Formal Methods in System Design

Monitorability for the Hennessy-Milner Logic with Recursion

--Manuscript Draft--

Manuscript Number:	FORM-D-16-00031
Full Title:	Monitorability for the Hennessy-Milner Logic with Recursion
Article Type:	S.I. : FMSD for RV 2015
Keywords:	monitorability; branching-time logics, monitor correctness; labelled transition systems; process calculi
Corresponding Author:	Adrian Francalanza, PhD University of Malta Msida, MALTA
Corresponding Author Secondary Information:	
Corresponding Author's Institution:	University of Malta
Corresponding Author's Secondary Institution:	
First Author:	Adrian Francalanza, PhD
First Author Secondary Information:	
Order of Authors:	Adrian Francalanza, PhD
	Luca Aceto
	Anna Ingolfsdottir
Order of Authors Secondary Information:	
Funding Information:	Icelandic Centre for Research (163406-051) — Prof Luca Aceto
Abstract:	We study muHML, a branching-time logic with least and greatest fix-points, from a runtime verification perspective. The logic may be used to specify properties of programs whose behaviour may be expressed as a labelled transition system. We establish which subset of this logic can be monitored for at runtime by merely observing the runtime execution of a program. We define a monitor- synthesis algorithm for this subset and then show that the synthesised monitors correctly perform the required analysis from the observed behaviour. We also prove completeness results with respect to this logical subset that show that, up to logical equivalence, no other properties apart from those identified can be monitored for and verified at runtime.

Noname manuscript No.
(will be inserted by the editor)

Monitorability for the Hennessy-Milner Logic with Recursion

Adrian Francalanza · Luca Aceto · Anna Ingolfsdottir

Abstract We study μHML, a branching-time logic with least and greatest fixpoints, from a runtime verification perspective. The logic may be used to specify properties of programs whose behaviour may be expressed as a labelled transition system. We establish which subset of this logic can be monitored for at runtime by merely observing the runtime execution of a program. We define a monitor-synthesis algorithm for this subset and then show that the synthesised monitors correctly perform the required analysis from the observed behaviour. We also prove completeness results *wrt.* this logical subset that show that, up to logical equivalence, no other properties apart from those identified can be monitored for and verified at runtime.

The research of L. Aceto and A. Ingolfsdottir was partly supported by the project 001-ABEL-CM-2013 of the NILS Science and Sustainability Programme. The research of all three authors was also supported by the project Theoretical Foundations of Monitorability (nr. 163406-051) of the Icelandic Research Fund.

Adrian Francalanza
Dept. of Computer Science,
ICT, University of Malta, Msida, Malta.
Tel.: +356-2340-2745
Fax: +356-2132-0539
E-mail: adrian.francalanza@um.edu.mt

Luca Aceto
School of Computer Science,
Reykjavik University, Reykjavik, Iceland
Tel. : +354 599 6419
Fax. : +354 599 6301
E-mail: luca@ru.is

Anna Ingolfsdottir
School of Computer Science,
Reykjavik University, Reykjavik, Iceland
Tel. : +354 599 6319
Fax. : +354 599 6301
E-mail: anna@ru.is

1 Introduction

Runtime Verification (RV) [29,21] is a lightweight verification technique whereby the *execution of a system* is analysed with the aim of inferring correctness *wrt.* some property. The technique is often used to mitigate scalability issues such as state-explosion problems, typically associated with more traditional verification techniques like Model Checking. RV is also useful in settings where the full execution state space of a system is not accessible or available prior deployment, and the only means of analysing the behaviour of a system is by observing its current system run. Despite its advantages, the technique is generally limited when compared to other verification techniques that consider system behaviour beyond its current execution [30,16]. For instance, *online* RV incrementally analyses *partial* executions (up to the current execution point of the system) which limits its applicability to satisfaction analyses relating to correctness properties that describe complete (*i.e.,* potentially infinite) executions.

There are broadly two approaches in the field of RV that are used to address such a limitation. The first approach is to *restrict* the expressive power of the correctness specifications: typically, one either limits specifications to *descriptions of finite traces* such as regular expressions (RE) [7,23], or else *redefines* the semantics of existing logics (*e.g.,* LTL) so as to reflect the limitations of the runtime setting [20,9,11,10]. The second approach is to leave the semantics of the specification logic *unchanged*, and then identifying which *subsets* of the logic can be verified at runtime, *wrt.* a specific monitoring setup [25,33,17,24,19,35].

Both approaches have their merits. For instance, the first approach is generally more popular and tends to produce specifications that are closely related to the monitors that check for them (*e.g.,* RE and automata in [32,7,23]), thus facilitating aspects such as monitor correctness (which is well-understood in certain settings). On the other hand, the second approach does not hinder the expressive power of the logic. Instead, it allows a verification framework to determine whether to synthesise a monitor that checks for a property at runtime (when possible), or else to employ more powerful (and expensive) verification techniques such as model-checking. One can even envisage a combined approach, where some parts of a property are verified using RV and other parts of the property are checked using other techniques, along the lines of [4].

More importantly, however, the second approach leads to better *separation of concerns*: since it is *agnostic* of the verification technique used, one can change the method of verification without impinging on the property semantics. Prosaically, when specifying the correct behaviour expected of a system (typically during the design stage of development) one should ideally be unencumbered by how this correctness specification is checked for in the *resp.* target system. Instead, the focus at this development stage should be on the expressiveness of the specification formalism used. Indeed, there are many cases where the verification strategy to be adopted becomes apparent later on in the development process or else is changed due to evolving requirements of the software project (*e.g.,* the development team may decide to use a proprietary software component whose source code is unavailable). Alternatively, the project requirements may change in such a way so as to require more expressive logics to be able to specify the augmented correctness requirements.

This work adheres to the second approach discussed above. In particular, it revisits the Hennessy-Milner Logic with recursion [28,3], μHML, a reformulation of the modal μ-calculus [27], used to describe correctness properties of reactive system. We choose to conduct our study in this logic for a number of reasons. The logic has been shown to be a very powerful and expressive formalism for describing behavioural properties of systems; it is more expressive than widely used temporal logics such as CTL and CTL* [18,6] which suggests that the findings of this work can be carried over to these logics. In addition, our work can serve to complete the picture for existing work in the context of RV: fragments of μHML have already been adapted for detectEr [34], an RV tool used to specify and verify at runtime actor-based reactive systems [24,12–14], and constructs from the modal μ-calculus have been used in other RV tools such as Eagle [8]. In this study, we consider the logic in its entirety, and investigate the monitorability of fragments of the logic *wrt.* an operational definition of a general class of monitors that employ both acceptance and rejection verdicts [11,21]. It turns out that our results extend the class of monitorable μHML properties used in [24] and establish monitorability upper bounds for this logic. We also present new results that relate the utility of multi-verdict monitors for logics that are defined over program execution graphs, in contrast to other work on monitorability that considers logics and properties defined over traces [11,21,17]. The concrete contributions of the paper are:

1. An operational definition of what it means for a formula in a branching-time logic to be monitored by a specific monitor, Definition 4. Underpinning this definition is an instrumentation relation unifying the individual behaviour of processes and monitors as a single monitored system entity, given in Figure 4 that complements other foundational work on monitor behavior [22,13].
2. A definition that formalises when a logical formula is considered to be monitorable in this setting, Definition 5, that is then lifted to logical languages, used to define monitorable subsets of μHML.
3. The identification of a subset of μHML, Definition 6, that is shown to be monitorable according to Definition 5 (see Theorem 1) but also maximally expressive *wrt.* this monitorability definition, Theorem 4. This means that there are no other (semantically equivalent) properties in μHML that can be monitored for in the setting that we consider.
4. A result asserting that with respect to the monitor setting of Definition 4, univerdict monitors, *i.e.,* monitors with just an acceptance or a rejection verdict, suffice to attain the full expressive monitoring power, Theorem 2. We assert that this property is inherent to branching-time specifications in general.

To the best of our knowledge, this is one of the first bodies of work (along with [35]) investigating the limits of RV *wrt.* the *branching-time* dimension of logical specifications (*i.e.,* properties that talk about the *execution graph* of a program). Other work pertaining to the aforementioned second approach tend to follow a more traditional approach, focussing on *linear-time* logics defined over *execution traces*, exploring RV's limits along the linear-time dimension, *e.g.,* [17].

The rest of the paper is structured as follows. Section 2 introduces our model for reactive systems and Section 3 presents the logic μHML defined over this model. Section 4 formalises our abstract RV operational setup in terms of monitors and an instrumentation relation. In Section 5 we argue for a particular correspondence between monitors and μHML properties within this setup. Section 6 identifies

Syntax

$$
\begin{array}{llllll}
p,q,r \in \textsc{Proc} ::= \mathsf{nil} & \text{(inaction)} & \mid \alpha.p & \text{(prefixing)} & \mid p+q & \text{(choice)} \\
\mid p \parallel q & \text{(parallel)} & \mid \mathsf{new}\,\alpha.p & \text{(scoping)} & \mid \mathsf{rec}\,x.p & \text{(recursion)} \\
\mid x & \text{(rec. variable)} & & & &
\end{array}
$$

Dynamics

$$\textsc{Act}\ \frac{}{\alpha.p \xrightarrow{\alpha} p} \qquad \textsc{SelL}\ \frac{p \xrightarrow{\mu} p'}{p+q \xrightarrow{\mu} p'} \qquad \textsc{SelR}\ \frac{q \xrightarrow{\mu} q'}{p+q \xrightarrow{\mu} q'}$$

$$\textsc{Syn}\ \frac{p \xrightarrow{\alpha} p' \quad q \xrightarrow{\overline{\alpha}} q'}{p \parallel q \xrightarrow{\tau} p' \parallel q'} \qquad \textsc{ParL}\ \frac{p \xrightarrow{\mu} p'}{p \parallel q \xrightarrow{\mu} p' \parallel q} \qquad \textsc{ParR}\ \frac{q \xrightarrow{\mu} q'}{p \parallel q \xrightarrow{\mu} p \parallel q'}$$

$$\textsc{Rec}\ \frac{}{\mathsf{rec}\,x.p \xrightarrow{\tau} p[\mathsf{rec}\,x.p/x]} \qquad \textsc{Scp}\ \frac{p \xrightarrow{\mu} p' \quad \mu \notin \{\alpha, \overline{\alpha}\}}{\mathsf{new}\,\alpha.p \xrightarrow{\mu} \mathsf{new}\,\alpha.p'}$$

Fig. 1 A Model for describing Systems

monitorability limits for the logic but also establishes a monitorable logical subset that satisfies the correspondence of Section 5. Section 7 shows that this subset is maximally expressive, using a result about multi-verdict monitors. Section 8 concludes by discussing related and future work.

2 The Model

We describe systems abstractly as Labelled Transition Systems (LTSs) [3,27]. An LTS is a triple $\langle \textsc{Proc}, (\textsc{Act} \cup \{\tau\}), \longrightarrow \rangle$ consisting of a set of states, $p, q, \ldots \in \textsc{Proc}$, a set of actions, $\alpha, \overline{\alpha}, \beta, \overline{\alpha}, \ldots \in \textsc{Act}$ with distinguished silent action τ, and a transition relation, $\longrightarrow \subseteq (\textsc{Proc} \times (\textsc{Act} \cup \{\tau\}) \times \textsc{Proc})$. We assume that the set of actions Act comes equipped with an involution function $\overline{\alpha}$, where $\overline{\overline{\alpha}} = \alpha$, and where $\overline{\alpha}$ is considered to be the complement of α. We also assume that $\mu \in \textsc{Act} \cup \{\tau\}$, that $\tau \notin \textsc{Act}$ and refer to the actions in Act as *visible* actions (as opposed to the silent action τ).

LTS states can be expressed as processes, Proc, from a standard variant of CCS [31] as defined by the syntax in Figure 1. Assuming a specific set of (visible) actions, Act and a denumerable set of (recursion) variables $x, y, z \in \textsc{Vars}$, processes are defined as either the inactive process nil, a prefixed process by an action α, a mutually-exclusive choice amongst two processes, a parallel composition of two processes, a scoping of an action in a process, or a recursive process. The process $\mathsf{rec}\,x.p$ acts as a *binder* for x in p whereas the process $\mathsf{new}\,\alpha.p$ acts as a binder for actions $\alpha, \overline{\alpha}$ in p; we work up to alpha-conversion of bound variables and scoped actions. All recursive processes are assumed to be guarded, meaning that all occurrences of bound variables occur under an action prefix (either directly or indirectly). Closed terms are processes where all occurrences of variables are bound. Occasionally, we elide inactive processes when describing systems and write p instead of $\mathsf{nil} \parallel p$ or $p \parallel \mathsf{nil}$ and $\alpha.\beta.\mathsf{nil}$ as $\alpha.\beta$; in order to lighten our presen-

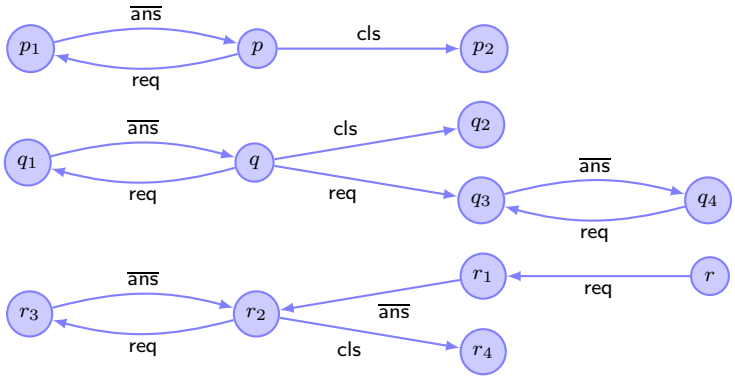

Fig. 2 A depiction of the system in Example 1

tation we also implicitly work up to associativity and commutativity of parallel compositions where, for example, $p \| q$ is some times treated as $q \| p$.[1]

The dynamic behaviour of a process is described by the transition rules of Figure 1, defined over the closed and guarded terms in Proc. The suggestive notation $p \xrightarrow{\mu} p'$ denotes $(p, \mu, p') \in \longrightarrow$; we also write $p \xrightarrow{\mu}\!\!\!\!\!\!/\;$ to denote $\neg(\exists p'.\ p \xrightarrow{\mu} p')$. The rules in Figure 1 are standard. For example, $p_1 + p_2 \xrightarrow{\mu} q$ if either $p_1 \xrightarrow{\mu} q$ or $p_2 \xrightarrow{\mu} q$ by rules SelL or SelR; dually, $p_1 \| p_2 \xrightarrow{\mu} q$ if $p_1 \xrightarrow{\mu} p_1'$ and $q = p_1' \| p_2$ for some p_1', or $p_2 \xrightarrow{\mu} p_2'$ and $q = p_1 \| p_2'$ for some p_2'. We note that rule Syn describes the synchronisation of two parallel processes on complementing actions whereas rule Scp restricts visible actions to those that are not scoped by α in $\mathsf{new}\,\alpha.p$. We employ the usual notation for weak transitions and write $p \Longrightarrow p'$ in lieu of $p(\xrightarrow{\tau})^* p'$ and $p \overset{\mu}{\Longrightarrow} p'$ for $p \Longrightarrow \cdot \xrightarrow{\mu} \cdot \Longrightarrow p'$, referring to p' as a μ-derivative of p. We let $t, u \in \text{Act}^*$ range over sequences of visible actions and write sequences of transitions $p \overset{\alpha_1}{\Longrightarrow} \ldots \overset{\alpha_n}{\Longrightarrow} p_n$ as $p \overset{t}{\Longrightarrow} p_n$, where $t = \alpha_1, \ldots, \alpha_n$. For more details the reader is invited to consult standard texts such as [31,3] .

Example 1 A (reactive) system that acts as a server that repeatedly accepts *requests* and subsequently *answers* them, with the possibility of terminating through the special *close* request, may be expressed as the following process, p.

$$p = \mathsf{rec}\, x.\big(\mathsf{req}.\overline{\mathsf{ans}}.x + \mathsf{cls}.\mathsf{nil}\big)$$

A server that non-deterministically stops offering the close action is denoted by process q, whereas r only offers the close action after the first serviced request.

$$q = \mathsf{rec}\, x.\big(\mathsf{req}.\overline{\mathsf{ans}}.x + \mathsf{cls}.\mathsf{nil} + (\mathsf{rec}\, y.\mathsf{req}.\overline{\mathsf{ans}}.y)\big)$$
$$r = \mathsf{req}.\overline{\mathsf{ans}}.\mathsf{rec}\, x.\big(\mathsf{req}.\overline{\mathsf{ans}}.x + \mathsf{cls}.\mathsf{nil}\big)$$

Pictorially, the *resp.* LTSs denoted by processes p, q and r may be represented by the graphs in Figure 2, where the nodes correspond to processes and the arcs

[1] These are normally expressed as structural equivalence rules such as $p \parallel \mathsf{nil} \equiv p$ and $p \| q \equiv q \| p$ in standard CCS. We elide them here to alleviate our exposition.

correspond to weak transitions, $\stackrel{\alpha}{\Longrightarrow}$. Note that these graphs do not describe the full branching structure of the *resp.* processes and abstract away from τ-derivatives such as $\big(\mathsf{req}.\overline{\mathsf{ans}}.p + \mathsf{cls}.\mathsf{nil}\big)$ (in the case of p). This omission is intensional and the intuition conveyed by these graphs, namely finite depictions of *sets of transions* $p' \stackrel{t}{\Longrightarrow} p''$, suffices for the technical development to follow. ■

Example 2 Consider the system described by the process p' below:

$$p' = \mathsf{rec}\, x.\big(\mathsf{req}.\mathsf{work}.\mathsf{nil} + \mathsf{cls}.\mathsf{nil} \parallel \overline{\mathsf{work}}.\overline{\mathsf{ans}}.x\big)$$

By abstracting away from τ-transitions, we can also use the finite graph for p from Figure 2 to describe the behaviour of the process $\mathsf{new}\,\mathsf{work}.p'$, where the action work is locally scoped. In particular, such a graph abstracts away from the τ-transitions (1) and (2) below.

$$\begin{aligned} \mathsf{new}\,\mathsf{work}.p' &\xrightarrow{\tau} \mathsf{new}\,\mathsf{work}.\big(\mathsf{req}.\mathsf{work}.\mathsf{nil} + \mathsf{cls}.\mathsf{nil} \parallel \overline{\mathsf{work}}.\overline{\mathsf{ans}}.p'\big) && (1)\\ &\xrightarrow{\mathsf{req}} \mathsf{new}\,\mathsf{work}.\big(\mathsf{work}.\mathsf{nil} \parallel \overline{\mathsf{work}}.\overline{\mathsf{ans}}.p'\big)\\ &\xrightarrow{\tau} \mathsf{new}\,\mathsf{work}.\big(\overline{\mathsf{ans}}.p'\big) && (2)\\ &\xrightarrow{\overline{\mathsf{ans}}} \mathsf{new}\,\mathsf{work}.p' \end{aligned}$$

Importantly, *not* all processes describe a finite execution graph. As a simple demonstration of this, consider the process variants p'' and p''' below:

$$\begin{aligned} p'' &= \mathsf{rec}\, x.\mathsf{new}\,\mathsf{work}.\big(\overline{\mathsf{work}}.\mathsf{nil} \parallel \mathsf{req}.\overline{\mathsf{ans}}.x + \mathsf{cls}.\mathsf{nil}\big)\\ p''' &= \mathsf{rec}\, x.\big(\mathsf{req}.\overline{\mathsf{ans}}.(x \parallel x) + \mathsf{cls}.\mathsf{nil}\big) \end{aligned}$$

In the case of p'', we can easily observe the sequence of non-repeating states were new parallel copies of $\overline{\mathsf{work}}.\mathsf{nil}$ are spawned (we rename the bound name work to work' to highlight distinct bindings):

$$\begin{aligned} p'' &\stackrel{\mathsf{req}\cdot\overline{\mathsf{ans}}}{\Longrightarrow} \mathsf{new}\,\mathsf{work}.(\overline{\mathsf{work}}.\mathsf{nil} \parallel p'')\\ &\stackrel{\mathsf{req}\cdot\overline{\mathsf{ans}}}{\Longrightarrow} \mathsf{new}\,\mathsf{work}.\big(\overline{\mathsf{work}}.\mathsf{nil} \parallel \mathsf{new}\,\mathsf{work'}.(\overline{\mathsf{work'}}.\mathsf{nil} \parallel p'')\big) \stackrel{\mathsf{req}\cdot\overline{\mathsf{ans}}}{\Longrightarrow} \ldots \end{aligned}$$

We can also observer a similar expansion in the case of p'''. For instance, consider the execution trace below:

$$p''' \stackrel{\mathsf{req}\cdot\overline{\mathsf{ans}}}{\Longrightarrow} p''' \parallel p''' \stackrel{\mathsf{req}\cdot\overline{\mathsf{ans}}}{\Longrightarrow} p''' \parallel p''' \parallel p''' \stackrel{\mathsf{req}\cdot\overline{\mathsf{ans}}}{\Longrightarrow} \ldots$$

In fact, the execution graph of p''' is far more complicated than that since req and $\overline{\mathsf{ans}}$ from the generated parallel copies of p''' can arbitrarily be interleaved. ■

Syntax

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

Semantics

$$
\begin{array}{ll}
[\![\mathsf{tt}, \rho]\!] \stackrel{\text{def}}{=} \textsc{Proc} & [\![\mathsf{ff}, \rho]\!] \stackrel{\text{def}}{=} \emptyset \\
[\![\varphi_1 \wedge \varphi_2, \rho]\!] \stackrel{\text{def}}{=} [\![\varphi_1, \rho]\!] \cap [\![\varphi_2, \rho]\!] & [\![\varphi_1 \vee \varphi_2, \rho]\!] \stackrel{\text{def}}{=} [\![\varphi_1, \rho]\!] \cup [\![\varphi_2, \rho]\!] \\
[\![[\alpha]\varphi, \rho]\!] \stackrel{\text{def}}{=} \left\{ p \mid p \stackrel{\alpha}{\Rightarrow} q \text{ implies } q \in [\![\varphi, \rho]\!] \right\} & [\![\langle \alpha \rangle \varphi, \rho]\!] \stackrel{\text{def}}{=} \left\{ p \mid p \stackrel{\alpha}{\Rightarrow} q \text{ and } q \in [\![\varphi, \rho]\!] \right\} \\
[\![\mathsf{min} X.\varphi, \rho]\!] \stackrel{\text{def}}{=} \bigcap \{ S \mid [\![\varphi, \rho[X \mapsto S]]\!] \subseteq S \} & \\
[\![\mathsf{max} X.\varphi, \rho]\!] \stackrel{\text{def}}{=} \bigcup \{ S \mid S \subseteq [\![\varphi, \rho[X \mapsto S]]\!] \} & [\![X, \rho]\!] \stackrel{\text{def}}{=} \rho(X)
\end{array}
$$

Fig. 3 μHML Syntax and Semantics

3 The Logic

The logic μHML [28,3] assumes a countable set of logical variables $X, Y \in \textsc{LVar}$, and is defined as the set of *closed* formulae generated by the grammar of Figure 3. Apart from the standard constructs for truth, falsehood, conjunction and disjunction, the logic is equipped with possibility and necessity modal operators, together with recursive formulae expressing least or greatest fixpoints; formulae $\mathsf{min}\, X.\varphi$ and $\mathsf{max}\, X.\varphi$ *resp.* 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.

Formulae are interpreted over the process powerset domain, $S \in \mathcal{P}(\textsc{Proc})$. The semantic definition of Figure 3 is given for *both* open and closed formulae and employs an environment from logical variables to sets of processes, $\rho \in \textsc{LVar} \rightharpoonup \mathcal{P}(\textsc{Proc})$. This permits an inductive definition of $[\![\varphi, \rho]\!]$, the set of processes satisfying the formula φ *wrt.* an environment ρ, based on the structure of the formula. For instance, in Figure 3, the semantic meaning of a variable X *wrt.* an environment ρ is the mapping for that variable in ρ. The semantics of truth, falsehood, conjunction and disjunction are standard (*e.g.,* $\vee$ and $\wedge$ are interpreted as set-theoretic union and intersection). Possibility formulae $\langle \alpha \rangle \varphi$ describe processes with *at least one* α-derivative satisfying φ whereas necessity formulae $[\alpha]\varphi$ describe processes where *all* of their α-derivatives (possibly none) satisfy φ. The powerset domain $\mathcal{P}(\textsc{Proc})$ is a complete lattice *wrt.* set-inclusion, $\subseteq$, which guarantees the existence of least and largest solutions for the recursive formulae of the logic; as usual, these can be *resp.* specified as the intersection of all the pre-fixpoint solutions and the union of all post-fixpoint solutions [3]. Note that $\rho[X \mapsto S]$ denotes an environment ρ' where $\rho'(X) = S$ and $\rho'(Y) = \rho(Y)$ for all other $Y \neq X$. Since the interpretation of closed formulae is independent of the environment ρ, we sometimes write $[\![\varphi]\!]$ in lieu of $[\![\varphi, \rho]\!]$. We say that a process p *satisfies* a closed formula φ whenever $p \in [\![\varphi]\!]$, and violates a formula whenever $p \notin [\![\varphi]\!]$.

Example 3 Formula $\langle \alpha \rangle \mathsf{tt}$ describes processes that *can* perform action α whereas formula $[\alpha]\mathsf{ff}$ describes processes that *cannot* perform action α. Consider the for-

mulae defined thus:

$$\varphi_1 = \mathsf{min}\, X.(\langle \mathsf{req}\rangle\langle \overline{\mathsf{ans}}\rangle X \vee [\mathsf{cls}]\mathsf{ff})$$
$$\varphi_2 = \mathsf{max}\, X.(\langle \mathsf{req}\rangle\langle \overline{\mathsf{ans}}\rangle X \vee [\mathsf{cls}]\mathsf{ff})$$
$$\varphi_3 = \mathsf{max}\, X.([\mathsf{req}][\overline{\mathsf{ans}}]X \wedge \langle \mathsf{cls}\rangle \mathsf{tt})$$
$$\varphi_4 = \mathsf{max}\, X.([\mathsf{req}][\overline{\mathsf{ans}}]X \wedge [\mathsf{cls}]\mathsf{ff})$$

Formula φ_1 denotes a *liveness* property describing processes that stop offering the action cls after some number of serviced request, *i.e.,* after performing some sequence of actions in $(\mathsf{req}.\overline{\mathsf{ans}})^*$—processes q and r from Example 1 satisfy this property but p does not. Concretely, in the case of q we can observe the computation $q \stackrel{\mathsf{req}\cdot\overline{\mathsf{ans}}}{\Longrightarrow} q_4$ and $q_4 \in [\![[\mathsf{cls}]\mathsf{ff}]\!]$ since $q_4 \stackrel{\mathsf{cls}}{\not\Longrightarrow}$, whereas in the case of r we immediately have $r \stackrel{\mathsf{cls}}{\not\Longrightarrow}$. Intuitively, p does not satisfy φ_1 because we have $p \stackrel{\mathsf{cls}}{\Longrightarrow}$ and whenever $p \stackrel{\mathsf{req}\cdot\overline{\mathsf{ans}}}{\Longrightarrow} p'$ for some p', then it necessarily the case that $p' = p$ again (or its unfolding $\mathsf{req}.\overline{\mathsf{ans}}.p + \mathsf{cls}.\mathsf{nil}$). If we change the fixpoint of φ_1 into a maximal one, *i.e.,* φ_2, the resulting formula would include p in the property as well. This is because $[\![\varphi_2]\!]$ also includes processes that exhibit the *infinite* trace $(\mathsf{req}\cdot\overline{\mathsf{ans}})^*$, which process p can display by following the looping transition sequence $p \stackrel{\mathsf{req}\cdot\overline{\mathsf{ans}}}{\Longrightarrow} p$.

Formulae φ_3 and φ_4 denote *safety* properties: *e.g.,* φ_3, describes (terminating and non-terminating) processes that can *always* perform a cls action after any number of serviced request, *i.e.,* after having performed any trace in $(\mathsf{req}.\overline{\mathsf{ans}})^*$. Process p satisfies property φ_3, due to an analogous argument to the one above justifying $p \notin [\![\varphi_1]\!]$. However q and r do not satisfy this property because not all execution paths satisfy this property. In the case of process q, although paths along the looping sequence $q \stackrel{\mathsf{req}\cdot\overline{\mathsf{ans}}}{\Longrightarrow} q$ *validate* this property (since q itself can perform the (weak) transition $q \stackrel{\mathsf{cls}}{\Longrightarrow}$), it *violates* this property along a different path, such as in the case of $q \stackrel{\mathsf{req}\cdot\overline{\mathsf{ans}}}{\Longrightarrow} q_4$.

$$\varphi_5 = \langle \mathsf{req}\rangle\langle \overline{\mathsf{ans}}\rangle \mathsf{max}\, X.\big(([\mathsf{req}]\mathsf{ff} \vee \langle \mathsf{req}\rangle\langle \overline{\mathsf{ans}}\rangle X) \wedge [\mathsf{cls}]\mathsf{ff}\big)$$
$$\varphi_6 = \mathsf{min}\, X.\big((\langle \mathsf{req}\rangle\langle \overline{\mathsf{ans}}\rangle \mathsf{tt} \wedge [\mathsf{req}][\overline{\mathsf{ans}}]X) \vee \langle \mathsf{cls}\rangle \mathsf{tt}\big)$$

Formula φ_5 is satisfied by processes that, after one serviced request, $\mathsf{req}.\overline{\mathsf{ans}}$, exhibit *a* maximal[2] transition sequence in $(\mathsf{req}.\overline{\mathsf{ans}})^*$ that never offers action cls. Process q satisfies this property through the path $q \stackrel{\mathsf{req}\cdot\overline{\mathsf{ans}}}{\Longrightarrow} q_4$, whereas processes p and r do not satisfy such a property. Formula φ_6 describes processes that along *all* serviced request sequences, *i.e.,* traces in $(\mathsf{req}.\overline{\mathsf{ans}})^*$, eventually reach a state where they offer action cls. All the processes in Figure 2 satisfy this property: p and q satisfy the criteria immediately, process whereas r satisfies it for $(\mathsf{req}.\overline{\mathsf{ans}})^n$ sequences where $n \geq 1$. At an intuitive level, property φ_6 is, in some sense, analogous to the LTL formula $\langle \mathsf{req}\rangle\langle \overline{\mathsf{ans}}\rangle$ **U** $\langle \mathsf{cls}\rangle \mathsf{tt}$. ■

4 Monitors and instrumentation

Monitors are often expressed in terms of automata [7,21] and may therefore also be viewed as LTSs, through the syntax of Figure 4. The syntax is similar to a

[2] A transition sequence is maximal if it is either infinite or affords no more actions.

Syntax

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

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

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

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

Fig. 4 Monitors and Instrumentation

sub-language of processes, with the exception that the inert process nil, is replaced by three *verdict* constructs, yes, no and end, *resp.* denoting acceptance, rejection and termination (*i.e.,* an inconclusive outcome). Monitor behaviour is similar to that of processes for the common constructs; see the transition rules in Figure 4. The only new transition rule concerns verdicts, mVer, stating that a verdict may transition with *any* $\alpha \in \textsc{Act}$ and go back to the same state, thereby modelling the requirement that verdicts are *irrevocable*.

Figure 4 also describes an instrumentation relation, connecting the behaviour of a process p with that of a monitor m: the configuration $m \triangleleft p$ denotes a *monitored system*. In an instrumentation, the process leads the (visible) behaviour of a monitored system (*i.e.,* if the process cannot α-transition, then the monitored system will not either) while the monitor passively follows, transitioning accordingly; this is in contrast with well-studied parallel composition relations of LTSs [31, 26]. Specifically, rule iMon states that if a process can transition with action α and the *resp.* monitor can follow this by transitioning with the same action, then in an instrumented monitored system they transition in lockstep. However, if the monitor cannot follow such a transition, $m \overset{\alpha}{\not\longrightarrow}$, even after any number of internal actions, $m \overset{\tau}{\not\longrightarrow}$, instrumentation forces it to terminate with an inconclusive verdict, end, while the process is allowed to proceed unaffected; see rule iTer; we remark that since end can produce *any* action, future transitions by p are still allowed while the terminated monitor maintains its state, using the rule iMon. Rules iAsyP and iAsyM allow monitors and processes to transition independently *wrt.* internal moves. This means that when a monitor cannot match a process action, but can transition silently, the instrumentation allows it to do so, and the matching check is then postponed to the τ-derivative monitor.

Example 4 Consider the monitor m defined below, that reaches an *acceptance* verdict after observing the action req followed by the action $\overline{\mathsf{ans}}$, but reaches a rejection

verdict when it observes the action $\overline{\mathsf{ans}}$ first.

$$m = (\mathsf{req}.\overline{\mathsf{ans}}.\mathsf{yes}) + (\overline{\mathsf{ans}}.\mathsf{no})$$

When it is instrumented with process p from Example 1, we observe the following behaviour for the monitored system whereby on line (**) the monitor preserves the yes verdict for all remaining transitions.

$$\begin{aligned}
m \triangleleft p &\xrightarrow{\tau} m \triangleleft \mathsf{req}.\overline{\mathsf{ans}}.p + \mathsf{cls}.\mathsf{nil} && \text{using iAsyP}\\
&\xrightarrow{\mathsf{req}} \overline{\mathsf{ans}}.\mathsf{yes} \triangleleft \overline{\mathsf{ans}}.p && \text{using iMon}\\
&\xrightarrow{\overline{\mathsf{ans}}} \mathsf{yes} \triangleleft p && \text{using iMon}\\
&\xRightarrow{\mathsf{req}.\overline{\mathsf{ans}}} \mathsf{yes} \triangleleft p \xRightarrow{\mathsf{req}.\overline{\mathsf{ans}}} \mathsf{yes} \triangleleft p \xRightarrow{\mathsf{req}.\overline{\mathsf{ans}}} \ldots && (**)
\end{aligned}$$

When it is instrumented with a different system, such as $\overline{\mathsf{ans}}.p$, the monitor yields a different (persistent) verdict,

$$m \triangleleft \overline{\mathsf{ans}}.p \xrightarrow{\overline{\mathsf{ans}}} \mathsf{no} \triangleleft p \xRightarrow{\mathsf{req}.\overline{\mathsf{ans}}} \mathsf{no} \triangleleft p \xRightarrow{\mathsf{req}.\overline{\mathsf{ans}}} \ldots$$

Importantly, when m is instrumented with the process req.ans.nil we observe the following monitored system behaviour below. In the second transition, rule iTer is used because the emitted action, ans does not correspond with any of the actions the monitor is expecting at that stage, namely $\overline{\mathsf{ans}}$ (note the bar on one of the latter action, which makes it distinct from the former).

$$\begin{aligned}
m \triangleleft \mathsf{req}.\mathsf{ans}.\mathsf{nil} &\xrightarrow{\mathsf{req}} (\overline{\mathsf{ans}}.\mathsf{yes}) \triangleleft (\mathsf{ans}.\mathsf{nil}) && \text{using iMon}\\
&\xrightarrow{\mathsf{ans}} \mathsf{end} \triangleleft \mathsf{nil} && \text{using iTer}
\end{aligned}$$

It is important that a monitor is terminated (with an inconclusive verdict) whenever it cannot match the exhibited trace event by the process, since leaving the monitor state unaltered may lead to unwanted detections. Consider the instrumentation of monitor m with the process req.cls.$\overline{\mathsf{ans}}$.nil and let us analyse the runtime behaviour below:

$$\begin{aligned}
&m \triangleleft \mathsf{req}.\mathsf{cls}.\overline{\mathsf{ans}}.\mathsf{nil} \xrightarrow{\mathsf{req}} (\overline{\mathsf{ans}}.\mathsf{yes}) \triangleleft \mathsf{cls}.\overline{\mathsf{ans}}.\mathsf{nil}\\
&\qquad\xrightarrow{\mathsf{cls}} \mathsf{end} \triangleleft \overline{\mathsf{ans}}.\mathsf{nil} \xrightarrow{\overline{\mathsf{ans}}} \mathsf{end} \triangleleft \mathsf{nil} && \text{(Good)}\\
&m \triangleleft \mathsf{req}.\mathsf{cls}.\overline{\mathsf{ans}}.\mathsf{nil} \xrightarrow{\mathsf{req}} (\overline{\mathsf{ans}}.\mathsf{yes}) \triangleleft \mathsf{cls}.\overline{\mathsf{ans}}.\mathsf{nil}\\
&\qquad\xrightarrow{\mathsf{cls}} (\overline{\mathsf{ans}}.\mathsf{yes}) \triangleleft \overline{\mathsf{ans}}.\mathsf{nil} \xrightarrow{\overline{\mathsf{ans}}} \mathsf{end} \triangleleft \mathsf{nil} && \text{(Bad)}
\end{aligned}$$

The transition sequence labelled (Good) describes the behaviour according to the instrumentation rules in Figure 4 where, in particular, when the monitor cannot analyse action cls, it is terminated with an inconclusive state, which is then retained for the remaining transitions. By contrast, the transition sequence labelled (Bad) describes the behaviour obtained if we had to leave the state of a monitor unaltered when an action cannot be matched. Precisely, in (Bad), the residual monitor ends up matching the subsequent action, $\overline{\mathsf{ans}}$ and reaching an unintended acceptance verdict. ■

It is worth highlighting a few additional aspects of the instrumentation relation in Figure 4. First, note that the instrumentation LTS models the fact that a monitor does *not* have access to the execution graph of a process, and can only analyse the part of the execution graph exhibited by the process at runtime. We illustrate this in the following example.

Example 5 Recall monitor m defined in Example 4 and process p from Example 1. The monitored system may exhibit the following runtime behaviour

$$m \triangleleft p \xrightarrow{\tau} m \triangleleft \mathsf{req}.\overline{\mathsf{ans}}.p + \mathsf{cls}.\mathsf{nil} \xrightarrow{\mathsf{cls}} \mathsf{end} \triangleleft \mathsf{nil}$$

which prohibits the monitor from observing the execution path we saw before at (**) in Example 4, leading to an acceptance verdict. ■

Second, we draw attention to the fact that the instrumentation composition relation is *asymmetric* and lets a monitored system produce a transition as long as a system can produce that transition (even though the monitor *cannot* perform it). This models the requirement that monitors should (as much as possible) have a passive role in an instrumented setting and not interfere with the execution of a system. It also contrasts with the standard intersection and union constructions used in automata theory. We formalise these important properties in Proposition 1 below; for a more comprehensive study of the properties of this instrumentation relation, we refer the reader to [22].

Proposition 1 $m \triangleleft p \overset{t}{\Longrightarrow} m' \triangleleft p'$ *iff* $p \overset{t}{\Longrightarrow} p'$ *and*

(i) either $m \overset{t}{\Longrightarrow} m'$

(ii) or $m' = \mathsf{end}$ *and* $\exists t', \alpha, t'', m''.\ t = t'\alpha t'',\ m'' \overset{\tau}{\not\longrightarrow}$ *and* $m \overset{t'}{\Longrightarrow} m'' \overset{\alpha}{\not\longrightarrow}$.

Proof The *only-if* case is proved by numerical induction on the number of transions n in $m \triangleleft p \overset{t}{\Longrightarrow} m' \triangleleft p'$, where $\overset{t}{\Longrightarrow} = (\xrightarrow{\mu_n})^n$. The inductive case, *i.e.,* $n = k + 1$, performs a rule case analysis on the first monitored transition. The subcase where rule iTrm is used yields subcase (ii).

For the *if* case, we prove the two subcases separately. We first show the first subcase through

Lemma 1 $p \overset{t}{\Longrightarrow} p'$ *and* $m \overset{t}{\Longrightarrow} m'$ *implies* $m \triangleleft p \overset{t}{\Longrightarrow} m' \triangleleft p'$

by numerical induction on the number of transions n in $p \overset{t}{\Longrightarrow} p'$, where $\overset{t}{\Longrightarrow} = (\xrightarrow{\mu_n})^n$. The second subcase is proved by numerical induction on the number of transions n in $m \overset{t}{\Longrightarrow} m'' \overset{\alpha}{\not\longrightarrow}$ and $m'' \overset{\tau}{\not\longrightarrow}$, for $(\xrightarrow{\mu_n})^n = (\overset{t}{\Longrightarrow})$, where $t = t'\alpha t''$ for some t', α, t'', m'', *i.e.,* t can be decomposed into three parts where t' is the prefix of t that can be followed by the monitor, α is the first action that the monitor cannot follow, and t'' is the remaining trace after α. The subproof relies on the property that for any trace t and verdict v, we have $v \overset{t}{\Longrightarrow} v$, and uses Lemma 1 as well. □

Remark 1 Since we strive towards a general theory of monitorability, the monitor syntax and semantics in Figure 4 allows us to specify non-deterministic monitors such as $\alpha.\mathsf{yes} + \alpha.\mathsf{no}$ or $\alpha.\mathsf{nil} + \alpha.\beta.\mathsf{yes}$. There are settings where determinism

is unattainable (*e.g.,* distributed monitoring [23]) or desirable (*e.g.,* testers [32]), and others where non-determinism may be used to express behavioural under-specification, which can be then employed in frameworks for program refinement [1, 22]. Thus, expressing non-determinism allows us to study the cases where it is either considered useful, tolerated or considered erroneous. This also allows us to differentate between different forms of non-deterministic monitor behaviour, as we discuss in more detail in the following sections. ■

We conclude the section by considering monitors that can detect arbitrarily long execution traces, some of which also behave non-deterministically. We revisit these monitors in subsequent sections.

Example 6 Monitor m_1 (defined below) monitors for runtime executions whose trace is of the form $(\mathsf{req.ans})^*.\mathsf{cls}$ and returning the acceptance verdict yes, whereas m_2 dually rejects executions of that form. When composed with process p from Example 1, the monitored system $m_1 \triangleleft p$ may either service requests forever, $m_1 \triangleleft p \stackrel{\mathsf{req}}{\Longrightarrow} \cdot \stackrel{\mathsf{ans}}{\longrightarrow} m_1 \triangleleft p$, or else terminate with a yes verdict, $m_1 \triangleleft p \stackrel{\mathsf{cls}}{\Longrightarrow} \mathsf{yes} \triangleleft \mathsf{nil}$.

$$
\begin{aligned}
m_1 &= \mathsf{rec}\, x.\big(\mathsf{req}.\overline{\mathsf{ans}}.x + \mathsf{cls}.\mathsf{yes}\big)\\
m_2 &= \mathsf{rec}\, x.\big(\mathsf{req}.\overline{\mathsf{ans}}.x + \mathsf{cls}.\mathsf{no}\big)\\
m_3 &= \mathsf{rec}\, x.\big(\mathsf{req}.\overline{\mathsf{ans}}.x + \mathsf{cls}.\mathsf{yes} + \mathsf{req}.\mathsf{req}.x\big)\\
m_4 &= \mathsf{rec}\, x.\big(\mathsf{req}.\overline{\mathsf{ans}}.x + \mathsf{cls}.\mathsf{yes} + \mathsf{cls}.\mathsf{no}\big)
\end{aligned}
$$

Monitor m_3 may either behave like m_1 or non-deterministically terminate upon a serviced request, *i.e.,* $m_3 \triangleleft p \stackrel{\mathsf{req}}{\Longrightarrow} \mathsf{req}.m_3 \triangleleft p_1 \stackrel{\mathsf{ans}}{\longrightarrow} \mathsf{end} \triangleleft p$. Conversely, monitor m_4 non-deterministically returns verdict yes or no upon a cls action, *e.g.,* $m_4 \triangleleft p \stackrel{\mathsf{cls}}{\Longrightarrow} \mathsf{yes} \triangleleft \mathsf{nil}$ but also $m_4 \triangleleft p \stackrel{\mathsf{cls}}{\Longrightarrow} \mathsf{no} \triangleleft \mathsf{nil}$. ■

5 Correspondence

Although the techniques, algorithms and tools for statically checking system correctness *wrt.* μHML are well established [3, 5], there are situations where checking for system correctness at runtime through monitors is appealing and/or useful. Indeed, the aforementioned static techniques are prone to state-explosion problems in the setting of finite-state systems and are often not applicable to analyse infinite-state systems. In these cases, RV might offer a scalable compromise for determining system correctness. Alternatively, the system may not be available for analysis prior to deployment, and RV would constitute the only means of analysing system behaviour for correctness.

Our goal is therefore to establish a correspondence between the verdicts reached by monitors over an instrumented system from Section 4 and the properties specified using the logic of Section 3. In particular, we would like to relate acceptances (yes) and rejections (no) reached by a monitor m when monitoring a process p with satisfactions ($p \in [\![\varphi]\!]$) and violations ($p \notin [\![\varphi]\!]$) for that process *wrt.* some closed μHML formula, φ. This will, in turn, allow us to determine when a monitor m *represents* (in some precise sense) a property φ.

Example 7 Recall monitor m_1 from Example 6

$$m_1 = \mathsf{rec}\, x.\big(\mathsf{req}.\overline{\mathsf{ans}}.x + \mathsf{cls}.\mathsf{yes}\big)$$

Intuitively, one can argue that m_1 monitors for any *satisfactions* of the property

$$\varphi_7 = \mathsf{min}\, X.(\langle \mathsf{req}\rangle\langle \overline{\mathsf{ans}}\rangle X \vee \langle \mathsf{cls}\rangle \mathsf{tt})$$

This property describes processes that are able to perform a cls action after a number of serviced requests, *i.e.,* sequences of $\mathsf{req}.\overline{\mathsf{ans}}$ along one particular execution path. In fact, monitor m_1 produces a yes verdict for a computation labelled with a trace in the language $(\mathsf{req}.\mathsf{ans})^*.\mathsf{cls}$ from a process p, attesting that $p \in [\![\varphi_7]\!]$. Similarly, one can argue that m_2 from Example 6 monitors for *violations* of the property φ_4 from Example 3. The same *cannot* be said for

$$m_4 = \mathsf{rec}\, x.\big(\mathsf{req}.\overline{\mathsf{ans}}.x + \mathsf{cls}.\mathsf{yes} + \mathsf{cls}.\mathsf{no}\big)$$

again from Example 6, and φ_7 above. This is because, for some processes, *e.g.,* p from Example 1, m_4 may produce both verdicts yes and no for the same witness execution traces in $(\mathsf{req}.\mathsf{ans})^*.\mathsf{cls}$; this, in turn, leads to contradictions at a logical level since we cannot have both semantic judgements $p \in [\![\varphi_7]\!]$ and $p \notin [\![\varphi_7]\!]$. A similar argument applies to m_4 and m_2 from Example 6. ■

When instrumented with a process, a monitor may behave non-deterministically in other ways from the behaviour discussed in Example 7 for the case of m_4.

Example 8 Since the process drives the execution in a monitored system, non-deterministic behaviour from the side of the process may affect the detection capabilities of the instrumented monitor. For instance, we argued in Example 7 that m_1 monitors for property φ_7. When monitor m_1 is instrumented with process q from Example 1, m_1 may reach an acceptance verdict along various executions such as in the case of $m_1 \triangleleft q \stackrel{\mathsf{cls}}{\Longrightarrow} \mathsf{yes} \triangleleft \mathsf{nil}$. However, the monitored system may also exhibit the execution

$$m_1 \triangleleft q \stackrel{\mathsf{req}.\overline{\mathsf{ans}}}{\Longrightarrow} m_1 \triangleleft q_4 \qquad \text{where} \quad q_4 = \mathsf{rec}\, y.\mathsf{req}.\overline{\mathsf{ans}}.y$$

from which m_1 cannot issue an acceptance verdict for q, even though $q \in [\![\varphi_7]\!]$. ■

Example 9 Recall $m_3 = \mathsf{rec}\, x.\big(\mathsf{req}.\overline{\mathsf{ans}}.x + \mathsf{cls}.\mathsf{yes} + \mathsf{req}.\mathsf{req}.x\big)$ from Example 6. This monitor may sometimes flag an acceptance but also refrain from doing so when monitoring p from Example 1. This inconsistent behaviour can be observed *even when* p produces the same trace. *E.g.,*, for $t = \mathsf{req}.\mathsf{ans}.\mathsf{cls}$ we have $m_3 \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{yes} \triangleleft \mathsf{nil}$ but also $m_3 \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{end} \triangleleft \mathsf{nil}$. ■

There is however a key difference between the non-determinism exhibited by monitor m_4 in Example 7 and that discussed in Example 8 and Example 9. Specifically, in the latter two cases, the non-deterministic behaviour was between detections (*i.e.,* yes and no) and non-detections (*i.e.,* end or any other non-verdict state), which does not lead to any inconsistencies from a logical perspective. By contrast, non-determinism exhibited by monitor m_4 in Example 7 leads to logical contradictions. This difference provides a blueprint for the subsequent formal framework. Specifically, we investigate conditions that one could require for establishing the

correspondence between monitors and formulae. We start with Definition 2, which defines the criteria expected of a monitor m when it *monitors soundly for a property* φ, $\mathbf{smon}(m,\varphi)$, by requiring that acceptances (*resp.* rejections) imply satisfactions (*resp.* violations) for *every* monitored execution of a process p.

Definition 1 (Acceptance and Rejection Predicates)

1. $\mathbf{acc}(p,m) \stackrel{\mathbf{def}}{=} \exists t, p'.\ m \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{yes} \triangleleft p'$
2. $\mathbf{rej}(p,m) \stackrel{\mathbf{def}}{=} \exists t, p'.\ m \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p'$ ■

Definition 2 (Sound Monitoring) We say that a monitor m monitors soundly for the property represented by the formula φ, denoted as $\mathbf{smon}(m,\varphi)$, whenever for *all* processes $p \in \textsc{Proc}$ the following conditions hold:

1. $\mathbf{acc}(p,m)$ implies $p \in [\![\varphi]\!]$
2. $\mathbf{rej}(p,m)$ implies $p \notin [\![\varphi]\!]$ ■

We note that Definition 2 rules out contradicting verdicts by monitors. Whenever $\mathbf{smon}(m,\varphi)$ holds for some monitor m and formula φ, and there exists some process p where $\mathbf{acc}(p,m)$, it must be the case that $p \in [\![\varphi]\!]$ by Definition 2. Thus, from the logical satisfaction definition we have $\neg(p \notin [\![\varphi]\!])$ and by the contrapositive of Definition 2, we must also have $\neg\mathbf{rej}(p,m)$. A symmetric argument also applies for any process p where $\mathbf{rej}(p,m)$, from which $\neg\mathbf{acc}(p,m)$ follows.

Example 10 From Example 7 and Example 9, we can formally state and prove that $\mathbf{smon}(m_1,\varphi_7)$ and $\mathbf{smon}(m_3,\varphi_7)$. Moreover, from Example 3 and Example 6 we can also show that $\mathbf{smon}(m_2,\varphi_4)$.

For instance, recall that $m_2 = \mathsf{rec}\, x.\big(\mathsf{req}.\overline{\mathsf{ans}}.x + \mathsf{cls}.\mathsf{no}\big)$, which means that the only verdicts corresponding to logical satisfactions or logical violations that this monitor can produce are rejections. Moreover, from the semantics of Figure 4, each time a rejection is produced by m_2, this can only be the result of analysing an execution trace in the language $(\mathsf{req}.\overline{\mathsf{ans}})^*.\mathsf{cls}$. But then, from the logic semantics of Figure 3, any such execution trace is enough for the process to violate property $\varphi_4 = \mathsf{max}\, X.([\mathsf{req}][\overline{\mathsf{ans}}]X \wedge [\mathsf{cls}]\mathsf{ff})$.

Using Definition 2, we can also show that certain monitors do not soundly monitor for specific formulas, such as $\neg\mathbf{smon}(m_4,\varphi_7)$ and $\neg\mathbf{smon}(m_4,\varphi_4)$. ■

Sound monitoring is arguably the least requirement for relating a monitor with a logical property. Further to this, the obvious additional requirement would be to ask for the dual of Definition 2, *i.e., complete monitoring* for m and φ. Intuitively, this would state that for all p, $p \in [\![\varphi]\!]$ implies $\mathbf{acc}(p,m)$, and also that $p \notin [\![\varphi]\!]$ implies $\mathbf{rej}(p,m)$. However, such a requirement turns out to be too strong for a large part of the logic presented in Figure 3.

Example 11 Consider the core basic formula $\langle\alpha\rangle\mathsf{tt}$, describing processes that can perform action α. One could check that the simple monitor $\alpha.\mathsf{yes}$ satisfies the condition that $p \in [\![\varphi]\!]$ implies $\mathbf{acc}(p,m)$ for all p. However, for this same formula, there does *not* exist a sound monitor satisfying $p \notin [\![\varphi]\!]$ implies $\mathbf{rej}(p,m)$ for any arbitrary process p. We can show this by contradiction. Assume that one such (sound) monitor m exists. Since $\mathsf{nil} \notin [\![\langle\alpha\rangle\mathsf{tt}]\!]$ then we should have $\mathbf{rej}(\mathsf{nil},m)$. By Definition 1 and Proposition 1, this means that this particular monitor can produce the behaviour $m \Longrightarrow \mathsf{no}$ which, in turn, implies that $\mathbf{rej}(\alpha.\mathsf{nil},m)$ also holds

although, clearly, $\alpha.\mathsf{nil} \in [\![\langle\alpha\rangle\mathsf{tt}]\!]$. This makes m unsound, contradicting our initial assumption.

A similar, albeit dual, argument can be carried out for another core basic formula in μHML, namely $[\alpha]\mathsf{ff}$, describing the property of not being able to produce action α. Although there are sound monitors satisfying the condition $\forall p.\, p \notin [\![\varphi]\!]$ implies $\mathbf{rej}(p, m)$, there are none that also satisfy the other condition $\forall p.\, p \in [\![\varphi]\!]$ implies $\mathbf{acc}(p, m)$. ■

The core formulas discussed in Example 11 are enough evidence to convince us that requiring complete monitoring would limit correspondence to a trivial subset of the logic. In fact, it would limit it to the semantic subset tt and ff. In view of this, we define weaker forms of completeness, as stated below, where we require completeness *wrt.* either logical satisfactions or violations (but not both).

Definition 3 (Satisfaction, Violation, and Partially-Complete Monitoring)

$$\begin{array}{rcll} \mathbf{scmon}(m,\varphi) & \stackrel{\text{def}}{=} & \forall p.\, p \in [\![\varphi]\!] \text{ implies } \mathbf{acc}(p,m) & \text{(satisfaction complete)} \\ \mathbf{vcmon}(m,\varphi) & \stackrel{\text{def}}{=} & \forall p.\, p \notin [\![\varphi]\!] \text{ implies } \mathbf{rej}(p,m) & \text{(violation complete)} \\ \mathbf{cmon}(m,\varphi) & \stackrel{\text{def}}{=} & \mathbf{scmon}(m,\varphi) \text{ or } \mathbf{vcmon}(m,\varphi) & \text{(partially complete)} \end{array}$$ ■

Using the partially-complete monitoring predicate $\mathbf{cmon}(m, \varphi)$ of Definition 3 and the sound monitoring predicate of Definition 2, we can formalise our touchstone notion for monitor-formula correspondence. Specifically, Definition 4 below states that m *monitors* for formula φ, $\mathbf{mon}(m, \varphi)$, if it can do it *soundly*, and in a *partially-complete* manner, *i.e.,* if it is *either* satisfaction complete *or* violation complete as stated in Definition 3.

Definition 4 (Monitor-Formula Correspondence)

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

6 Monitorability

At an general level, *monitorability* is a property of a correctness specification describing the ability to be adequately analysed at runtime; it can also be lifted in pointwise fashion to sets of such specifications. This definition is thus fundamentally dependent on the monitoring setup assumed and the conditions that constitute an adequate runtime analysis. In this section we investigate the monitorability of our logic μHML using Definition 4 as our base notion for adequate monitoring.

Definition 5 (Monitorability) Formula φ is *monitorable* iff there exists a monitor m such that $\mathbf{mon}(m, \varphi)$. A logical language $\mathcal{L} \subseteq \mu\text{HML}$ is monitorable iff every $\varphi \in \mathcal{L}$ is monitorable. ■

We immediately note that not all logical formulae are monitorable. The following example gives evidence of this.

Example 12 Through the witness monitor outlined in Example 11, we can show that formulae $\langle\alpha\rangle\mathsf{tt}$ and $\langle\beta\rangle\mathsf{tt}$ are monitorable with a satisfaction complete monitor. However, the composite formula φ_8 defined below is *not*.

$$\varphi_8 = \langle\alpha\rangle\mathsf{tt}\wedge\langle\beta\rangle\mathsf{tt}$$

At an intuitive level, this is the case because once a monitor observes one of the actions required to check either sub-formula $\langle\alpha\rangle\mathsf{tt}$ or $\langle\beta\rangle\mathsf{tt}$, it cannot "go back in time" to observe another action and thus check for the other sub-formula.

Formally, we show this by arguing towards a contradiction. Assume that there exists some monitor $m.$ such that $\mathbf{mon}(m, \varphi_8)$. There are two sub-cases to consider.

- If m is satisfaction complete, then $\mathbf{acc}(\alpha.\mathsf{nil} + \beta.\mathsf{nil}, m)$ since $\alpha.\mathsf{nil} + \beta.\mathsf{nil} \in [\![\varphi_8]\!]$. By Proposition 1, m reaches verdict yes along one of the traces ϵ, α or β. If the trace is the empty trace ϵ, then m must also accept nil, which is unsound (since $\mathsf{nil} \notin [\![\varphi_8]\!]$). If the trace is α, m must also accept the process $\alpha.\mathsf{nil}$, which is also unsound ($\alpha.\mathsf{nil} \notin [\![\varphi_8]\!]$); the case for β is analogous.
- If m is violation complete then $\mathbf{rej}(\beta.\mathsf{nil}, m)$ since $\beta.\mathsf{nil} \notin [\![\varphi_8]\!]$. By Proposition 1, we either have $m \stackrel{\epsilon}{\Longrightarrow} \mathsf{no}$ or $m \stackrel{\beta}{\Longrightarrow} \mathsf{no}$ and for both cases we can argue that m also rejects process $\alpha.\mathsf{nil} + \beta.\mathsf{nil}$, which is unsound since $\alpha.\mathsf{nil} + \beta.\mathsf{nil} \in [\![\varphi_8]\!]$. ■

We now identify the following syntactic subset of μHML formulae called MHML, with the aim of showing that it is a monitorable subset of the logic. At an intuitive level, logical language consists of the safe and co-safe syntactic subsets of μHML, denoted as SHML and CHML respectively.

Definition 6 (Monitorable Logic) $\psi, \chi \in \text{MHML} \stackrel{\mathbf{def}}{=} \text{SHML} \cup \text{CHML}$ where:

$$\begin{array}{rllllll} \theta, \vartheta \in \text{SHML} ::= & \mathsf{tt} & \mid \mathsf{ff} & \mid [\alpha]\theta & \mid \theta\wedge\vartheta & \mid \mathsf{max}\, X.\theta & \mid X \\ \pi, \varpi \in \text{CHML} ::= & \mathsf{tt} & \mid \mathsf{ff} & \mid \langle\alpha\rangle\pi & \mid \pi\vee\varpi & \mid \mathsf{min}\, X.\pi & \mid X \end{array}$$

Example 13 Recall formulas φ_4 from Example 3 and φ_7 from Example 7. We have $\varphi_4 \in \text{SHML} \subseteq \text{MHML}$ and $\varphi_7 \in \text{CHML} \subseteq \text{MHML}$. However, φ_8 from Example 12 is not in MHML; this can be easily determined via a simple syntactic analysis. ■

Clearly $\text{MHML} \subseteq \mu\text{HML}$. To show that MHML is monitorable in the sense of Definition 5, we still need to find a representative monitor for every formula in MHML. Thus, we define a monitor synthesis function $(\!|-|\!)$ generating a monitor for each $\psi \in \text{MHML}$ and then show that $(\!|\psi|\!)$ is the witness monitor required by Definition 5 to demonstrate the monitorability of ψ.

Definition 7 (Monitor Synthesis)

$$(\!|\mathsf{ff}|\!) \stackrel{\mathbf{def}}{=} \mathsf{no} \qquad (\!|\mathsf{tt}|\!) \stackrel{\mathbf{def}}{=} \mathsf{yes} \qquad (\!|X|\!) \stackrel{\mathbf{def}}{=} x$$

$$(\!|[\alpha]\psi|\!) \stackrel{\text{def}}{=} \begin{cases} \alpha.(\!|\psi|\!) & \text{if } (\!|\psi|\!) \neq \mathsf{yes} \\ \mathsf{yes} & \text{otherwise} \end{cases} \qquad (\!|\langle\alpha\rangle\psi|\!) \stackrel{\text{def}}{=} \begin{cases} \alpha.(\!|\psi|\!) & \text{if } (\!|\psi|\!) \neq \mathsf{no} \\ \mathsf{no} & \text{otherwise} \end{cases}$$

$$(\!|\psi_1 \wedge \psi_2|\!) \stackrel{\text{def}}{=} \begin{cases} \mathsf{no} & \begin{cases} \text{if } (\!|\psi_1|\!) = \mathsf{no} \\ \text{or } (\!|\psi_2|\!) = \mathsf{no} \end{cases} \\ (\!|\psi_1|\!) & \text{if } (\!|\psi_2|\!) = \mathsf{yes} \\ (\!|\psi_2|\!) & \text{if } (\!|\psi_1|\!) = \mathsf{yes} \\ (\!|\psi_1|\!) + (\!|\psi_2|\!) & \text{otherwise} \end{cases} \qquad (\!|\psi_1 \vee \psi_2|\!) \stackrel{\text{def}}{=} \begin{cases} \mathsf{yes} & \begin{cases} \text{if } (\!|\psi_1|\!) = \mathsf{yes} \\ \text{or } (\!|\psi_2|\!) = \mathsf{yes} \end{cases} \\ (\!|\psi_1|\!) & \text{if } (\!|\psi_2|\!) = \mathsf{no} \\ (\!|\psi_2|\!) & \text{if } (\!|\psi_1|\!) = \mathsf{no} \\ (\!|\psi_1|\!) + (\!|\psi_2|\!) & \text{otherwise} \end{cases}$$

$$(\!|\mathsf{max}X.\psi|\!) \stackrel{\text{def}}{=} \begin{cases} \mathsf{rec}\, x.(\!|\psi|\!) & \text{if } (\!|\psi|\!) \neq \mathsf{yes} \\ \mathsf{yes} & \text{otherwise} \end{cases} \qquad (\!|\mathsf{min}X.\psi|\!) \stackrel{\text{def}}{=} \begin{cases} \mathsf{rec}\, x.(\!|\psi|\!) & \text{if } (\!|\psi|\!) \neq \mathsf{no} \\ \mathsf{no} & \text{otherwise} \end{cases}$$

A few comments are in order regarding Definition 7. We first note that the synthesis function $(\!|-|\!)$ is *compositional*; see *e.g.,* [32] for reasons why this is desirable. It also assumes a bijective mapping between the denumerable sets $X \in \text{LVAR}$ and $x \in \text{VARS}$; see synthesis for X, $\mathsf{max}\, X.\psi$ and $\mathsf{min}\, X.\psi$, where the logical variable X is converted to the process variable x. We also note that, although Definition 7 covers both sHML and cHML, the syntactic constraints of Definition 6 mean that synthesis for a formula ψ uses at most the first row and then either the first column (in the case of sHML) or the second column (in case of cHML). Finally, we note that the conditional cases used in synthesis of conjunctions, disjunctions, necessity formulas, possibility formulas, and maximal and minimal fixpoints are necessary to be able to generate sound and complete monitors, and handle logically equivalent formulae. We illustrate this in the next two examples.

Example 14 Consider the cHML formula $\langle\alpha\rangle\mathsf{tt}\vee\mathsf{ff}$ (which is logically equivalent to $\langle\alpha\rangle\mathsf{tt}$). A naive synthesis without the case checks of Definition 7 would generate the monitor $\alpha.\mathsf{yes} + \mathsf{no}$ which is not only redundant, but *unsound*. For instance, for process $p = \alpha.\mathsf{nil}$ we have both $\mathbf{acc}(p, \alpha.\mathsf{yes} + \mathsf{no})$ and $\mathbf{rej}(p, \alpha.\mathsf{yes} + \mathsf{no})$. Similar problems would manifest themselves for less obvious cases. Consider the formula cHML formula

$$\langle\alpha\rangle\mathsf{tt}\vee\langle\beta\rangle\mathsf{ff}\vee(\mathsf{min}\, X.\langle\alpha\rangle\mathsf{ff})\vee(\langle\alpha\rangle\mathsf{min}\, X.\mathsf{ff})$$

which also turns out to to be logically equivalent to $\langle\alpha\rangle\mathsf{tt}$ because $[\![\langle\beta\rangle\mathsf{ff}]\!] = [\![\mathsf{ff}]\!]$, as is $[\![\mathsf{min}\, X.\langle\alpha\rangle\mathsf{ff}]\!] = [\![\mathsf{ff}]\!] = [\![\langle\alpha\rangle\mathsf{min}\, X.\mathsf{ff}]\!]$. However, in all of these cases, the monitor synthesis of Definition 7 generates the (sound) monitor $\alpha.\mathsf{yes}$ due to its side conditions. ■

Example 15 Consider the cHML formula $\mathsf{tt}\vee\langle\alpha\rangle\mathsf{tt}$. A naive synthesis without the case checks of Definition 7 would generate the monitor $\mathsf{yes} + \alpha.\mathsf{yes}$ which is clearly *not* violation complete, but its *not* satisfaction complete either. In fact, one can easily see that $\mathsf{nil} \in [\![\mathsf{tt}\vee\langle\alpha\rangle\mathsf{tt}]\!]$ but $\mathbf{acc}(\mathsf{nil}, \mathsf{yes} + \alpha.\mathsf{yes})$ does not hold, because there is no trace t such that $\mathsf{yes} + \alpha.\mathsf{yes} \lhd \mathsf{nil} \stackrel{t}{\Longrightarrow} \mathsf{yes} \lhd \mathsf{nil}$. ■

Remark 2 The monitor synthesis algorithm of Definition 7 may generate nondeterministic monitors for certain formulas in mHML. Consider, for example, the formula

$$\varphi_{10} = (\langle\alpha\rangle\mathsf{tt})\vee(\langle\alpha\rangle\langle\beta\rangle\mathsf{tt}) \in \text{cHML}$$

From Definition 7 we have $(\!|\varphi_{10}|\!) = \alpha.\mathsf{yes}+\alpha.\beta.\mathsf{yes}$, where the generated monitor may sometimes refrain from producing an acceptance verdict for satisfying processes.

Specifically, for the process $\alpha.\mathsf{nil} \in [\![\varphi_{10}]\!]$ we have $\mathbf{acc}(\alpha.\mathsf{nil}, (\alpha.\mathsf{yes}+\alpha.\beta.\mathsf{yes}))$ because of the monitored execution $(\alpha.\mathsf{yes}+\alpha.\beta.\mathsf{yes}) \triangleleft (\alpha.\mathsf{nil}) \stackrel{\alpha}{\Longrightarrow} \mathsf{yes} \triangleleft \mathsf{nil}$. At the same time, we can also have the alternate monitored execution $(\alpha.\mathsf{yes} + \alpha.\beta.\mathsf{yes}) \triangleleft (\alpha.\mathsf{nil}) \stackrel{\alpha}{\Longrightarrow} \beta.\mathsf{yes} \triangleleft \mathsf{nil}$. Note, however that the generated monitor still adequately monitors for the formula, *i.e.,* $\mathbf{mon}(\alpha.\mathsf{yes} + \alpha.\beta.\mathsf{yes}, \varphi_{10})$.

Although we could have defined a more sophisticated monitor synthesis algorithm that handles this non-determinism (at the expense of obscuring further its intended meaning and complicating the ensuing proofs), we opted for a simpler translation that yields correct monitors while still preserving a degree of modularity. An alternative solution could also be to use the synthesis algorithm of Definition 7 and massage mHML formulas before performing the translation (*e.g.,* φ_{10} would be translated to the semantically equivalent formula $\langle\alpha\rangle(\mathsf{tt}\vee\langle\beta\rangle\mathsf{tt})$ which would then be synthesised to the monitor $\alpha.\mathsf{yes}$). ■

Theorem 1 (Monitorability) *$\varphi \in$ mHML implies φ is monitorable.*

To prove this theorem, we use $(\!|-|\!)$ to generate witnesses and show that for all $\varphi \in$ mHML, $\mathbf{mon}((\!|\varphi|\!), \varphi)$ holds. This proof is given in the dedicated Section 6.1. The reader may safely skip this subsection and proceed to Section 7 upon first reading. Before this, however, we discuss the import of this result.

Theorem 1 provides us with a simple syntactic check to determine whether a formula is monitorable; as shown earlier in Example 12, determining whether a formula is monitorable is in general non-trivial. The proof of Theorem 1 yields also a pleasing side effect: through Definition 7, we obtain an automatic monitor synthesis algorithm that is correct according to Definition 4.

Example 16 Since φ_4 from Example 3 is in mHML, we know it is monitorable. Moreover, we can generate the *correct* monitor $(\!|\varphi_4|\!) = \mathsf{rec}\, x.\big(\mathsf{req}.\overline{\mathsf{ans}}.x + \mathsf{cls}.\mathsf{no}\big) = m_2$ (from Example 6). Using similar reasoning, φ_7 (Example 7) is also monitorable, and a correct monitor for it is m_1 from Example 6. ■

6.1 Proving Monitorability

Using the monitor synthesis $(\!|-|\!)$ of Definition 7, Theorem 1 is proved in two steps. First we show that the witness monitors generated by $(\!|-|\!)$ are sound, according to Definition 2. In the second step, we show that they are partially complete, according to Definition 3.

We start with soundness. Expanding Definition 2(Monitor Soundness), we need to show that for all $\psi \in$ mHML, and for all $p \in$ Proc, we have:

$$\mathbf{acc}(p, (\!|\psi|\!)) \quad \text{implies} \quad p \in [\![\psi]\!] \tag{3}$$

$$\mathbf{rej}(p, (\!|\psi|\!)) \quad \text{implies} \quad p \notin [\![\psi]\!] \tag{4}$$

To simplify the technical exposition, we proceed by proving soundness for the two syntactic subsets, sHML and cHML, separately. We here present the proofs for the subset sHML; those for the cHML subset are similar and can be constructed by interested reader following a similar procedure.

For sHML, property (3) follows from Lemma 5. This lemma, in turn uses a number of auxiliary lemmata stated below.

Lemma 2 (Verdict Persistence) $v \stackrel{t}{\Longrightarrow} m$ *implies* $m = v$.

Proof By induction on the length of the transition sequence $v \stackrel{t}{\Longrightarrow} m$. □

Lemma 3 $\forall \theta \in \text{sHML}.\ (\!|\theta|\!) = \mathit{yes}$ *implies* $[\![\theta]\!] = [\![\mathsf{tt}]\!]$

Proof By structural induction on θ. Some of the main cases are:

$\theta = \theta_1 \wedge \theta_2$: By Definition 7 $(\!|\theta_1 \wedge \theta_2|\!) = \mathsf{yes}$ only when $(\!|\theta_1|\!) = \mathsf{yes}$ and $(\!|\theta_2|\!) = \mathsf{yes}$. By I.H. we obtain $[\![\theta_1]\!] = [\![\mathsf{tt}]\!]$ and $[\![\theta_2]\!] = [\![\mathsf{tt}]\!]$, from which $[\![\theta_1 \wedge \theta_2]\!] = [\![\mathsf{tt}]\!]$ follows.

$\theta = \mathsf{max}\, X.\vartheta$: By Definition 7 $(\!|\mathsf{max}\, X.\vartheta|\!) = \mathsf{yes}$ only when $(\!|\vartheta|\!) = \mathsf{yes}$. By I.H. we obtain $[\![\vartheta]\!] = [\![\mathsf{tt}]\!]$ from which the required result follows. □

Lemma 4 $\forall \theta \in \text{sHML}.\ (\!|\theta|\!) \stackrel{t}{\Longrightarrow} \mathit{yes}$ *implies* $(\!|\theta|\!) = \mathit{yes}$.

Proof By numerical induction on the length of the transition sequence $\stackrel{t}{\Longrightarrow}$.

$n = 0$: Immediate.

$n = k + 1$: We have $(\!|\theta|\!) \xrightarrow{\mu} m \stackrel{t'}{\Longrightarrow} \mathsf{yes}$. We proceed by case analysis of the structure of θ. We here outline one of the main cases:

$\theta = [\alpha]\vartheta$: From Definition 7 we have two sub-cases to consider:

$(\!|\vartheta|\!) = \mathsf{yes}$: By Definition 7, this implies $(\!|[\alpha]\vartheta|\!) = \mathsf{yes}$ as required.

$(\!|\vartheta|\!) \neq \mathsf{yes}$: By Definition 7, his implies $(\!|[\alpha]\vartheta|\!) = \alpha.(\!|\vartheta|\!)$. We can therefore rewrite $(\!|\theta|\!) \xrightarrow{\mu} m \stackrel{t'}{\Longrightarrow} \mathsf{yes}$ as $\alpha.(\!|\vartheta|\!) \xrightarrow{\alpha} (\!|\vartheta|\!)$ and $(\!|\vartheta|\!) \stackrel{t'}{\Longrightarrow} \mathsf{yes}$ where μ could have only been α. However, by $(\!|\vartheta|\!) \stackrel{t'}{\Longrightarrow} \mathsf{yes}$ and I.H. we know that $(\!|\vartheta|\!) = \mathsf{yes}$ which gives us a contradiction. □

Lemma 5 $\forall \theta \in \text{sHML}, p \in \text{Proc}.\ \mathit{acc}(p, (\!|\theta|\!))$ *implies* $p \in [\![\theta]\!]$

Proof By expanding the definition of $\mathbf{acc}(p, (\!|\theta|\!))$, and then by Proposition 1, we know that there exists a trace t such that $(\!|\theta|\!) \stackrel{t}{\Longrightarrow} \mathsf{yes}$. By Lemma 4 we know that $(\!|\theta|\!) = \mathsf{yes}$ and subsequently, by Lemma 3, we obtain $[\![\theta]\!] = [\![\mathsf{tt}]\!]$, from which $p \in [\![\theta]\!]$ follows trivially. □

For sHML, property (4) follows from Lemma 8. This lemma, again uses a number of auxiliary lemmata, some of which have already been presented; the rest are stated below.

Lemma 6 $\forall \theta \in \text{sHML}.\ p' \notin [\![\theta]\!]$ *and* $p \xrightarrow{\tau} p'$ *implies* $p \notin [\![\theta]\!]$.

Proof By structural induction on θ. □

Lemma 7 $\forall \theta \in \text{sHML}.\ (\!|\theta|\!) = \mathit{no}$ *implies* $[\![\theta]\!] = [\![\mathsf{ff}]\!]$

Proof By structural induction on θ. Most cases are immediate, where the only non-trivial case is:

$\theta = \theta_1 \wedge \theta_2$: By Definition 7 $(\!|\theta_1 \wedge \theta_2|\!) = \mathsf{no}$ only when $(\!|\theta_1|\!) = \mathsf{no}$ or $(\!|\theta_2|\!) = \mathsf{no}$. Without loss of generality, assume the former, *i.e.,* $(\!|\theta_1|\!) = \mathsf{no}$. By I.H. we obtain $[\![\theta_1]\!] = [\![\mathsf{ff}]\!]$ and thus by Figure 3 we deduce $[\![\theta_1 \wedge \theta_2]\!] = [\![\theta_1]\!] \cap [\![\theta_2]\!] = [\![\mathsf{ff}]\!] \cap [\![\theta_2]\!] = \emptyset \cap [\![\theta_2]\!] = \emptyset = [\![\mathsf{ff}]\!]$ as required. □

Lemma 8 $\forall \theta \in \text{sHML}, p \in \text{Proc}.\ \textit{rej}(p, (\!|\theta|\!))$ *implies* $p \notin [\![\theta]\!]$

Proof By expanding $\mathbf{rej}(p, (\!|\theta|\!))$, we know that there exists a trace t such that $(\!|\theta|\!) \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p'$. The proof thus proceeds by numerical induction on the length of the transition sequence $\stackrel{t}{\Longrightarrow}$ and then by induction on the structure of θ.

$n = 0$: We thus know that $(\!|\theta|\!) = \mathsf{no}$. By Lemma 7 we obtain $[\![\theta]\!] = [\![\mathsf{ff}]\!] = \emptyset$, from which the required result follows.

$n = k + 1$: We have $(\!|\theta|\!) \triangleleft p \xrightarrow{\mu} m'' \triangleleft p'' \stackrel{t'}{\Longrightarrow} \mathsf{no} \triangleleft p'$. The main sub-cases are:

- $\theta = \mathsf{tt}$: Thus $(\!|\theta|\!) = \mathsf{yes}$. This however contradicts Lemma 2, since it can never be the case that a yes monitor transitions to a monitor that constitutes a no verdict.
- $\theta = [\alpha]\vartheta$: By Definition 7, we have two sub-cases. If $(\!|\theta|\!) = \mathsf{yes}$ then we get a contradiction as in the case for $\theta = \mathsf{tt}$. The other sub-case is where $(\!|\theta|\!) = \alpha.(\!|\vartheta|\!)$. We have three sub-cases to consider:
  - $\alpha.(\!|\vartheta|\!) \triangleleft p \xrightarrow{\tau} \alpha.(\!|\vartheta|\!) \triangleleft p'' \stackrel{t'}{\Longrightarrow} \mathsf{no} \triangleleft p'$ because $p \xrightarrow{\tau} p''$. By I.H. and $\alpha.(\!|\vartheta|\!) \triangleleft p'' \stackrel{t'}{\Longrightarrow} \mathsf{no} \triangleleft p'$ we obtain $p'' \notin [\![[\alpha]\vartheta]\!]$, from which $p \notin [\![[\alpha]\vartheta]\!]$ follows by $p \xrightarrow{\tau} p''$ and Lemma 6.
  - $\alpha.(\!|\vartheta|\!) \triangleleft p \xrightarrow{\beta} \mathsf{end} \triangleleft p'' \stackrel{t'}{\Longrightarrow} \mathsf{no} \triangleleft p'$. We get a contradiction by $\mathsf{end} \stackrel{t'}{\Longrightarrow} \mathsf{no}$ and Lemma 2.
  - $\alpha.(\!|\vartheta|\!) \triangleleft p \xrightarrow{\alpha} (\!|\vartheta|\!) \triangleleft p'' \stackrel{t'}{\Longrightarrow} \mathsf{no} \triangleleft p'$ because $p \xrightarrow{\alpha} p''$. By I.H. we know that $p \notin [\![\vartheta]\!]$ and by $p \xrightarrow{\alpha} p''$ we deduce that $p \notin [\![[\alpha]\vartheta]\!]$.
- $\theta = \theta_1 \wedge \theta_2$: By Definition 7, we have four sub-cases:
  - $(\!|\theta_1 \wedge \theta_2|\!) = \mathsf{no}$ because $(\!|\theta_1|\!) = \mathsf{no}$ or $(\!|\theta_2|\!) = \mathsf{no}$. By Lemma 7 we obtain $[\![\theta]\!] = [\![\mathsf{ff}]\!] = \emptyset$, from which the required result follows.
  - $(\!|\theta_1 \wedge \theta_2|\!) = (\!|\theta_1|\!)$ because $(\!|\theta_2|\!) = \mathsf{yes}$. By I.H. we know $p \notin [\![\theta_1]\!]$. Moreover, by $(\!|\theta_2|\!) = \mathsf{yes}$ and Lemma 3 we know $[\![\theta_2]\!] = [\![\mathsf{tt}]\!]$, and thus $[\![\theta_1 \wedge \theta_2]\!] = [\![\theta_1]\!]$. Thus $p \notin [\![\theta_1 \wedge \theta_2]\!]$ follows.
  - $(\!|\theta_1 \wedge \theta_2|\!) = (\!|\theta_2|\!)$ because $(\!|\theta_1|\!) = \mathsf{yes}$. Analogous.
  - $(\!|\theta_1 \wedge \theta_2|\!) = (\!|\theta_1|\!) + (\!|\theta_2|\!)$. We have four further sub-cases to consider:
    - $(\!|\theta_1|\!) + (\!|\theta_2|\!) \triangleleft p \xrightarrow{\tau} (\!|\theta_1|\!) + (\!|\theta_2|\!) \triangleleft p'' \stackrel{t'}{\Longrightarrow} \mathsf{no} \triangleleft p'$ because $p \xrightarrow{\tau} p''$. By I.H. and $(\!|\theta_1|\!) + (\!|\theta_2|\!) \triangleleft p'' \stackrel{t'}{\Longrightarrow} \mathsf{no} \triangleleft p'$ we obtain $p'' \notin [\![\theta_1 \wedge \theta_2]\!]$, from which $p \notin [\![\theta_1 \wedge \theta_2]\!]$ follows by $p \xrightarrow{\tau} p''$ and Lemma 6.
    - $(\!|\theta_1|\!) + (\!|\theta_2|\!) \triangleleft p \xrightarrow{\beta} \mathsf{end} \triangleleft p'' \stackrel{t'}{\Longrightarrow} \mathsf{no} \triangleleft p'$. We get a contradiction by $\mathsf{end} \stackrel{t'}{\Longrightarrow} \mathsf{no}$ and Lemma 2.
    - $(\!|\theta_1|\!) + (\!|\theta_2|\!) \triangleleft p \xrightarrow{\mu} m \triangleleft p'' \stackrel{t'}{\Longrightarrow} \mathsf{no} \triangleleft p'$ because $(\!|\theta_1|\!) \triangleleft p \xrightarrow{\mu} m \triangleleft p''$. We can thus construct the transition sequence $(\!|\theta_1|\!) \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p'$ from which we obtain, by I.H., $p \notin [\![\theta_1]\!]$ and hence $p \notin [\![\theta_1 \wedge \theta_2]\!]$.
    - $(\!|\theta_1|\!) + (\!|\theta_2|\!) \triangleleft p \xrightarrow{\mu} m \triangleleft p'' \stackrel{t'}{\Longrightarrow} \mathsf{no} \triangleleft p'$ because $(\!|\theta_2|\!) \triangleleft p \xrightarrow{\mu} m \triangleleft p''$. Analogous.
- $\theta = \mathsf{max}\, X.\vartheta$: By Definition 7, we have two sub-cases. If $(\!|\theta|\!) = \mathsf{yes}$ then we get a contradiction as in the case for $\theta = \mathsf{tt}$. The other sub-case is where $(\!|\theta|\!) = \mathsf{rec}\, x.(\!|\vartheta|\!)$. Again we have two sub-cases to consider:
  - $\mathsf{rec}\, x.(\!|\vartheta|\!) \triangleleft p \xrightarrow{\tau} \mathsf{rec}\, x.(\!|\vartheta|\!) \triangleleft p'' \stackrel{t'}{\Longrightarrow} \mathsf{no} \triangleleft p'$ because $p \xrightarrow{\tau} p''$. It follows I.H. and Lemma 6, as in the cases above.

– $\mathsf{rec}\, x.(\!|\vartheta|\!) \lhd p \xrightarrow{\tau} (\!|\vartheta|\!)[\mathsf{rec}\, x.(\!|\vartheta|\!)/x] \lhd p \overset{t'}{\Longrightarrow} \mathsf{no} \lhd p'$. By I.H. we obtain $p \notin [\![\vartheta[\mathsf{max}\, X.\vartheta/X]]\!]$ and, since $[\![\mathsf{max}\, X.\vartheta]\!] = [\![\vartheta[\mathsf{max}\, X.\vartheta/X]]\!]$, the required result follows. □

Although we can prove the *resp.* lemmata for cHML in analogous fashion, we just state them here. From these intermediary results, Proposition 2 follows.

Lemma 9 $\forall \pi \in \text{cHML}, p \in \text{Proc}.\ \boldsymbol{acc}(p, (\!|\pi|\!))$ *implies* $p \in [\![\pi]\!]$

Lemma 10 $\forall \pi \in \text{cHML}, p \in \text{Proc}.\ \boldsymbol{rej}(p, (\!|\pi|\!))$ *implies* $p \notin [\![\pi]\!]$

Proposition 2 (Monitorability Soundness) $\forall \psi \in \text{mHML}.\ \boldsymbol{smon}((\!|\psi|\!), \psi)$

Proof Immediate from Lemma 5, Lemma 8, Lemma 9 and Lemma 10.

To prove partial completeness of mHML, we again tackle the sub-languages sHML and cHML separately, which allows us to prove finer results. Namely, we can show that all sHML formulae are violation complete *wrt.* their synthesised monitors of Definition 7, and that all cHML formulae are satisfaction complete. Once again, we provide full details for sHML, and leave the analogous proofs for the cHML case for the interested reader.

Lemma 11 $\forall \theta \in \text{sHML}. p \in \text{Proc}.\ p \notin [\![\theta]\!]$ *implies* $\boldsymbol{rej}(p, (\!|\theta|\!))$

Proof Following [2], the semantic definition of sHML can be given an alternative coinductive characterisation as the largest satisfiability relation satisfying the implications of Table 1 in [2]. In our specific case, in order to prove the required statement, it suffices to show that the relation

$$\mathcal{R} \overset{\text{def}}{=} \left\{ (p, \theta) \mid \quad \not\exists t \in \text{Act}^*, p' \in \text{Proc}.\ [\![\theta]\!] \lhd p \overset{t}{\Longrightarrow} \mathsf{no} \lhd p' \right\} \tag{5}$$

is a satisfiability relation. In effect, this proves the contrapositive of the statement we want to show, namely:

$$\neg \mathbf{rej}(p, [\![\theta]\!]) \quad \text{implies} \quad p \in [\![\theta]\!]$$

The proof proceeds by induction on the structure of θ. The main cases are:

$\theta = \theta_1 \wedge \theta_2$: From definition (5), we know that

$$\not\exists t \in \text{Act}^*, p' \in \text{Proc}.\ [\![\theta_1 \wedge \theta_2]\!] \lhd p \overset{t}{\Longrightarrow} \mathsf{no} \lhd p'. \tag{6}$$

To show that $\mathcal{R}$ is a satisfiability relation, we have to show that $(p, \theta_1) \in \mathcal{R}$ and $(p, \theta_2) \in \mathcal{R}$. By Definition 7, we have three sub-cases to consider:

– $(\!|\theta_1 \wedge \theta_2|\!) = (\!|\theta_1|\!)$ because $(\!|\theta_2|\!) = \mathsf{yes}$. From (6) we know $\not\exists t \in \text{Act}^*, p' \in \text{Proc}.\ [\![\theta_1]\!] \lhd p \overset{t}{\Longrightarrow} \mathsf{no} \lhd p'$, hence $(p, \theta_1) \in \mathcal{R}$. Also, since $(\!|\theta_2|\!) = \mathsf{yes}$, Lemma 2 states that this verdict is persistent. Hence we also know that $\not\exists t \in \text{Act}^*, p' \in \text{Proc}.\ [\![\theta_2]\!] \lhd p \overset{t}{\Longrightarrow} \mathsf{no} \lhd p'$, which means that $(p, \theta_2) \in \mathcal{R}$ as required.

– $(\!|\theta_1 \wedge \theta_2|\!) = (\!|\theta_2|\!)$ because $(\!|\theta_1|\!) = \mathsf{yes}$. Analogous.

– $(\!|\theta_1 \wedge \theta_2|\!) = (\!|\theta_1|\!) + (\!|\theta_2|\!)$. From (6) we know $\nexists t \in \text{Act}^*, p' \in \text{Proc}.\ [\![\theta_1]\!] \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p'$ and $\nexists t \in \text{Act}^*, p' \in \text{Proc}.\ [\![\theta_2]\!] \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p'$ which imply $(p, \theta_1) \in \mathcal{R}$ and $(p, \theta_2) \in \mathcal{R}$ as required.

$\theta = [\alpha]\vartheta$: From definition (5), we know that

$$\nexists t \in \text{Act}^*, p' \in \text{Proc}.\ [\![[\alpha]\vartheta]\!] \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p'. \tag{7}$$

According to [2], to show that $\mathcal{R}$ is a satisfiability relation, we have to show the following implication holds: $p \stackrel{\alpha}{\Longrightarrow} p'$ implies $(p', \vartheta) \in \mathcal{R}$. By Definition 7, we have two sub-cases to consider:

– $(\!|[\alpha]\vartheta|\!) = \mathsf{yes}$ because $(\!|\vartheta|\!) = \mathsf{yes}$. By Lemma 2 we know that, for any p', $\nexists t \in \text{Act}^*, p'' \in \text{Proc}.\ \mathsf{yes} \triangleleft p' \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p''$ which, by $(\!|\vartheta|\!) = \mathsf{yes}$ implies $(p', \vartheta) \in \mathcal{R}$ for any p'. The implication $p \stackrel{\alpha}{\Longrightarrow} p'$ implies $(p', \vartheta) \in \mathcal{R}$ thus holds trivially.

– $(\!|[\alpha]\vartheta|\!) = \alpha.[\![\vartheta]\!]$. By (7) we know $\nexists t \in \text{Act}^*, p' \in \text{Proc}.\ \alpha.[\![\vartheta]\!] \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p'$ which either implies:
  – $p \stackrel{\alpha}{\not\Longrightarrow}$, in which case the implication holds trivially.
  – $\alpha.[\![\vartheta]\!] \triangleleft p \stackrel{\alpha}{\Longrightarrow} [\![\vartheta]\!] \triangleleft p'$ where $p \stackrel{\alpha}{\Longrightarrow} p'$ and $\nexists t' \in \text{Act}^*, p'' \in \text{Proc}.\ [\![\vartheta]\!] \triangleleft p' \stackrel{t'}{\Longrightarrow} \mathsf{no} \triangleleft p''$, which by I.H., implies that for any such p', we have $(p', \vartheta) \in \mathcal{R}$ as the implication requires.

$\theta = \mathsf{max}\, X.\vartheta$: From definition (5), we know that

$$\nexists t \in \text{Act}^*, p' \in \text{Proc}.\ [\![\mathsf{max}\, X.\vartheta]\!] \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p'. \tag{8}$$

According to [2], to show that $\mathcal{R}$ is a satisfiability relation, we have to show that $(p, \vartheta[\mathsf{max}\, X.\vartheta/X]) \in \mathcal{R}$. By Definition 7, we have two sub-cases to consider:

– $(\!|\mathsf{max}\, X.\vartheta|\!) = \mathsf{yes}$ because $(\!|\vartheta|\!) = \mathsf{yes}$. This means that $(\!|\vartheta[\mathsf{max}\, X.\vartheta/X]|\!) = \mathsf{yes}$ as well, which, by Lemma 2 implies that

$$\nexists t \in \text{Act}^*, p' \in \text{Proc}.\ (\!|\vartheta[\mathsf{max}\, X.\vartheta/X]|\!) \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p'.$$

Hence $(p, \vartheta[\mathsf{max}\, X.\vartheta/X]) \in \mathcal{R}$ as required.

– $(\!|\mathsf{max}\, X.\vartheta|\!) = \mathsf{rec}\, x.[\![\vartheta]\!]$. From (8) we obtain

$$\nexists t \in \text{Act}^*, p' \in \text{Proc}.\ (\!|\vartheta|\!)[(\!|\mathsf{max}\, X.\vartheta|\!)/x] \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p'.$$

Hence $(p, \vartheta[\mathsf{max}\, X.\vartheta/X]) \in \mathcal{R}$ as required. □

Lemma 12 $\forall \pi \in \text{cHML}.p \in \text{Proc}.\ p \in [\![\pi]\!]$ *implies* $\boldsymbol{acc}(p, (\!|\pi|\!))$ □

Proposition 3 (Monitorability Partial Completeness)

$$\forall \psi \in \text{mHML}.\ \boldsymbol{cmon}((\!|\psi|\!), \psi)$$

Proof Follows from Lemma 11, Lemma 12 and Definition 3. □

Following these results, Theorem 1 follows.

Theorem 1. (Monitorability). $\varphi \in \text{mHML}$ *implies* φ *is monitorable.*

Proof Immediate from Proposition 2 and Proposition 3. □

7 Expressiveness

The results obtained in Section 6 beg the question of whether mHML is the *largest* monitorable subset of μHML. One way to provide an answer to this question would be to show that, in some sense, every monitor corresponds to a formula in mHML according to Definition 4. However, this approach quickly runs into problems since there are monitors, such as $\mathsf{yes} + \mathsf{no}$, that make little sense from the point of view of Definition 4.

To this end, we prove a general (and perhaps surprising) result. Theorem 2 below asserts that multi-verdict monitors are necessarily *unsound*, at least *wrt.* properties defined over processes (as opposed to logics defined over other domains such as traces, *e.g.,* [17,11,21]).

Theorem 2 (Multi-verdict Monitors) *For all monitors* $m \in \textsc{Mon}$

$$\big(\exists t, u \in \textsc{Act}^*.\ m \stackrel{t}{\Longrightarrow} \mathsf{yes} \text{ and } m \stackrel{u}{\Longrightarrow} \mathsf{no}\big) \quad \text{implies} \quad \nexists \varphi \in \mu\text{HML}.\ \mathbf{mon}(m, \varphi).$$

Proof By contradiction. Assume that there exists a logical formula $\varphi \in \mu$HML such that $\mathbf{mon}(m, \varphi)$. From the traces t and u, we can construct the obvious representative processes for each trace using the construction:

$$[\![\alpha.t]\!]_{\mathbf{trc2proc}} = \alpha.([\![t]\!]_{\mathbf{trc2proc}}) \qquad [\![\epsilon]\!]_{\mathbf{trc2proc}} = \mathsf{nil}$$

We can therefore also construct the process $p = ([\![t]\!]_{\mathbf{trc2proc}}) + ([\![u]\!]_{\mathbf{trc2proc}})$, where clearly $p \stackrel{t}{\Longrightarrow} \mathsf{nil}$ and $p \stackrel{u}{\Longrightarrow} \mathsf{nil}$. Using the assumptions $m \stackrel{t}{\Longrightarrow} \mathsf{yes}$, $m \stackrel{u}{\Longrightarrow} \mathsf{no}$ and Proposition 1 we can thus deduce $m \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{yes} \triangleleft \mathsf{nil}$ $m \triangleleft p \stackrel{u}{\Longrightarrow} \mathsf{no} \triangleleft \mathsf{nil}$. We therefore have $\mathbf{acc}(p, m)$ and $\mathbf{rej}(p, m)$, and by monitor soundness (Definition 2), that $p \in [\![\varphi]\!]$ and $p \notin [\![\varphi]\!]$. This is clearly a contradiction. □

This result also implies that, in order to answer the aforementioned question of whether for every sensible monitor m there exists a formula $\varphi \in$ mHML that corresponds to it, $\mathbf{mon}(m, \varphi)$, it suffices to focus on *uni-verdict* monitors that flag either acceptances or rejections (but not both). In fact, a closer inspection of the synthesis algorithm of Definition 7 reveals that all the monitors generated using it are, in fact, uni-verdict.

We further restrict the set of what we deem to be sensible monitors in terms of how verdicts are specified. In particular, since monitor verdicts are expected to be definite, it makes little sense to place them at the top level of an external choice, *e.g.,* $\mathsf{no} + m$ since the meaning of such verdicts is unclear. In fact, a similarly problematic monitor has already been discussed in Example 15. We therefore consider all such monitors as *ill-formed* and restrict our expressivity analysis to *well-formed* monitors, *i.e.,* the complement of ill-formed monitors. It is not hard to see that the monitors generated by Definition 7 are, in fact, all well-formed; this can be show by a simple induction on the structure of the formula.

We partition the set of well-formed, uni-verdict monitors into the obvious classes: *acceptance monitors*, AMon (using verdict yes), and *rejection monitors*, RMon (using no). In what follows, we focus our technical development on one monitor class, in the knowledge that the corresponding development for the other class is analogous.

Definition 8 (Rejection Expressive-Complete) A subset $\mathcal{L} \subseteq \mu\text{HML}$ is *expressive-complete wrt.* (well-formed) rejection monitors iff for all $m \in \text{RMon}$, there exists some $\varphi \in \mathcal{L}$ such that $\mathbf{mon}(m, \varphi)$. ■

We show that the language sHML (Definition 6) is rejection expressive-complete. We do so with the aid of a mapping function $\langle\!\langle - \rangle\!\rangle$ from a rejection monitor to a corresponding formula in sHML defined below. Definition 9 is fairly straightforward, thanks to the fact that we only need to contend with a single verdict. Again, it assumes a bijective mapping between the denumerable sets LVar and Vars (as in the case of Definition 7). The mapping function is defined inductively on the structure of m where we note that (i) no translation is given for the monitor yes (since these are rejection monitors) and (ii) the base case end is mapped to formula tt, which contrasts with the mapping used in Definition 7.

Definition 9 (Rejection Monitors to sHML Formulae)

$$\langle\!\langle \mathsf{no} \rangle\!\rangle \stackrel{\mathbf{def}}{=} \mathsf{ff} \qquad \langle\!\langle \mathsf{end} \rangle\!\rangle \stackrel{\mathbf{def}}{=} \mathsf{tt} \qquad \langle\!\langle x \rangle\!\rangle \stackrel{\mathbf{def}}{=} X$$

$$\langle\!\langle \alpha.m \rangle\!\rangle \stackrel{\mathbf{def}}{=} [\alpha]\langle\!\langle m \rangle\!\rangle \qquad \langle\!\langle m + n \rangle\!\rangle \stackrel{\mathbf{def}}{=} \langle\!\langle m \rangle\!\rangle \wedge \langle\!\langle n \rangle\!\rangle \qquad \langle\!\langle \mathsf{rec}\, x.m \rangle\!\rangle \stackrel{\mathbf{def}}{=} \mathsf{max}\, X.\langle\!\langle m \rangle\!\rangle$$

In the case of rejection monitors, RMon, the monitorability constraint, Definition 4, relating a monitor m with a formula φ can be simplified to the condition below, since the *resp.* monitors never produce an acceptance.

$$\forall p \in \text{Proc}.\ \mathbf{rej}(p, m) \quad \text{iff} \quad p \notin [\![\varphi]\!] \tag{9}$$

The required result, namely Proposition 4, follows from Lemma 13, the proof of which may be skipped upon first reading.

Lemma 13 *For all rejection monitors $m \in \text{RMon}$ and arbitrary process $p \in \text{Proc}$ we have $\mathbf{rej}(p, m)$ iff $p \notin [\![\langle\!\langle m \rangle\!\rangle]\!]$*

Proof For the *only-if* case we need to show that

$$m \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p' \quad \text{implies} \quad p \notin [\![\langle\!\langle m \rangle\!\rangle]\!]$$

The proof proceeds by numerical induction on n, the length of the transition sequence $\stackrel{t}{\Longrightarrow}$ and then by induction on the structure of m. The main cases are:

$n = 0$: We necessarily have that $m = \mathsf{no}$. By Definition 9 $\langle\!\langle \mathsf{no} \rangle\!\rangle = \mathsf{ff}$ from which $p \notin [\![\mathsf{ff}]\!] = \emptyset$ follows.

$n = k + 1$: We know that $m \triangleleft p \xrightarrow{\mu} m' \triangleleft p'' \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p'$. The main cases are:

$m = \alpha.n$: From the structure of the monitor, we know

$$\alpha.n \triangleleft p \stackrel{\alpha}{\Longrightarrow} n \triangleleft p'' \quad \text{where } p \stackrel{\alpha}{\Longrightarrow} p'' \tag{10}$$

$$n \triangleleft p' \stackrel{t'}{\Longrightarrow} \mathsf{no} \triangleleft p \tag{11}$$

By (11) and I.H. we know $p'' \notin [\![\langle\!\langle n \rangle\!\rangle]\!]$ and by $p \stackrel{\alpha}{\Longrightarrow} p''$ from (10) we can conclude that $p \notin [\![[\alpha]\langle\!\langle n \rangle\!\rangle]\!] = [\![\langle\!\langle \alpha.n \rangle\!\rangle]\!]$.

$m = n_1 + n_2$: From the structure of the monitor, we know that we have

$$\text{either } n_1 \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p' \qquad\qquad \text{or } n_2 \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p'$$

If $n_1 \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p'$, then by I.H. we obtain $p \notin [\![\langle\!\langle n_1 \rangle\!\rangle]\!]$, which implies $p \notin [\![\langle\!\langle n_1 \rangle\!\rangle \wedge \langle\!\langle n_2 \rangle\!\rangle]\!] = [\![\langle\!\langle n_1 + n_2 \rangle\!\rangle]\!]$. The sub-case for $n_2 \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p'$ is analogous.

$m = \mathsf{rec}\, x.n$: From the structure of the monitor, we know

$$\mathsf{rec}\, x.n \triangleleft p \stackrel{\tau}{\Longrightarrow} n[\mathsf{rec}\, x.n/x] \triangleleft p'' \quad \text{where } p \Longrightarrow p'' \tag{12}$$

$$n[\mathsf{rec}\, x.n/x] \triangleleft p'' \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p' \tag{13}$$

In particular, for (12), the instrumentation rules of 4 prohibit process p from performing any visible actions until at least $\mathsf{rec}\, x.n$ unfolds (using a τ-transition). From (13) and $p \Longrightarrow p''$ of (12) we can construct the transition sequence

$$n[\mathsf{rec}\, x.n/x] \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p' \tag{14}$$

Where (14) uses (strictly) one less transition than $\mathsf{rec}\, x.n \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p'$. Hence, by the I.H. we obtain $p \notin [\![\langle\!\langle n[\mathsf{rec}\, x.n/x] \rangle\!\rangle]\!] = [\![\langle\!\langle n \rangle\!\rangle[\mathsf{max}\, X.\langle\!\langle n \rangle\!\rangle/X]]\!]$. Now, by Definition 9 and $\langle\!\langle \mathsf{rec}\, x.n \rangle\!\rangle = \mathsf{max}\, X.\langle\!\langle n \rangle\!\rangle$, the required result follows from the fact that $[\![\mathsf{max}\, X.\langle\!\langle n \rangle\!\rangle]\!] = [\![\langle\!\langle n \rangle\!\rangle[\mathsf{max}\, X.\langle\!\langle n \rangle\!\rangle/X]]\!]$.

For the *if* case we need to show that

$$p \notin [\![\langle\!\langle m \rangle\!\rangle]\!] \quad \text{implies} \quad m \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p' \text{ for some } t$$

We exploit the fact that our monitors are all assumed to be guarded, and prove the above statement by induction on the structure of m *and* the number of consecutive top-level $\mathsf{rec}\, x.m$ constructs a monitor has. We outline the main cases below:

$m = \alpha.n$: By Definition 9 we know $p \notin [\![[\alpha]\langle\!\langle n \rangle\!\rangle]\!]$, which implies that

$$p \stackrel{\alpha}{\Longrightarrow} p' \quad \text{for some } p' \tag{15}$$

$$p' \notin [\![\langle\!\langle n \rangle\!\rangle]\!] \tag{16}$$

By (16) and I.H. we obtain $n \triangleleft p' \stackrel{t'}{\Longrightarrow} \mathsf{no} \triangleleft p''$ for some t', p''. Thus by $p \stackrel{\alpha}{\Longrightarrow} p'$ from (15) we derive $\alpha.n \triangleleft p \stackrel{\alpha t'}{\Longrightarrow} \mathsf{no} \triangleleft p''$ as required.

$m = \mathsf{rec}\, x.n$: By Definition 9 we know $p \notin [\![\mathsf{max}\, X.\langle\!\langle n \rangle\!\rangle]\!] = [\![\langle\!\langle n \rangle\!\rangle[\mathsf{max}\, X.\langle\!\langle n \rangle\!\rangle/X]]\!]$. Note that $\langle\!\langle n[\mathsf{rec}\, x.n/x] \rangle\!\rangle = \langle\!\langle n \rangle\!\rangle[\mathsf{max}\, X.\langle\!\langle n \rangle\!\rangle/X]$, and since $n[\mathsf{rec}\, x.n/x]$ must have fewer top-level consecutive $\mathsf{rec}\, y.-$ constructs than $\mathsf{rec}\, x.n$ (processes and monitors are assumed to be guarded), we can use $p \notin [\![\langle\!\langle n \rangle\!\rangle[\mathsf{max}\, X.\langle\!\langle n \rangle\!\rangle/X]]\!]$ and I.H. to obtain $n[\mathsf{rec}\, x.n/x] \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p'$ for some t, p'. From this, we can derive $\mathsf{rec}\, x.n \triangleleft p \stackrel{t}{\Longrightarrow} \mathsf{no} \triangleleft p'$ as required. □

Proposition 4 sHML *is Rejection Expressive-Complete.*

Proof Follows from Lemma 13, (9), and the fact that the co-domain of $\langle\!\langle - \rangle\!\rangle$ is that of sHML formulae. □

Definition 10 (Acceptance Expressive-Complete) Language $\mathcal{L} \subseteq \mu\text{HML}$ is *expressive-complete wrt.* acceptance monitors iff for all $m \in \text{AMon}$ there exists some $\varphi \in \mathcal{L}$ such that $\mathbf{mon}(m, \varphi)$. ■

Proposition 5 cHML *is Acceptance Expressive-Complete.*

Proof The proof is analogous to that of Proposition 4. □

Equipped with Proposition 4 and Proposition 5, it follows that mHML is expressive complete *wrt.* uni-verdict monitors.

Definition 11 (Expressive-Complete) $\mathcal{L} \subseteq \mu\text{HML}$ is *expressive-complete wrt.* well-formed uni-verdict monitors iff for all $m \in \text{AMon} \cup \text{RMon}$ there exists some $\varphi \in \mathcal{L}$ such that $\mathbf{mon}(m, \varphi)$. ■

Theorem 3 mHML *is Expressive-Complete.*

Proof Follows immediately from Proposition 4 and Proposition 5. □

We are now in a position to prove the result alluded to at the beginning of this section, namely that mHML is the largest monitorable subset of μHML up to logical equivalence, *i.e.,* Theorem 4. First, however, we define what we understand by language inclusion up to formula semantic equivalence, Definition 12.

Definition 12 (Language Inclusion) For all $\mathcal{L}_1, \mathcal{L}_2 \in \mu\text{HML}$

$$\mathcal{L}_1 \sqsubseteq \mathcal{L}_2 \stackrel{\text{def}}{=} \forall \varphi_1 \in \mathcal{L}_1.\, \exists \varphi_2 \in \mathcal{L}_2 \text{ such that } [\![\varphi_1]\!] = [\![\varphi_2]\!]$$

We also prove the following important proposition, that gives an upper bound to the expressiveness of languages satisfying monitorability properties.

Proposition 6 *For any* $\mathcal{L} \subseteq \mu\text{HML}$*:*

1. $\big(\forall \varphi \in \mathcal{L}.\, \exists m \in \text{AMon}.\, \forall p.(\boldsymbol{acc}(p, m) \text{ iff } p \in [\![\varphi]\!])\big)$ *implies* $\mathcal{L} \sqsubseteq$ cHML.
2. $\big(\forall \varphi \in \mathcal{L}.\, \exists m \in \text{RMon}.\, \forall p.(\boldsymbol{rej}(p, m) \text{ iff } p \notin [\![\varphi]\!])\big)$ *implies* $\mathcal{L} \sqsubseteq$ sHML.

Proof We prove the first clause; the second clause is analogous. Assume $\varphi \in \mathcal{L}$. We need to show that $\exists \pi \in \text{cHML}$ such that $[\![\varphi]\!] = [\![\pi]\!]$. For φ we know that

$$\exists m \in \text{AMon}.\, \big(\forall p.(\mathbf{acc}(p, m) \text{ iff } p \in [\![\varphi]\!])\big). \tag{17}$$

By Proposition 5, for the monitor m used in (17), we also know

$$\exists \pi \in \text{cHML}.\, \big(\forall p.(\mathbf{acc}(p, m) \text{ iff } p \in [\![\pi]\!])\big). \tag{18}$$

Assume an arbitrary $p \in [\![\varphi]\!]$. By (17) we obtain $\mathbf{acc}(p, m)$, and by (18) we obtain $p \in [\![\pi]\!]$. Thus $[\![\varphi]\!] \subseteq [\![\pi]\!]$. Dually, we can also reason that $[\![\pi]\!] \subseteq [\![\varphi]\!]$. □

Theorem 4 (Completeness) *If a language* $\mathcal{L} \subseteq \mu\text{HML}$ *is monitorable then* $\mathcal{L}$ *cannot (semantically) express more properties than* mHML*, that is* $\mathcal{L} \sqsubseteq$ mHML*.*

Proof Since $\mathcal{L}$ is monitorable, by Definition 5 and Definition 4 we know:

$$\forall \varphi \in \mathcal{L}.\ \exists m \text{ such that } \mathbf{smon}(m, \varphi) \text{ and } \mathbf{cmon}(m, \varphi) \tag{19}$$

By Theorem 2, we know that every m used in (19) is uni-verdict. This means that we can partition the formulae in $\mathcal{L}$ into two disjoint sets $\mathcal{L}_{\text{acc}} \uplus \mathcal{L}_{\text{rej}}$ where:

$$\big(\forall \varphi \in \mathcal{L}_{\text{acc}}.\ \exists m \in \text{AMon}.\ \forall p.(\mathbf{acc}(p, m) \text{ iff } p \in [\![\varphi]\!])\big) \tag{20}$$

$$\big(\forall \varphi \in \mathcal{L}_{\text{rej}}.\ \exists m \in \text{RMon}.\ \forall p.(\mathbf{rej}(p, m) \text{ iff } p \notin [\![\varphi]\!])\big) \tag{21}$$

By (20), (21) and Proposition 6 we obtain $\mathcal{L}_{\text{acc}} \sqsubseteq$ cHML and $\mathcal{L}_{\text{rej}} \sqsubseteq$ sHML *resp.*, from which the required result follows. □

Theorem 2 and Theorem 4 constitute powerful results *wrt.* the monitorability of our branching-time logic. Completeness, Theorem 4, guarantees that limiting oneself to the syntactic subset mHML does not hinder the *expressive power* of the specifier when formulating monitorable properties. Alternatively, one could also determine whether a formula is monitorable by rewriting it as a logically equivalent formula in mHML.[3] This would enable a verification framework to decide whether to check for a μHML property at runtime, or resort to more expressive (but expensive means) otherwise. Whenever the property is monitorable, Theorem 2 guarantees that a uni-verdict monitor is the best monitor that we can synthesise. This is important since multi-verdict monitor constructions, such as those in [11], generally carry *higher overheads* than uni-verdict monitors.

Example 17 By virtue of Theorem 4, we can conclude that properties φ_1, φ_2, φ_3, φ_5 and φ_6 from Example 3 are all non-monitorable properties according to Definition 5, since no logically equivalent formulae in mHML exist. Arguably, the problem of establishing logical equivalence through syntactic manipulation of formulae is easier to determine and automate, when compared to direct reasoning about the semantic definitions of monitorability and those of the *resp.* properties; recall that Definition 5 (Monitorability) — through Definition 2 and Definition 3 — universally quantifies over all processes, which generally poses problems for automation.

For instance, in the case of φ_1, we could use Theorem 2 to substantially reduce the search space of our witness monitor to the uni-verdict ones, but this still leaves us with a lot of work to do. Specifically, we can reason that the witness *cannot* be an *acceptance monitor*, since it would need to accept process nil, which implies that it must erroneously also accept the process $\mathsf{cls.nil}$ (using reasoning similar to that used in Example 11). It is less straightforward to argue that the witness *cannot* be a *rejection monitor* either. We argue towards a contradiction by assuming that such a monitor exists. Since it is violation-complete (Definition 3) it should reject the process $\mathsf{req.nil} + \mathsf{cls.nil}$ since this process does not satisfy φ_1: by Proposition 1 we know that it can do so along either of the traces ϵ, req or cls. If it rejects it along ϵ, then it also rejects the satisfying process nil; if it rejects along trace req, it also rejects the satisfying process $\mathsf{req.ans.nil}$; finally, if it rejects it along cls, it must also reject the satisfying process $\mathsf{req.ans.nil} + \mathsf{cls.nil}$. Thus, the monitor must be unsound, meaning that it cannot be a rejection monitor. ■

[3] The problem of determining whether a (general) formula is logically equivalent to one in mHML is decidable in exponential time — probably EXPTIME complete.

8 Conclusion

We have investigated monitorability aspects of a branching-time logic called μHML, a reformulation of the modal μ-calculus that is regarded as one of the most expressive temporal logics [27,28] for graph-based system descriptions (such as LTSs). More specifically, we identified a syntactic subset of this logic, Definition 6, that is the largest monitorable subset in terms of semantic expressivity, Theorem 1 and Theorem 4. We conjecture that our results and methodologies are applicable, at least in part, to other logical specifications that are not necessarily bespoke for RV, thereby extending the utility of the verification technique to a wider class of program logics. Our methods for delineating monitorable logical subsets can also be useful as a foundation for establishing guiding principles when apportioning verification across the pre-deployment and post-deployment phases of software development [4,19].

Future Work: The results obtained in this work are cast *wrt.* a specific definition of monitorability, Definition 5, which relies on a particular set of requirements for adequate monitoring, Definition 4. However, there are arguably other valid notions of adequate monitoring, depending the application targetted, and it is worth exploring how such alternative definitions for monitorability affect the monitorable subset of μHML identified in this work. For instance, one could relax the conditions of Definition 4 by only requiring soundness (Definition 2), or require more stringent conditions *wrt.* verdicts and monitor non-determinism; see [24] for a practical motivation of this.

Monitorability is, in turn, largely dependent on the underlying monitoring setup and instrumentation relation. Potentially, there are other sensible relations apart from the one defined in Figure 4 that are worth considering and it would be interesting to investigate whether this impacts on the monitorable logical subset identified in this work.

In a parallel line of research, we intend to investigate manipulation techniques that decompose formulae into monitorable components as proposed in [19]. For instance, reformulating a generic μHML formula φ as the disjunction $\phi \vee \varpi$ (recall that $\varpi \in$ cHML) could allow for a *combined* verification approach where ϖ can be adequately verified at runtime using an automatically synthesised dedicated monitor, $(\!|\varpi|\!)$, whereas the remainder formula ϕ can be verified using traditional techniques such as model checking.

Related Work: We are not the first to address the issue of monitorability in specification logics. In [32], a notion of monitorability is defined for *formulae defined over traces* (*e.g.,* LTL) whenever the formula semantics does not contain *ugly* prefixes; an ugly prefix is a trace from which *no* finite extension will ever lead to a conclusive verdict. They however do not investigate any maximally expressive monitorable subset of the logic. The closest result in this regard is that found in [17]: the authors identify an LTL subset for which satisfactions are backed up by derivations in a sound proof system for the logic, and subsequently show how a monitoring algorithm can be automated to infer such proof derivations. They however do not show whether this subset is in some sense maximal in terms of monitorability. Falcone *et al.* [21] revisit the monitorability definition of [32], and extend it to the Safety-Progress property classification, while proposing an alternative definition

in terms of the structure of the recognising Streett Automata of the *resp.* property. Although our definition is cast within a different setting (a logic over processes), and has a distinct operational flavour in terms of monitored system executions, it is certainly worthwhile to try to reconcile these different definitions. We however anticipate such a reconciliation effort to be non-trivial, partly due to the difference in the technical machinery used by the respective bodies of work.

The logic μHML has been previously studied from a linear-time perspective in [2,15], in order to find subsets that characterise may/must testing equivalences. Tests are different from the monitors used in our study (*e.g.,* they actively interact with the system under scrutiny) and the composition relation between tests and systems is distinctly different from the our monitor instrumentation relation. Moreover, their notion of successful computations is disparate from our definition of acceptances and rejections (*e.g.,* in a testing setup, success may be revoked and computations are anonimised, whereas in monitored executions detections are persistent and may be distinguished by their visible trace of actions); see [22] for a more detailed comparison. Interestingly however, the logic subset identified in [2] is related to mHML; in fact the testable subset *wrt.* may testing coincides with sHML.

Subsets of the logic μHML have already been used to develop RV tools for checking the behaviour of reactive systems. In [24], the monitorable subset sHML was used to specify safety properties of concurrent Erlang programs. For this logical subset, a monitor synthesis algorithm is defined that generates decentralised monitor arrangements that check for property violations in a concurrent fashion. In recent work [14,13], the sub-logic has been extended into a form of domain specific language so as to specify adaptation procedures to be taken by the monitor once a violation is detected, thereby mitigating program errors at runtime.

The instrumentation relation used in this work has been investigated further in [22], where it is used to define a number of contextual refinement preorders for monitors that share a similar structure to the ones employed in our study. The work subsequently develops compositional reasoning techniques for the preorders based on our instrumentation relation. In relation to our work, the monitor refinement relations of [22] can be used to compare alternative monitor synthesis procedures such as ours and that of [24] for the monitorable logical subsets identified.

References

1. J. R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.
2. L. Aceto and A. Ingólfsdóttir. Testing Hennessy-Milner Logic with Recursion. In *FoSSaCS'99*, pages 41–55. Springer, 1999.
3. L. Aceto, A. Ingólfsdóttir, K. G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification.* Cambridge Univ. Press, New York, NY, USA, 2007.
4. W. Ahrendt, J. M. Chimento, G. J. Pace, and G. Schneider. A specification language for static and runtime verification of data and control properties. In N. Bjørner and F. deBoer, editors, *FM 2015*, pages 108–125. Springer, 2015.
5. J. R. Andersen, N. Andersen, S. Enevoldsen, M. M. Hansen, K. G. Larsen, S. R. Olesen, J. Srba, and J. Wortmann. CAAL: concurrency workbench, aalborg edition. In *ICTAC*, 2015.
6. C. Baier and J. P. Katoen. *Principles of Model Checking.* MIT Press, New York, May 2008.

7. H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. E. Rydeheard. Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In *FM*, volume 7436 of *LNCS*, pages 68–84. Springer, 2012.
8. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
9. A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *RV*, volume 4839 of *LNCS*, pages 126–138. Springer, 2007.
10. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *Logic and Comput.*, 20(3):651–674, 2010.
11. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *TOSEM*, 20(4):14, 2011.
12. I. Cassar and A. Francalanza. On Synchronous and Asynchronous Monitor Instrumentation for Actor Systems. In *FOCLASA*, volume 175, pages 54–68, 2014.
13. I. Cassar and A. Francalanza. Runtime Adaptation for Actor Systems. In *Runtime Verification (RV)*, volume 9333 of *LNCS*, pages 38–54. Springer, 2015.
14. I. Cassar and A. Francalanza. On Implementing a Monitor-Oriented Programming Framework for Actor Systems. In *integrated Forma Methods (iFM)*, LNCS. Springer, 2016. (to appear).
15. A. Cerone and M. Hennessy. Process behaviour: Formulae vs. tests. In *EXPRESS*, volume 41 of *EPTCS*, pages 31–45, 2010.
16. E. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In *ALP LNCS*, pages 474–486. Springer-Verlag, 1992.
17. C. Cini and A. Francalanza. An LTL Proof System for Runtime Verification. In *TACAS*, volume 9035, pages 581–595. Springer, 2015.
18. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
19. D. Della Monica and A. Francalanza. Towards a Hybrid Approach to Software Verification. In *NWPT*, number SCS16001 in RUTR, pages 51–54. RU Press, 2015.
20. C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. V. Campenhout. Reasoning with temporal logic on truncated paths. In *CAV*, volume 2725 of *LNCS*, pages 27–39. Springer, 2003.
21. Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.
22. A. Francalanza. A Theory of Monitors (Extended Abstract). In *FoSSaCS*, volume 9634 of *LNCS*, pages 145–161. Springer, 2016.
23. A. Francalanza, A. Gauci, and G. J. Pace. Distributed System Contract Monitoring. *JLAP*, 82(5-7):186–215, 2013.
24. A. Francalanza and A. Seychell. Synthesising Correct concurrent Runtime Monitors. *FMSD*, 46(3):226–261, 2015.
25. M. Geilen. On the Construction of Monitors for Temporal Logic Properties. In *RV*, volume 55 of *ENTCS*, pages 181–199, 2001.
26. C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, 1985.
27. D. Kozen. Results on the propositional μ-calculus. *TCS*, 27:333–354, 1983.
28. K. G. Larsen. Proof systems for satisfiability in hennessy-milner logic with recursion. *TCS*, 72(2):265 – 288, 1990.
29. M. Leucker and C. Schallhart. A brief account of Runtime Verification. *JLAP*, 78(5):293 – 303, 2009.
30. Z. Manna and A. Pnueli. Completing the Temporal Picture. *TCS*, 83(1):97–130, 1991.
31. R. Milner. *A Calculus of Communicating Systems*. Springer, 1982.
32. A. Pnueli and A. Zaks. Psl model checking and run-time verification via testers. In *FM*, pages 573–586. Springer, 2006.
33. K. Sen, G. Rosu, and G. Agha. Generating optimal linear temporal logic monitors by coinduction. In *ASIAN*, LNCS, pages 260–275. Springer, 2004.
34. detectEr Project. `http://www.cs.um.edu.mt/svrg/Tools/detectEr/`.
35. A. Vella and A. Francalanza. Preliminary results towards contract monitorability. In *PrePost*, 2016. (to appear).