no} + b.\mathsf{no}) + b.\mathsf{no}$. After at most two consecutive actions, m will definitely reject, and therefore it correctly rejects all traces. However, a more *"efficient"* correct monitor for φ is no, which rejects immediately. □

A finite trace for which every extension violates (*resp.,* satisfies) a property φ is often called a *bad prefix* (*resp.,* a *good prefix*) for φ [Alpern and Schneider 1985; Bauer et al. 2010; Pnueli and Zaks 2006]; good/bad prefixes provide sufficient finite information for acceptance/rejection.

Example 4.4. $[A]\mathsf{tt}$ is equivalent to tt, and thus ε is a good prefix for it. But $\mathsf{m}([A]\mathsf{tt})$ from Def. 4.4 would first need to observe one action before accepting. Similarly, $[\text{Act}]\mathsf{ff}$ is equivalent to ff and ε is a valid bad prefix. Yet the synthesised monitor only rejects after observing one action. □

Although the monitors synthesised in Sec. 4.1 are complete, there may be a delay from the moment a good/bad prefix is seen to the point when a verdict is reached. This observation does not affect monitor completeness: the assurance that the stream of events is *infinite* guarantees that any delay in reporting a verdict will not affect the formula's monitorability. However, it may be important for a monitor to report a verdict as soon as it gathers sufficient information to do so.

Definition 4.8. A monitor m is *tight* when, for every $s \in \text{Act}^*$, if m rejects (*resp.,* accepts) st for every $t \in \text{Trc}$, then $m \overset{s}{\Longrightarrow} \mathsf{no}$ (*resp.,* $m \overset{s}{\Longrightarrow} \mathsf{yes}$). □

Although, as Example 4.3 demonstrates, Def. 4.4 does not always yield tight monitors we can identify a fragment of HML for which it does.

Definition 4.9. A *slim* formula is defined by the following grammar:

$$\varphi \ ::= \ \mathsf{tt} \quad | \ \mathsf{ff} \quad | \ \textstyle\bigwedge_{\alpha\in B}[\alpha]\varphi_\alpha \quad | \ \textstyle\bigvee_{\alpha\in D}\langle\alpha\rangle\varphi_\alpha,$$

where $B, D \neq \emptyset$, $\forall\alpha \in B.\varphi_\alpha \neq \mathsf{tt}$, $\forall\alpha \in D.\varphi_\alpha \neq \mathsf{ff}$, either $B \neq \text{Act}$ or $\exists\alpha \in B.\varphi_\alpha \neq \mathsf{ff}$, and either $D \neq \text{Act}$ or $\exists\alpha \in D.\varphi_\alpha \neq \mathsf{tt}$. □

All slim formulae are HML formulae. However, the conditions imposed on their syntax exclude redundancies that yield non-tight monitors. We proceed to show that if φ is slim, then $\mathsf{m}(\varphi)$ is tight. To this end, we prove a lemma showing the absence of redundancy in slim formulae.

Lemma 4.9. *If $\varphi \in$ HML is slim and $[\![\varphi]\!]_L = \emptyset$ (resp., $[\![\varphi]\!]_L = \text{Trc}$), then $\varphi = \mathsf{ff}$ (resp., $\varphi = \mathsf{tt}$).* □

Lemma 4.10. *If φ is a slim HML formula, then $m(\varphi)$ is tight.*

Proof. By Prop. 4.3, $t \notin [\![\varphi]\!]_L$ implies that there is a finite prefix s of t such that $m(\varphi) \overset{s}{\Longrightarrow} \mathsf{no}$. We prove by induction on $s$that if $\forall t.\, \mathbf{rej}(m, st)$, then $m \overset{s}{\Longrightarrow} \mathsf{no}$ (the case for acceptance is symmetric). □

We can transform every HML formula into an equivalent slim formula. This transformation is based on a set of rewrite rules of the form $\varphi \Rrightarrow_L \psi$, given in Fig. 4, that allows us to iteratively replace the formula on the left-hand side with that on the right-hand side.

Lemma 4.11. *$\varphi \Rrightarrow_L \psi$ implies $[\![\varphi]\!]_L = [\![\psi]\!]_L$ and $l(\varphi) > l(\psi)$* □

Proposition 4.12 (HML normalisation). *For every formula $\varphi \in$ HML, there exists $k \leq l(\varphi)$ such that $\varphi = \varphi_0 \Rrightarrow_L \varphi_1 \Rrightarrow_L \ldots \Rrightarrow_L \varphi_k = \psi$ where ψ is slim and $[\![\varphi]\!]_L = [\![\psi]\!]_L$.* □

$$[\textsc{Act}]\mathsf{ff} \Rrightarrow_L \mathsf{ff} \qquad \langle\textsc{Act}\rangle\mathsf{tt} \Rrightarrow_L \mathsf{tt} \qquad \langle\alpha\rangle\mathsf{ff} \Rrightarrow_L \mathsf{ff} \qquad [\alpha]\mathsf{tt} \Rrightarrow_L \mathsf{tt} \tag{1}$$

$$\mathsf{tt} \wedge \varphi \Rrightarrow_L \varphi \qquad \mathsf{ff} \wedge \varphi \Rrightarrow_L \mathsf{ff} \qquad \mathsf{ff} \vee \varphi \Rrightarrow_L \varphi \qquad \mathsf{tt} \vee \varphi \Rrightarrow_L \mathsf{tt} \tag{2}$$

$$\bigwedge_{\alpha\in A}[\alpha]\varphi_\alpha \wedge \bigwedge_{\alpha\in B}[\alpha]\psi_\alpha \Rrightarrow_L \bigwedge_{\alpha\in A\cap B}[\alpha](\varphi_\alpha \wedge \psi_\alpha) \wedge \bigwedge_{\alpha\in A\setminus B}[\alpha]\varphi_\alpha \wedge \bigwedge_{\alpha\in B\setminus A}[\alpha]\psi_\alpha \tag{3}$$

$$\bigwedge_{\alpha\in A}[\alpha]\varphi_\alpha \vee \bigwedge_{\alpha\in B}[\alpha]\psi_\alpha \Rrightarrow_L \begin{cases} \bigwedge_{\alpha\in A\cap B}[\alpha](\varphi_\alpha \vee \psi_\alpha) & \text{if } A\cap B \neq \emptyset \\ \mathsf{tt} & \text{otherwise} \end{cases} \tag{4}$$

$$\bigvee_{\alpha\in A}\langle\alpha\rangle\varphi_\alpha \vee \bigvee_{\alpha\in B}\langle\alpha\rangle\psi_\alpha \Rrightarrow_L \bigvee_{\alpha\in A\cap B}\langle\alpha\rangle(\varphi_\alpha \vee \psi_\alpha) \vee \bigvee_{\alpha\in A\setminus B}\langle\alpha\rangle\varphi_\alpha \vee \bigvee_{\alpha\in B\setminus A}\langle\alpha\rangle\psi_\alpha \tag{5}$$

$$\bigvee_{\alpha\in A}\langle\alpha\rangle\varphi_\alpha \wedge \bigvee_{\alpha\in B}\langle\alpha\rangle\psi_\alpha \Rrightarrow_L \begin{cases} \bigvee_{\alpha\in A\cap B}\langle\alpha\rangle(\varphi_\alpha \wedge \psi_\alpha) & \text{if } A\cap B \neq \emptyset \\ \mathsf{ff} & \text{otherwise} \end{cases} \tag{6}$$

$$\bigwedge_{\alpha\in A}[\alpha]\varphi_\alpha \wedge \bigvee_{\alpha\in B}\langle\alpha\rangle\psi_\alpha \Rrightarrow_L \bigvee_{\alpha\in A\cap B}\langle\alpha\rangle(\varphi_\alpha \wedge \psi_\alpha) \vee \bigvee_{\alpha\in B\setminus A}\langle\alpha\rangle\psi_\alpha \tag{7}$$

$$\bigwedge_{\alpha\in A}[\alpha]\varphi_\alpha \vee \bigvee_{\alpha\in B}\langle\alpha\rangle\psi_\alpha \Rrightarrow_L \bigwedge_{\alpha\in A\cap B}[\alpha](\varphi_\alpha \wedge \psi_\alpha) \wedge \bigwedge_{\alpha\in A\setminus B}[\alpha]\varphi_\alpha \tag{8}$$

Fig. 4. HML rewrite rules where $A, B \subseteq \textsc{Act}$. $\Rrightarrow_L$ is the smallest binary relation on HML that satisfies the rules above and is closed with respect to HML contexts.

Example 4.5. Assume $\textsc{Act} = \{a, b\}$ and consider the *non*-slim HML formula $\varphi = \langle a\rangle\langle a\rangle\mathsf{ff} \wedge [b]\mathsf{ff}$. The synthesised monitor $\mathsf{m}(\varphi) = (a.(a.\mathsf{no} + b.\mathsf{no}) + b.\mathsf{no}) \otimes (b.\mathsf{no} + a.\mathsf{no})$ is *not* tight. However, we can apply the transformations based on the given equivalences to obtain an equivalent slim formula thus: $\langle a\rangle\langle a\rangle\mathsf{ff} \wedge [b]\mathsf{ff} \Rrightarrow_L \langle a\rangle\langle a\rangle\mathsf{ff} \Rrightarrow_L \langle a\rangle\mathsf{ff} \Rrightarrow_L \mathsf{ff}$. □

4.3 Partially-Complete Monitors

As opposed to the branching-time semantics of recHML, where only properties that are semantically equivalent to tt and ff have sound and complete monitors [Francalanza et al. 2017b], the linear-time semantics permits a far richer class of complete-monitorable properties, namely HML. By some measures, however, this monitorable fragment is still quite restrictive. For example, whereas the property "*initialise* occurs within the first ten actions" can be expressed in terms of HML, the property "*initialise* eventually occurs"—which can be expressed using least fixpoints—*cannot*. In fact, although the latter property *cannot* be monitored for in a complete manner, it can be monitored completely for satisfaction. In this section, we relax the notion of monitorability to *partial-completeness*, which only requires a monitor to be either violation- or satisfaction-complete.

Definition 4.10. A formula $\varphi \in$ recHML is *monitorable for satisfaction* (*resp., for violation*) iff there exists a monitor m that is a sound and satisfaction-complete (*resp.*, and violation-complete) monitor for φ. It is *partially-monitorable* when it is monitorable for satisfaction or for violation. □

We can extend these definitions to fragments of recHML in a similar way to that in Def. 4.2. Here, the trade-off between the guarantees we expect from monitors and the monitorable specifications is clear: for the linear-time interpretation, recursion can be traded for partial-completeness, while no such option exists for branching-time. We can extend the observations of Sec. 4.1 to the context of partial monitorability.

Proposition 4.13. *If m is sound and satisfaction-complete (resp., violation-complete) for φ, then*

(1) $m \simeq_{ver} n$ implies n is sound and satisfaction-complete (resp., violation-complete) for φ.
(2) If for all $v \in \{yes, no\}$, $n \overset{s}{\Longrightarrow} v$ implies $m \overset{s}{\Longrightarrow} v$, then n is sound for φ.
(3) $m \simeq_{acc} n$ (resp., $m \simeq_{rej} n$) implies n is satisfaction-complete (resp., violation-complete) for φ.
(4) m is sound and satisfaction-complete (resp., violation-complete) for φ' implies $[\![\varphi]\!]_L = [\![\varphi']\!]_L$. □

Example 4.6. Let $\textsc{Act} = \{a, b, c\}$ and $\varphi = \max X.([b]\mathsf{ff} \wedge [\{a,c\}]X) \vee \min Y.(\langle c\rangle\mathsf{tt} \vee [\{a,b\}]Y)$, which is satisfied by traces of the form $(a+c)^\omega + \big((a+b)^* c(a+b+c)^\omega\big)$, *i.e.*, traces where either b does *not* appear, or c *does* appear. We show that φ is not partially-monitorable. For if there was some m that is sound and satisfaction-complete for φ, it should accept a^ω; this means that m must reach yes after analysing a^k for some $k \geq 0$. In this case, the trace $a^k b^\omega$, which does not satisfy φ, must also be accepted by m, resulting in a contradiction. If, on the other hand, there was some m that is sound and violation-complete for φ, then it should reject b^ω. Again, m must reach no after b^k for some $k \geq 0$, but $b^k c^\omega$ satisfies φ. Therefore, φ *cannot* be partially monitorable. □

For partial monitorability, we can identify two fragments of recHML, namely minHML, which is monitorable for satisfaction, and maxHML, which is monitorable for violation.

Definition 4.11 (MAX and MIN Fragments of recHML)**.** The greatest-fixed-point and least-fixed point fragments of recHML are, respectively, defined as:

$$\varphi, \psi \in \textsc{max}\text{HML} ::= \mathsf{tt} \mid \mathsf{ff} \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \langle A\rangle\varphi \mid [A]\varphi \mid \max X.\varphi$$
$$\varphi, \psi \in \textsc{min}\text{HML} ::= \mathsf{tt} \mid \mathsf{ff} \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \langle A\rangle\varphi \mid [A]\varphi \mid \min X.\varphi$$ □

Both maxHML and minHML are extensions of HML. We can extend the monitor synthesis from Def. 4.4 to these fragments by using the recursion that is available for monitors.

Definition 4.12 (Monitor Synthesis)**.** The monitor synthesis for maxHML and minHML results by simply extending the definition of $\mathsf{m}(-)$ from Def. 4.4 with the cases for the respective fixed-point of each fragment: $\mathsf{m}(\max X.\varphi) = \mathsf{m}(\min X.\varphi) = \mathsf{rec}\, x.\mathsf{m}(\varphi)$ and $\mathsf{m}(X) = x$. □

We observe that the extended monitor synthesis function still produces reactive monitors. We also show the first important result of this subsection, namely that Def. 4.12 yields the required witness monitors to prove that the syntactic fragment maxHML ∪ minHML is partially-monitorable.

Proposition 4.14. *For every $\varphi \in$ maxHML ∪ minHML, $\mathsf{m}(\varphi)$ is reactive.* □

Proposition 4.15. *For every $\varphi \in$ maxHML, $\mathsf{m}(\varphi)$ is a sound and violation-complete monitor for φ. For every $\varphi \in$ minHML, $\mathsf{m}(\varphi)$ is a sound and satisfaction-complete monitor for φ.* □

As in the case of Sec. 4.1, we now turn our attention to the maximality of the syntactic fragment maxHML ∪ minHML for partial-monitorability. Particularly, we can define two formula synthesis functions that produce partially monitorable formulae from monitors: one maps monitors to formulae in maxHML, and the other one to formulae in minHML. Depending on the fragment, we then show that if m is mapped to φ, then m is sound and violation-complete, or satisfaction-complete *resp.*, for φ. Here we only present the synthesis function for maxHML; the case for minHML is dual.

Definition 4.13 (maxHML Formula Synthesis)**.**

$$\mathsf{f}(\mathsf{no}) = \mathsf{ff} \qquad \mathsf{f}(\mathsf{end}) = \mathsf{f}(\mathsf{yes}) = \mathsf{tt} \qquad \mathsf{f}(x) = X \qquad \mathsf{f}(\mathsf{rec}\, X.m) = \max X.\mathsf{f}(m)$$
$$\mathsf{f}(m+n) = f(m) \wedge \mathsf{f}(n) \qquad \mathsf{f}(m \otimes n) = \mathsf{f}(m) \wedge \mathsf{f}(n) \qquad \mathsf{f}(m \oplus n) = \mathsf{f}(m) \vee \mathsf{f}(n) \qquad \mathsf{f}(\alpha.m) = [\alpha]\mathsf{f}(m)$$ □

Example 4.7. Let $m = a.b.\mathsf{no} + a.a.\mathsf{yes}$. Then, $\mathsf{f}(m) = [a][b]\mathsf{ff} \wedge [a][a]\mathsf{tt}$ (which is equivalent to just $[a][b]\mathsf{ff}$). The monitor m rejects traces of the form abt which are *exactly* all the traces violating $\mathsf{f}(m)$. Thus m is sound and violation-complete for $\mathsf{f}(m)$. □

Note that $\mathsf{f}(m) \in$ MAXHML, for any m. However, when we apply the formula synthesis function from Def. 4.13 to a consistent monitor m to generate a formula φ, and then apply the monitor synthesis from Def. 4.12 to φ, we will generate a monitor that has similar parts to m, but it will be somewhat different due to the asymmetry of the *resp.*, syntheses. For example, for $\text{ACT} = \{a, b\}$, $\mathsf{f}(a.\mathsf{no} + b.\mathsf{yes}) = [a]\mathsf{ff} \wedge [b]\mathsf{tt}$, and $\mathsf{m}([a]\mathsf{ff} \wedge [b]\mathsf{tt}) = (a.\mathsf{no} + b.\mathsf{yes}) \otimes (b.\mathsf{yes} + a.\mathsf{yes})$. The following lemma allows us to abstract from these discrepancies, thereby enabling the proof of Prop. 4.17.

Lemma 4.16. *$m(f(m))$ rejects the same traces as m.* □

Proposition 4.17. *If m is consistent, then m is a sound and violation-complete monitor for $f(m)$.*

PROOF. From Lem. 4.16, $\mathsf{m}(\mathsf{f}(m))$ rejects the same traces as m, and therefore, by Props. 4.13 and 4.15, m is violation-complete for $\mathsf{f}(m)$. Since m rejects the same traces as $\mathsf{m}(\mathsf{f}(m))$, if m rejects a trace t, then $t \notin [\![\varphi]\!]_{\mathrm{L}}$. Since m is consistent, if m accepts a trace t, then it does not reject t, and because m rejects the same traces as $\mathsf{m}(\mathsf{f}(m))$, $\mathsf{m}(\mathsf{f}(m))$ does not reject t either. Since $\mathsf{m}(\mathsf{f}(m))$ is also violation-complete by Prop. 4.15, this yields that $t \in [\![\varphi]\!]_{\mathrm{L}}$. Therefore, m is also sound for φ. □

The following proposition tells us that, up to logical equivalence, MAXHML is the largest fragment of RECHML that is monitorable for violation. Dually, MINHML is the largest fragment of RECHML that is monitorable for satisfaction.

Proposition 4.18. *If a formula $\varphi \in$ RECHML has a sound and violation-complete monitor over infinite traces, then it is equivalent to a formula $\psi \in$ MAXHML over infinite traces.*

PROOF. Let m be a sound and violation-complete monitor for φ and let $\psi = \mathsf{f}(m) \in$ MAXHML be the witness formula. Since m is sound for φ, it must be consistent, and by Prop. 4.17, m is a sound and violation-complete monitor for $\mathsf{f}(m)$. Therefore, by Prop. 4.13, $[\![\varphi]\!]_{\mathrm{L}} = [\![\mathsf{f}(m)]\!]_{\mathrm{L}}$. □

Remark. Thm. 4.8 demonstrates that HML can express any property of infinite traces that has a complete monitor in any monitoring system, assuming that verdicts remain irrevocable. Unfortunately, this result cannot be replicated for partial completeness. For instance, let $L \subseteq (\text{ACT} \setminus \{c\})^*$ be a non-regular language, where $c \in \text{ACT}$ is some distinguished action, and $L_c = \{\, sct \mid s \in L \text{ and } t \in \text{ACT}^{\omega} \,\}$. If L_c could be expressed in MINHML, then there would be a sound and satisfaction-complete monitor for L_c, and by a straightforward use of Prop. 3.6, we could construct a finite automaton that recognizes L, which contradicts the assumption that L is non-regular. Yet, we could imagine appropriate choices for L and monitoring systems in which L_c is monitorable. For instance, suppose that monitors are described using pushdown automata and let L contain exactly the finite words on $\{0, 1\}$ that have the same number of occurrences of 0 and of 1. □

4.4 Tightly-Complete Monitors for Recursion

To synthesise a tight monitor for a formula φ of MAXHML (or MINHML), one can synthesise a parallel monitor $m(\varphi)$, then, using the methods of Subsection 3.3, turn $m(\varphi)$ into a verdict-equivalent deterministic regular monitor, and, finally, consecutively replace instances of $\sum_{\alpha \in \text{ACT}} \alpha.\mathsf{no}$ and $\mathsf{rec}\, x.\mathsf{no}$ by no and instances of $\sum_{\alpha \in \text{ACT}} \alpha.\mathsf{yes}$ and $\mathsf{rec}\, x.\mathsf{yes}$ by yes. The resulting monitor is tight.

Lemma 4.19. *Let m be a deterministic regular monitor, where $\sum_{\alpha \in \text{ACT}} \alpha.no$, $rec\, x.no$, $\sum_{\alpha \in \text{ACT}} \alpha.yes$, and $rec\, x.yes$ do not occur as submonitors. Then, m is tight.* □

We would like to be able to apply a convenient method to process the formula or the monitor, so that right after the monitor synthesis we could produce a tight monitor. However, as we will see, a more reasonable monitor synthesis function that produces tight monitors is unlikely, as one could use it to solve the satisfiability problem for MAXHML— by checking whether a produced monitor for the formula immediately evaluates to no (or to yes, for its negation), — which is PSPACE-complete.

Proposition 4.20. *For $|\textsc{Act}| \geq 2$, the satisfiability problem for* MAXHML *is PSPACE-complete.*

Proof. Satisfiability for RECHML (and therefore for MAXHML as well) is known to be in PSPACE [Vardi 1988a]. That satisfiability for MAXHML is PSPACE-hard results from the observation that MAXHML with at least two actions can encode the 1-variable, diamond-free fragment of $D \oplus_{\subseteq} D4$, which is PSPACE-complete [Achilleos 2016]. □

Remark. For singleton $\textsc{Act} = \{a\}$, RECHML-satisfiability is a lot simpler, as there is only one trace, a^{ω}. Therefore, satisfiability for MAXHML can be reduced to model-checking on a^{ω}. A more direct way to solve satisfiability is to reduce the given formula by using the following straightforward rewrite rules: $\mathsf{ff} \wedge \varphi \Rrightarrow_L \mathsf{ff}$, $\mathsf{ff} \vee \varphi \Rrightarrow_L \varphi$, $\langle \alpha \rangle \mathsf{ff} \Rrightarrow_L \mathsf{ff}$, $[\alpha]\mathsf{ff} \Rrightarrow_L \mathsf{ff}$, and $\max X.\mathsf{ff} \Rrightarrow_L \mathsf{ff}$; the cases for tt are symmetric. After applying these formula simplifications, we will either reach one of tt, ff, in which case the answer to satisfiability is obvious, or we will reach a formula φ without these constants. In the latter case, we can easily see that $\mathsf{m}(\varphi)$ can never reach a verdict, and therefore it will never reject a trace, which, from Prop. 4.15, implies that $[\![\varphi]\!]_{\mathrm{L}} = \textsc{Trc} = \{\alpha^{\omega}\}$. □

5 BRANCHING-TIME MONITORABILITY

Monitorability over branching-time semantics has been examined in Aceto et al. [2017a, 2018a] and Francalanza et al. [2017b] for various frameworks. In this section we compare the results of Francalanza et al. [2017b], the closest to our setting, with those of Sec. 4. We begin by revisiting the basic definitions and results for branching-time monitorability. Then, in Sec. 5.1 and Sec. 5.2, we extend the study of monitorability to a domain that allows *both* finite and infinite traces, and conclude, in Sec. 5.3, by comparing the monitorable fragments in this domain to those in the branching-time setting.

Definition 5.1 (Branching-time Monitor Soundness and Completeness)**.**

- A monitor m is *sound* for a (closed) formula φ over processes if, *for all* $p \in \textsc{Prc}$ of every LTS, *i.e.,* a triple $\langle \textsc{Prc}, (\textsc{Act} \cup \{\tau\}), \longrightarrow \rangle$:
  - $\mathbf{rej}(m,p)$ implies $p \notin [\![\varphi]\!]_{\mathrm{B}}$;
  - $\mathbf{acc}(m,p)$ implies $p \in [\![\varphi]\!]_{\mathrm{B}}$.
- A monitor m is *violation-complete* for a formula φ over processes if *for all* $p \in \textsc{Prc}$ of every LTS, $p \notin [\![\varphi]\!]_{\mathrm{B}}$ implies $\mathbf{rej}(m,p)$. It is *satisfaction-complete* if $p \in [\![\varphi]\!]_{\mathrm{B}}$ implies $\mathbf{acc}(m,p)$. □

Remark. The LTS is often omitted when it is clear from the context. As before, a monitor m is *complete* for φ if it is violation- and satisfaction-complete for it. A rejection monitor is a monitor without the verdict yes; an acceptance monitor is one without the verdict no. □

In the branching-time setting, monitors with both yes and no verdicts are unsound for any formula, as whenever one trace leads to an acceptance and another to a rejection, one can easily construct a process that can emit both traces. As a single-verdict (uni-verdict [Francalanza et al. 2017b]) monitor can only be either satisfaction- or violation-complete for a formula (except monitors for tt and ff which can be both), one cannot hope for complete monitors for RECHML, and therefore the best one can do is to identify its fragments for which partially complete monitors exist. These are sHML and cHML, defined by the following grammars:

Definition 5.2 (Safety and Cosafety Fragments for Branching-time RECHML)**.**

$$\varphi, \psi \in \mathrm{sHML} ::= \mathsf{tt} \mid \mathsf{ff} \mid [A]\varphi \mid \varphi \wedge \psi \mid \max X.\varphi \mid X$$
$$\varphi, \psi \in \mathrm{cHML} ::= \mathsf{tt} \mid \mathsf{ff} \mid \langle A \rangle\varphi \mid \varphi \vee \psi \mid \min X.\varphi \mid X. \quad \square$$

Theorem 5.1 (Branching-time Monitorability [Francalanza et al. 2017b])**.** *For every $\varphi \in$ sHML, there is a regular rejection monitor m that is sound and violation-complete for φ. For every $\varphi \in$ cHML, there is a regular acceptance monitor m that is sound and satisfaction-complete for φ.* □

Theorem 5.2 (Maximality of sHML and cHML [Francalanza et al. 2017b])**.** *For every regular rejection monitor m, there is a formula $\varphi \in$ sHML, such that m is sound and violation-complete for φ. For every regular acceptance monitor m, there is a formula $\varphi \in$ cHML, such that m is sound and satisfaction-complete for φ.* □

One can identify two key differences between the linear-time and the branching-time semantics introduced in Sec. 2. The first and most characteristic difference is that for branching-time semantics, where formulae are interpreted over processes, a process is allowed to emit more than one trace. In other words, a process may exhibit different behaviour each time it runs, and therefore, a trace does not give the whole picture of its possible executions. By contrast, for linear-time semantics, if one observes an action or a finite trace, then there is no possibility that another one could have been exhibited instead. This allows for constructs such as parallel monitors to monitor for conjunctions and disjunctions at the same time: simply decompose the formula as the monitor synthesis function directs in Defs. 4.4 and 4.12, and let each monitor component examine the trace until a conclusion is reached. For branching-time semantics, this method does not help to monitor a conjunction for satisfaction or a disjunction for rejection, as Francalanza et al. [2017b] demonstrates.

Example 5.1. Consider $\varphi = [a]\mathsf{ff} \vee [b]\mathsf{ff} \notin$ sHML. In contrast to the linear-time setting, φ is *not* monitorable for violation under a branching-time interpretation. For assume, towards a contradiction, that there is a rejection monitor for φ. Assume an LTS with a process p that has two transitions, $p \xrightarrow{a} \mathsf{nil}$ and $p \xrightarrow{b} \mathsf{nil}$. Then, $p \notin [\![\varphi]\!]_{\mathrm{B}}$ and p can produce three possible traces: ε, a, b. If a monitor rejected one of these, say a, then it would reject p, but also process q_a that has exactly one transition, $q_a \xrightarrow{a} \mathsf{nil}$. But we observe that $q_a \in [\![\varphi]\!]_{\mathrm{B}}$, meaning that the monitor would *not* be sound for φ. The formula $[a]\mathsf{ff} \vee [b]\mathsf{ff}$ is however monitorable in a linear-time setting (Defs. 4.3 and 4.11). □

The second difference is that, in the linear-time semantics, formulae are only interpreted over *infinite* traces while, in branching-time semantics, a trace is allowed to end. Unlike the first difference, this one is not inherent to the linear- versus branching-time distinction, but it is one we have lifted from standard LTL-style semantics [Bradfield and Stirling 2001; Vardi 1988b]. Therefore, as a first step to reconcile the two semantics, we focus on this less essential difference for our logic.

5.1 The Finfinite Domain

We introduce an alternative linear-time semantics for our logic, where formulae are interpreted over traces that are allowed to be either finite or infinite. For convenience, we call these kinds of traces *finfinite* and the resulting semantics *finfinite* linear-time semantics, or just finfinite semantics. (A semantics akin to ours for a linear-time temporal logic may be found in, for instance, Schneider [1997]. Falcone et al. [2012b] define linear-time properties over finite and infinite traces, but do not consider a specific logic.) The finfinite semantics, $[\![-]\!]_{\mathrm{F}}$, is presented in Fig. 5. The set of finfinite traces is $\textsc{fTrc} = \textsc{Trc} \cup \textsc{Act}^*$ and we use $g, h \in \textsc{fTrc}$ (*resp.*, $F \subseteq \textsc{fTrc}$) to range over (*resp.*, sets of) finfinite traces.

Remark. For recHML, $\langle \textsc{Act} \rangle \varphi$ and $[\textsc{Act}]\varphi$ can be seen as the strong and weak next operators, $X\varphi$ and $\overline{X}\varphi$ from LTL [Clarke et al. 1999]. In this same setting, $[A]\varphi$ may be seen as shorthand for $\langle \overline{A} \rangle \mathsf{tt} \vee \langle A \rangle \varphi$. However the encoding does *not* work for the finfinite interpretation of Fig. 5. □

$$
\begin{array}{rcl@{\qquad}rcl}
[\![\mathsf{tt},\sigma]\!]_{\mathrm{F}} & \overset{\text{def}}{=} & \textsc{fTrc} & [\![\mathsf{ff},\sigma]\!]_{\mathrm{F}} & \overset{\text{def}}{=} & \emptyset \\
[\![\varphi_1{\vee}\varphi_2,\sigma]\!]_{\mathrm{F}} & \overset{\text{def}}{=} & [\![\varphi_1,\sigma]\!]_{\mathrm{F}} \cup [\![\varphi_2,\sigma]\!]_{\mathrm{F}} & [\![\varphi_1{\wedge}\varphi_2,\sigma]\!]_{\mathrm{F}} & \overset{\text{def}}{=} & [\![\varphi_1,\sigma]\!]_{\mathrm{F}} \cap [\![\varphi_2,\sigma]\!]_{\mathrm{F}} \\
[\![\langle A\rangle\varphi,\sigma]\!]_{\mathrm{F}} & \overset{\text{def}}{=} & \{g \mid \exists h\cdot\exists\alpha\in A\cdot g=\alpha h \ \text{ and }\ h\in[\![\varphi,\sigma]\!]_{\mathrm{F}}\} & & & \\
[\![[A]\varphi,\sigma]\!]_{\mathrm{F}} & \overset{\text{def}}{=} & \{g \mid \forall h\cdot\forall\alpha\in A\cdot t=\alpha h \ \text{ implies }\ h\in[\![\varphi,\sigma]\!]_{\mathrm{F}}\} & & & \\
[\![\mathsf{min}X.\varphi,\sigma]\!]_{\mathrm{F}} & \overset{\text{def}}{=} & \bigcap\{F \mid [\![\varphi,\sigma[X\mapsto F]]\!]_{\mathrm{F}}\subseteq F\} & & & \\
[\![\mathsf{max}X.\varphi,\sigma]\!]_{\mathrm{F}} & \overset{\text{def}}{=} & \bigcup\{F \mid F\subseteq[\![\varphi,\sigma[X\mapsto F]]\!]_{\mathrm{F}}\} & [\![X,\sigma]\!]_{\mathrm{F}} & \overset{\text{def}}{=} & \sigma(X)
\end{array}
$$

Fig. 5. Finfinite Linear-Time Semantics

The two linear-time semantics for recHML still correspond in some sense; see Lem. 5.3. In particular, formula equivalence over finfinite traces implies equivalence over infinite traces.

Lemma 5.3. *For all $\varphi \in$ recHML, $[\![\varphi]\!]_F \cap \textsc{Trc} = [\![\varphi]\!]_L$.* □

We consider the same monitoring systems of regular and parallel monitors that were introduced in Sec. 3. However, what it means for m to monitor for φ depends on the semantics that we use for the formulae: the definition used in Sec. 4 is therefore not sufficient for the finfinite domain.

Definition 5.3 (Finfinite Linear-time Monitor Soundness and Completeness)**.**

- A monitor m is *sound* for a (closed) formula φ over finfinite traces if, *for all* $g \in \textsc{fTrc}$:
  - $\mathbf{rej}(m, g)$ implies $g \notin [\![\varphi]\!]_{\mathrm{F}}$;
  - $\mathbf{acc}(m, g)$ implies $g \in [\![\varphi]\!]_{\mathrm{F}}$.
- A monitor m is *violation-complete* for a formula φ over finfinite traces if *for all* $g \in \textsc{fTrc}$, $g \notin [\![\varphi]\!]_{\mathrm{F}}$ implies $\mathbf{rej}(m, g)$. It is *satisfaction-complete* if $g \in [\![\varphi]\!]_{\mathrm{F}}$ implies $\mathbf{acc}(m, g)$. It is *complete* for a formula φ over finfinite traces if it is *both* violation- and satisfaction complete for it. □

Monitorability of formulae and logics can be adjusted to finfinite traces analogously.

5.2 Monitorability over Finfinite Traces

We now identify the complete- and partial-monitorable fragments of recHML over finfinite traces. Our first observation is that under finfinite semantics, there are no complete-monitorable formulae, except the ones equivalent to tt or ff.

Lemma 5.4. *If m is sound and complete for φ over finfinite traces, then $[\![\varphi]\!]_F = \textsc{fTrc}$ or $[\![\varphi]\!]_F = \emptyset$.* □

Remark. Lem. 5.4 holds regardless of the considered logic: due to verdict-persistence (Lem. 3.1), a logical fragment that is complete-monitorable over finfinite traces must be trivial for any logic interpreted over finfinite traces. □

The concept of tightness, as defined in Def. 4.8, does not apply for the finfinite interpretation since there is no guarantee that a finfinite trace will have a continuation. A definition of tightness might stipulate that a rejection-monitor is tight for a formula when it is guaranteed to reject any finite trace as long as the trace and all of its (finfinite) continuations violate the formula (*i.e.,* bad prefixes). However, this notion of tightness is implied by partial completeness.

Example 5.2. In contrast to the infinite trace semantics, $\langle a\rangle$tt is not monitorable for violation under finfinite semantics. For assume towards a contradiction that m is a monitor that is sound and violation-complete for $\langle a\rangle$tt. Then, m must reject the empty trace, ε, and thus all of its extensions, including $a \in [\![\langle a\rangle\mathsf{tt}]\!]_{\mathrm{F}}$, making m unsound. Similarly, $[\alpha]$ff is not monitorable for satisfaction. □

Our next goal is to characterize the expressive power of monitors in finfinite semantics. To this end, we identify the following fragments of RECHML. Only one type of modality is kept in each of these fragments. This is because, as observed in Example 5.2, the two modalities are not mutually expressive and even simple formulae using them are not monitorable for violation or satisfaction.

Definition 5.4.

$$\varphi,\psi \in \text{UNHML} ::= \mathsf{tt} \quad | \quad \mathsf{ff} \quad | \quad [A]\varphi \quad | \quad \varphi \vee \psi \quad | \quad \varphi \wedge \psi \quad | \quad \max X.\varphi \quad | \quad X, \text{ and}$$
$$\varphi,\psi \in \text{EXHML} ::= \mathsf{tt} \quad | \quad \mathsf{ff} \quad | \quad \langle A\rangle\varphi \quad | \quad \varphi \vee \psi \quad | \quad \varphi \wedge \psi \quad | \quad \min X.\varphi \quad | \quad X. \qquad \square$$

The next lemma formalises the property that formulae in UNHML denote prefix-closed sets of (finfinite) traces whereas formulae in EXHML denote suffix-closed sets of traces.

Lemma 5.5. *For all $s \in \text{ACT}^*$ and $g \in \text{FTRC}$, (i) if $\varphi \in$ UNHML and $sg \in [\![\varphi]\!]_F$, then $s \in [\![\varphi]\!]_F$; (ii) if $\varphi \in$ EXHML and $s \in [\![\varphi]\!]_F$, then $sg \in [\![\varphi]\!]_F$.* $\square$

Interestingly, for UNHML and EXHML over finfinite traces, we can use the same monitor synthesis function that we used to generate monitors for MAXHML and MINHML over infinite traces.

Proposition 5.6. *For every $\varphi \in$ UNHML, $m(\varphi)$ is sound and violation-complete for φ over finfinite traces. For every $\varphi \in$ EXHML, $m(\varphi)$ is sound and satisfaction-complete for φ over finfinite traces.* $\square$

To facilitate our comparisons between the finfinite and the branching-time interpretations of RECHML, we define the notion of trace-processes.

Definition 5.5. Process p is a *trace-process* when $p \xrightarrow{\mu} q$ and $p \xrightarrow{\mu'} q'$ implies $\mu = \mu'$, $q = q'$ and q is a trace-process. A (trace) process p *represents* a finfinite g when $p \overset{s}{\Longrightarrow}$ iff s is a prefix of g. $\square$

For a trace g, we can assume the existence of a trace-process p_g that represents g: one can construct such a trace-process p_g whereby its states are all the prefixes of g and its transitions are those of the form $s \xrightarrow{\alpha} s\alpha$, where s and $s\alpha$ are prefixes of g.

Remark. We note that, unlike for monitors, we have *not* assumed any specific syntax for processes, which can come from an arbitrary LTS. This makes it possible to represent every finfinite trace, even one without a finite representation, by a process. $\square$

Example 5.3. A process representing ab is the three-state process p, with *just* the transitions $p \xrightarrow{a} p'$ and $p' \xrightarrow{b}$ nil. A process representing a^ω is q that has exactly one transition, $q \xrightarrow{a} q$. $\square$

Lem. 5.7 shows that, for RECHML, (finfinite) traces and trace-processes are different descriptions of the same model.

Lemma 5.7. *If p represents g, then $g \in [\![\varphi]\!]_F$ iff $p \in [\![\varphi]\!]_B$.* $\square$

Coincidentally, all formulae that are monitorable for violation or satisfaction over a finfinite semantics are equivalent to sHML or cHML formulae *resp.*, from Def. 5.2. Since UNHML and EXHML syntactically subsume sHML and cHML *resp.*, they are maximally monitorable fragments of RECHML when interpreted over finfinite traces.

Proposition 5.8. *If $\varphi \in$ RECHML has a sound and violation-complete (resp., satisfaction-complete) reactive parallel monitor over finfinite traces, then there is some $\psi \in$ sHML (resp., $\psi \in$ cHML) that is equivalent to φ over finfinite traces.*

Proof. Let m be a sound and violation-complete reactive parallel monitor for φ over finfinite traces. By Prop. 3.8, there is a regular monitor n that is verdict-equivalent to m, so it is also sound and violation-complete for φ over finfinite traces. We can then obtain a single-verdict monitor n' from n that is rejection equivalent to it by swapping any yes with end. n' is thus still sound and violation-complete for φ over finfinite traces. From Thm. 5.2 there is a formula $\psi \in$ sHML, such that n is sound and violation-complete for ψ over all LTSs, including the LTS of trace-processes. Since n is sound and violation-complete for ψ on trace processes, $p_g \in [\![\psi]\!]_{\mathrm{B}}$ is equivalent to claiming that n does not reject any trace that p_g can produce. However, this is equivalent to saying that n does not reject g which, by violation-completeness, is equivalent to $g \in [\![\varphi]\!]_{\mathrm{F}}$. By Lem. 5.7, $g \in [\![\psi]\!]_{\mathrm{F}}$ iff $p_g \in [\![\psi]\!]_{\mathrm{B}}$, and the proof is complete. The case for a satisfaction-complete monitor is similar. ◻

5.3 Monitorable Formulae Across Semantics

So far, we have identified a different pair of partial-monitorable syntactic fragments for each of the three semantics that we have presented in this paper. However, as the reader may suspect from Prop. 5.8, we may be able to further restrict the syntax that we allow for our formulae, and still be able to express all monitorable formulae, and therefore, an identified maximally monitorable fragment of recHML may be equally expressive as a syntactic fragment of its own.

Here we show that *for each of the semantics that we have presented, i.e.,* over infinite traces, finfinite traces, and processes, sHML and cHML are equally expressive as the corresponding identified partially monitorable fragment. That is to say, sHML is as expressive as unHML over finfinite traces and as expressive as maxHML over infinite traces — and dually, cHML is as expressive as exHML over finfinite traces and as expressive as minHML over infinite traces.

Proposition 5.9. *For finite* Act*, if $\varphi \in$ unHML (resp., $\varphi \in$ exHML), then there is some $\psi \in$ sHML (resp., $\psi \in$ cHML) that is equivalent to φ over finfinite traces.* ◻

Proposition 5.10. *For finite* Act*, if $\varphi \in$ maxHML (resp., $\varphi \in$ minHML), then there is some $\psi \in$ sHML (resp., $\psi \in$ cHML) that is equivalent to φ over infinite traces.* ◻

The proofs of both of these propositions proceed by considering a sound and partially complete monitor for a formula in unHML, maxHML or their duals, and using the formula synthesis to find an sHML formula that is equivalent to the original formula on finfinite and infinite traces respectively.

The import of Props. 5.9 and 5.10 is that, in settings where Act is finite, logical fragment sHML∪cHML can be used to *syntactically* characterise the class of monitorable properties (for sound and partial-completeness) for all three interpretations (*i.e.,* traces, finfinite traces and processes). In spite of this felicitous (and somewhat surprising) result, one should nevertheless stress that their interpretation is still *semantically different.* In fact, the synthesised monitors presented here in Defs. 4.4 and 4.12 yield *behaviourally different* monitors to those obtained by the synthesis in Francalanza et al. [2017b]. Moreover, they can *not* be used interchangeably: Defs. 4.4 and 4.12 produce multi-verdict monitors, even when applied to the syntactic fragment sHML ∪ cHML, which makes them immediately unsound for a branching-time interpretation. In Prop. 5.11, we can however show that the monitors synthesised by the procedure of Francalanza et al. [2017b] for the sHML fragment qualify also as correct monitors for the finfinite interpretation of the logic. This means that the tools developed in Attard et al. [2017] and Attard and Francalanza [2016], which are based on the branching-time synthesis of Francalanza et al. [2017b], can be used out of the box to monitor for finfinite properties.

Proposition 5.11. *For a process p and a formula $\varphi \in$ sHML, the following are equivalent: (i) $p \in [\![\varphi]\!]_B$ and (ii) If p produces a finfinite trace g, then $g \in [\![\varphi]\!]_F$.* ◻

6 CONCLUSION

We have presented a systematic study of the monitorability of recHML, a highly expressive specification logic: we have developed results relating to its linear-time interpretation and established correspondences with previous monitorability results for the branching-time interpretation of the logic. This allows us to use existing RV tools (developed for branching-time) to monitor linear-time recHML properties. To our knowledge, this is the first study of monitorability that spans across the linear-time/branching-time spectrum. Moreover, although monitorability has been studied extensively for linear-time specifications, we are unaware of any maximality results such as those presented in Props. 4.7, 4.18 and 5.8 to 5.10 and Thm. 4.8.

Concretely, in Sec. 3, we introduce parallel monitors and we gave a way to construct a deterministic regular monitor (introduced in Aceto et al. [2017b] and Francalanza et al. [2017b]) from a parallel one, establishing that the two monitoring frameworks are equivalent with respect to the properties they can monitor. In Sec. 4, we give a natural monitor synthesis from three fragments of recHML to parallel monitors, and establish that the resulting monitors satisfy the requirement of soundness and a version of the requirement for completeness. For complete monitors, we identify the requirement of tightness and show how one can satisfy it. In Sec. 5, we see how these findings apply in the intermediate finfinite setting, and we establish that sHML has the same expressive power as the respective maximal monitorable fragments of recHML in the finfinite and infinite-trace settings.

Multiple Ways to Monitor. These results show that there is more than one way to monitor for a property φ that is monitorable for violation. If φ is already in sHML, or if we want to make the effort to write the property as an sHML formula, we can use the monitor synthesis in Francalanza et al. [2017b] to synthesise a single-verdict, sound and violation-complete regular monitor for φ that will work in all (infinite-trace, finfinite, and branching-time) semantics. Alternatively, if we are interested in the linear-time domain (for either infinite or finfinite traces), we can synthesise a parallel monitor with the synthesis function from Def. 4.12, hoping that the possibly dual-verdict monitor may occasionally report the satisfaction of the formula, providing us with more information. In the latter case, we may choose to deploy the parallel monitor as is, or use the construction from Prop. 3.8 to obtain a verdict-equivalent regular monitor. An advantage of using the parallel monitor is that it can be significantly more concise than a regular monitor, at least at the early stages of the computation. An advantage of using a regular monitor is that it is guaranteed to be finite state (Prop. 3.2). Furthermore, regular monitors can be determinized and then minimized (see Prop. 3.11 and Aceto et al. [2016]), making their implementation more straightforward. Therefore, one can think of maxHML as a high-level specification language for properties that are monitorable for violation in the linear-time setting. From maxHML, we can generate parallel monitors that can then be compiled into (deterministic, minimized) regular monitors that can be implemented and deployed to monitor the system. On the other hand, sHML can be thought of as a lower-level language that is closer to regular monitors and can allow for better fine-tuning of the monitor's behaviour, and avoids the cost of constructing a regular monitor.

Future Work. We are interested in a detailed taxonomy and comparison of different notions of monitorability, and this work is a first step in that direction. Additionally, in Aceto et al. [2018a], the authors examine how the set of monitorable properties can be extended by encoding additional information into the trace that describes a system execution. Noticeably, their framework allows for the interaction of multiple verification methods, and this is an approach we would like to explore for our own framework.

Related Work on Runtime Verification. RV has been applied in the computer-aided verification of complex programs and models written in a variety of high-level languages. For example, RV has been used in the verification of properties written in an extension of PSL and SVA over SystemC models in Tabakov et al. [2012] (but see Pnueli and Zaks [2006] and references in Tabakov et al. [2012] for earlier work on monitor synthesis for PSL). Like we do in this paper, Tabakov *et al.* argue for the algorithmic generation of "correct" monitors from properties. However, their focus is on an experimental study of monitor-generation procedures that offer the best performance in terms of runtime overhead at simulation time. In order to do so, they employ the CHIMP tool [Dutta et al. 2014] to generate monitors (represented as DFAs) from LTL properties using a number of workflows that take into account various options regarding state minimization, alphabet representation, alphabet minimization and the representation of the transition function of the monitor.

Diagnosability. It is worth mentioning here work on *diagnosability*, *e.g.*, Bertrand et al. [2014]; Sampath et al. [1995]. Diagnosability is a similar notion to the one of monitorability. What is different is that, for diagnosability one knows a model of the system, and then, by observing the visible events of a system run, infers whether an unobservable fault event has occurred during this run. A further goal is to diagnose the kind of fault event that has occurred. Typically, the detection and diagnosis of fault events is performed by a diagnoser, which is synthesised from the model of the system. Although RV and diagnosability appear, at first glance, to work in different ways, one can view diagnosability as the runtime monitoring of a set of trace-properties (the occurrence of different types of fault events), using information about the system's branching structure, in a framework that considers unobservable events — as in Aceto et al. [2017a]; Francalanza et al. [2017b]. We feel that there is significant potential in addressing the two areas in a more unified manner. This is an interesting avenue for future research.

Related Work on Specification logics. recHML is a multi-modal variant of the μ-calculus that is interpreted over edge-labelled LTSs rather than node-labelled ones. The distinction is mainly a question of presentation; how to go between the two types of models is discussed by De Nicola and Vaandrager [1990]. The μ-calculus itself is a logic which subsumes CTL, CTL*, LTL, as well as more exotic variations thereof. Its links to automata theory are well established [Wilke 2001] and can be used in the implementation of verification tools. This makes the μ-calculus well suited for foundational research on verification, even though logics with more intuitive syntax may appeal to practitioners. recHML over traces is similar to the linear-time μ-calculus. The main difference is that in the linear-time μ-calculus, which is usually interpreted over infinite traces, it is common to have only one successor-modality: the difference between $[\alpha]$ and $\langle\alpha\rangle$ only manifests itself over finite traces. Here we have chosen to keep the two modalities, to enable the syntactic comparison between branching-time and linear-time monitorability. From an implementation point of view, recHML formulae, like those in the linear time μ-calculus, can be represented by weak automata [Lange 2005], which benefit from lower-complexity decision procedures than the more general parity automata, which are necessary to capture the expressiveness of the μ-calculus in a branching-time setting. Note, however, that, as shown in Markey and Schnoebelen [2006], the μ-calculus model-checking problem over paths of the form s^ω is, surprisingly, as hard as the general model-checking problem for that logic.

In the context of RV, many-valued logics [Barringer et al. 2004; Bauer et al. 2010; d'Angelo et al. 2005; Drusinsky 2003] have also emerged as a way to reconcile the infinitary semantics of, for example, LTL specifications with the finite observations of a monitor. Our concept of monitor can itself also be understood as a logic with three-valued semantics, consisting of accepted traces, rejected traces and traces on which the monitor remains indecisive. Conversely, these many-valued logics can also be seen as describing monitor behaviour, albeit without an operational semantics as in our case. Our parallel monitors are reminiscent of alternating automata. The use of alternating automata for RV is not new: Finkbeiner and Sipma [2004] propose this for the verification of their finite-trace semantics for LTL. The main difference in their approach is that their semantics is not suffix closed: for whether "infinitely often a" holds in a finite trace according to their semantics will depend on whether a holds in the last position. In contrast, our verdicts are irrevocable, so a sound monitor for "infinitely often a" in our setting will never reach a verdict.

Related Work on Monitorability. The question of exactly which specifications can be verified at runtime is very natural in the RV context. It is perhaps surprising that there is no consensus on what exactly it means for a specification to be monitorable.

The class Π_1^0 of the arithmetic hierarchy — the class of co-recursively enumerable safety properties — was proposed as the set of monitorable properties by Viswanathan and Kim [2004]. It seems that our notion of partial monitorability matches well with this classical definition. In this sense, partial monitorability could be seen as an operational account of Viswanathan and Kim's monitorability. On the other hand, Pnueli and Zaks [2006] and Bauer et al. [2011] propose a definition of monitorability that includes more properties: roughly, they call a property monitorable if every prefix has a finite continuation of which either all infinite continuations are in the property, or none is. This means that a monitor, although it does not necessarily ever reach a verdict, can never give up hope of reaching a verdict. Our definitions of monitorability appear to be stronger. For example, specifications such as "never `error` and eventually `success`" is monitorable according to Pnueli and Zaks [2006] and Bauer et al. [2011] but not according to our notions of monitorability and partial monitorability.

Diekert and Leucker have studied monitorability in a topological setting in Diekert and Leucker [2014], where they show that all ω-regular languages that are deterministic and co-deterministic are monitorable. Using their topological framework, they also establish that some deterministic liveness properties, such as "infinitely many a's", cannot be written as a countable union of monitorable languages. Diekert et al. [2015] discuss monitor constructions for deterministic ω-regular languages. They isolate a collection of deterministic ω-regular languages that properly includes all the languages that are deterministic and codeterministic, and for which one can construct accepting monitors. These classical definitions of monitorability are independent of how a monitor might be implemented. Conversely, implementations of LTL monitors [Giannakopoulou and Havelund 2001; Havelund and Rosu 2002] do not seem to refer to the concept of monitorability at all. In line with previous work [Francalanza et al. 2017b], our operational approach bridges this gap by defining *what* can be monitored explicitly in terms of *how* specifications are monitored.

ACKNOWLEDGMENTS

The authors thank Orna Kupferman and Moshe Vardi for clarifying remarks about guarded formulae and pointers to the literature. We thank the anonymous reviewers for their thorough reading of our paper and their insightful comments.

This research was partially supported by the projects "TheoFoMon: Theoretical Foundations for Monitorability" (grant number: 163406-051) and "Epistemic Logic for Distributed Runtime Monitoring" (grant number: 184940-051) of the Icelandic Research Fund, by the BMBF project "Aramis II" (project number: 01IS160253) and the EPSRC project "Solving parity games in theory and practice" (project number: EP/P020909/1).

REFERENCES

Luca Aceto, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfsdóttir. 2017a. Monitoring for silent actions. In *FSTTCS (LIPIcs)*, Satya Lokam and R. Ramanujam (Eds.), Vol. 93. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 7:1–7:14.

Luca Aceto, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfsdóttir. 2018a. A Framework for Parametrized Monitorability. In *FOSSACS (Lecture Notes in Computer Science)*, Vol. 10803. Springer, 203–220.

Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Sævar Örn Kjartansson. 2016. Determinizing Monitors for HML with Recursion. *CoRR* abs/1611.10212 (2016). arXiv:1611.10212 http://arxiv.org/abs/1611.10212

Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Sævar Örn Kjartansson. 2017b. On the Complexity of Determinizing Monitors. In *Implementation and Application of Automata*. Springer International Publishing, 1–13. https://doi.org/10.1007/978-3-319-60134-2_1

Luca Aceto, Ian Cassar, Adrian Francalanza, and Anna Ingólfsdóttir. 2018b. On Runtime Enforcement via Suppressions. In *CONCUR (LIPIcs)*, Vol. 118. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 34:1–34:17. https://doi.org/10.4230/LIPIcs.CONCUR.2018.34

Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. 2007. *Reactive Systems: Modelling, Specification and Verification.* Cambridge Univ. Press, New York, NY, USA.

Antonis Achilleos. 2016. Modal Logics with Hard Diamond-Free Fragments. In *Logical Foundations of Computer Science*, Sergei Artemov and Anil Nerode (Eds.). Springer International Publishing, Cham, 1–13.

Bowen Alpern and Fred B. Schneider. 1985. Defining Liveness. *Inform. Process. Lett.* 21, 4 (1985), 181–185. https://doi.org/10.1016/0020-0190(85)90056-0

Duncan Paul Attard, Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingolfsdottir. 2017. *Behavioural Types: from Theory to Tools.* River Publishers, Chapter A Runtime Monitoring Tool for Actor-Based Systems, 49–74.

Duncan Paul Attard and Adrian Francalanza. 2016. *A Monitoring Tool for a Branching-Time Logic.* Springer International Publishing, Cham, 473–481. https://doi.org/10.1007/978-3-319-46982-9_31

Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. 2008. *Principles of model checking.* MIT press.

Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. 2004. Rule-based runtime verification. In *VMCAI*, Vol. 2937. Springer, 44–57.

Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. *Introduction to Runtime Verification.* Springer, 1–33.

Andreas Bauer, Martin Leucker, and Christian Schallhart. 2010. Comparing LTL semantics for runtime verification. *Logic and Computation* 20, 3 (2010), 651–674.

Andreas Bauer, Martin Leucker, and Christian Schallhart. 2011. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20, 4 (2011), 14.

Nathalie Bertrand, Serge Haddad, and Engel Lefaucheux. 2014. Foundation of Diagnosis and Predictability in Probabilistic Systems. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'14).* New Delhi, India. https://hal.inria.fr/hal-01088117

Julian Bradfield and Colin Stirling. 2001. CHAPTER 4 - Modal Logics and mu-Calculi: An Introduction. In *Handbook of Process Algebra*, J.A. Bergstra, A. Ponse, and S.A. Smolka (Eds.). Elsevier Science, Amsterdam, 293 – 330. https://doi.org/10.1016/B978-044482830-9/50022-9

Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. 1981. Alternation. *Journal of the ACM* 28, 1 (jan 1981), 114–133. https://doi.org/10.1145/322234.322243

Edward Chang, Zohar Manna, and Amir Pnueli. 1992. Characterization of Temporal Property Classes. In *Automata, Languages and Properties (LNCS)*, Vol. 623. Springer-Verlag, 474–486.

Clare Cini and Adrian Francalanza. 2015. An LTL Proof System for Runtime Verification. In *TACAS*, Vol. 9035. Springer, 581–595.

Edmund M Clarke, Orna Grumberg, and Doron Peled. 1999. *Model Checking.* MIT press.

Ben d'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B Sipma, Sandeep Mehrotra, and Zohar Manna. 2005. LOLA: Runtime monitoring of synchronous systems. In *Temporal Representation and Reasoning, 2005. TIME 2005. 12th International Symposium on.* IEEE, 166–174.

Rocco De Nicola and Frits Vaandrager. 1990. Action versus state based logics for transition systems. In *Semantics of Systems of Concurrent Processes*, Irène Guessarian (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 407–419. https://doi.org/10

Volker Diekert and Martin Leucker. 2014. Topology, monitorable properties and runtime verification. *Theoretical Computer Science* 537 (2014), 29–41. https://doi.org/10.1016/j.tcs.2014.02.052

Volker Diekert, Anca Muscholl, and Igor Walukiewicz. 2015. A Note on Monitors and Büchi Automata. In *Theoretical Aspects of Computing - ICTAC 2015, 12th International Colloquium (Lecture Notes in Computer Science)*, Martin Leucker, Camilo Rueda, and Frank D. Valencia (Eds.), Vol. 9399. Springer, 39–57. https://doi.org/10.1007/978-3-319-25150-9

Doron Drusinsky. 2003. Monitoring temporal rules combined with time series. In *CAV*, Vol. 3. Springer, 114–118.

Sonali Dutta, Moshe Y. Vardi, and Deian Tabakov. 2014. CHIMP: A Tool for Assertion-Based Dynamic Verification of SystemC Models. In *Proceedings of the Second International Workshop on Design and Implementation of Formal Tools and Systems (CEUR Workshop Proceedings)*, Malay K. Ganai and Alper Sen (Eds.), Vol. 1130. CEUR-WS.org. http://ceur-ws.org/Vol-1130/paper_8.pdf

Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. 2012a. What can you verify and enforce at runtime? *STTT* 14, 3 (2012), 349–382.

Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. 2012b. What can you verify and enforce at runtime? *International Journal on Software Tools for Technology Transfer* 14, 3 (01 Jun 2012), 349–382. https://doi.org/10.1007/s10009-011-0196-8

A. Fellah, H. Jürgensen, and S. Yu. 1990. Constructions for alternating finite automata∗. *International Journal of Computer Mathematics* 35, 1-4 (jan 1990), 117–132. https://doi.org/10.1080/00207169008803893

Bernd Finkbeiner and Henny Sipma. 2004. Checking finite traces using alternating automata. *Formal Methods in System Design* 24, 2 (2004), 101–127.

Adrian Francalanza. 2016. A Theory of Monitors. In *FoSSaCS (LNCS)*, Vol. 9634. 145–161.

Adrian Francalanza. 2017. Consistently-Detecting Monitors. In *CONCUR (LIPIcs)*, Vol. 85. Schloss Dagstuhl, Dagstuhl, Germany, 8:1–8:19. https://doi.org/10.4230/LIPIcs.CONCUR.2017.8

Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. 2017a. A Foundation for Runtime Monitoring. In *RV*. 8–29.

Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. 2015. On Verifying Hennessy-Milner Logic with Recursion at Runtime. In *RV (LNCS)*, Vol. 9333. 71–86. https://doi.org/10.1007/978-3-319-23820-3_5

Adrian Francalanza, Luca Aceto, and Anna Ingolfsdottir. 2017b. Monitorability for the Hennessy–Milner logic with recursion. *FMSD* (2017), 1–30.

Adrian Francalanza and Aldrin Seychell. 2015. Synthesising correct concurrent runtime monitors. *Formal Methods in System Design* 46, 3 (2015), 226–261. https://doi.org/10.1007/s10703-014-0217-9

Dimitra Giannakopoulou and Klaus Havelund. 2001. Runtime analysis of linear temporal logic specifications. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering, San Diego, California.*

Klaus Havelund and Grigore Rosu. 2002. Synthesizing monitors for safety properties. In *TACAS*, Vol. 2. Springer, 342–356.

Matthew Hennessy and Robin Milner. 1985. Algebraic Laws for Nondeterminism and Concurrency. *J. ACM* 32, 1 (1985), 137–161. https://doi.org/10.1145/2455.2460

Robert M. Keller. 1976. Formal Verification of Parallel Programs. *Commun. ACM* 19, 7 (1976), 371–384. https://doi.org/10.1145/360248.360251

Dexter C. Kozen. 1983. Results on the Propositional μ-calculus. *Theoretical Computer Science* 27 (1983), 333–354.

Orna Kupferman, Moshe Y Vardi, and Pierre Wolper. 2000. An automata-theoretic approach to branching-time model checking. *Journal of the ACM (JACM)* 47, 2 (2000), 312–360.

Martin Lange. 2005. Weak Automata for the Linear Time μ-Calculus. In *Verification, Model Checking, and Abstract Interpretation*, Radhia Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 267–281. https://doi.org/10.1007/978-3-540-30579-8_18

Kim G. Larsen. 1990. Proof Systems for Satisfiability in Hennessy-Milner Logic with recursion. *Theoretical Computer Science (TCS)* 72, 2 (1990), 265 – 288. https://doi.org/10.1016/0304-3975(90)90038-J

Zohar Manna and Amir Pnueli. 1991. Completing the Temporal Picture. *TCS* 83, 1 (1991), 97–130. https://doi.org/10.1016/0304-3975(91)90041-Y

Nicolas Markey and Philippe Schnoebelen. 2006. Mu-calculus path checking. *Inform. Process. Lett.* 97, 6 (March 2006), 225–230. https://doi.org/10.1016/j.ipl.2005.11.010

R. Milner. 1989. *Communication and Concurrency.* Prentice-Hall.

A. Pnueli and A. Zaks. 2006. PSL Model Checking and Run-time Verification via Testers. In *FM*. Springer, 573–586. https://doi.org/10.1007/11813040_38

Meera Sampath, Raja Sengupta, Stephane Lafortune, Kasim Sinnamohideen, and Demosthenis Teneketzis. 1995. Diagnosability of discrete-event systems. *IEEE Transactions on automatic control* 40, 9 (1995), 1555–1575. https://doi.org/10.1109/9.412626

Fred B. Schneider. 1997. *On Concurrent Programming.* Springer-Verlag.

Deian Tabakov, Kristin Y. Rozier, and Moshe Y. Vardi. 2012. Optimized temporal monitors for SystemC. *Formal Methods in System Design* 41, 3 (2012), 236–268. https://doi.org/10.1007/s10703-011-0139-8

M. Y. Vardi. 1988a. A Temporal Fixpoint Calculus. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88).* ACM, New York, NY, USA, 250–259. https://doi.org/10.1145/73560.73582

Moshe Y. Vardi. 1988b. A Temporal Fixpoint Calculus. In *POPL*. ACM, New York, NY, USA, 250–259. https://doi.org/10.1145/73560.73582

Mahesh Viswanathan and Moonzoo Kim. 2004. Foundations for the run-time monitoring of reactive systems–fundamentals of the MaC language. In *International Colloquium on Theoretical Aspects of Computing*. Springer, 543–556.

Thomas Wilke. 2001. Alternating Tree Automata, Parity Games, and Modal m-Calculus. *Bulletin of the Belgian Mathematical Society Simon Stevin* 8, 2 (2001), 359.