On Verifying Hennessy-Milner Logic with Recursion at Runtime[⋆]

Adrian Francalanza[1], Luca Aceto[2], and Anna Ingolfsdottir[2]

[1] CS, ICT, University of Malta, Malta `adrian.francalanza@um.edu.mt`
[2] ICE-TCS, Reykjavik University, Iceland `{luca,anna}@ru.is`

Abstract. We study μHML (a branching-time logic with least and greatest fixpoints) from a runtime verification perspective. We establish which subset of the logic can be verified at runtime and define correct monitor-synthesis algorithms for this subset. We also prove completeness results *wrt.* these logical subsets that show that no other properties apart from those identified can be verified at runtime.

1 Introduction

Runtime Verification (RV) [20, 14] is a lightweight verification technique whereby the *execution of a system* is analysed with the aim of inferring correctness *wrt.* some property. Despite its advantages, the technique is generally limited when compared to other verification techniques such as model-checking because certain correctness properties cannot be verified at runtime [21, 10]. For instance, *online* RV analyses *partial* executions incrementally (up to the current execution point of the system) which limits its applicability to satisfaction verdicts relating to correctness properties describing complete (*i.e.*, potentially infinite) executions.

There are broadly two approaches 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) [12, 15], or else *redefines* the semantics of existing logics (*e.g.*, LTL) so as to reflect the limitations of the runtime setting [13, 5, 7, 6]. The second approach is to leave the semantics of the specification logic *unchanged*, and study which *subsets* of the logic can be verified at runtime [17, 24, 11, 16].

Both approaches have their merits. The first approach is, in general, more popular and tends to produce specifications that are closely related to the monitors that check for them (*e.g.*, RE and automata in [15, 23]), thus facilitating aspects such as monitor correctness. 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 either to check 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 hybrid approach,

[⋆] The research of L. Aceto and A. Ingolfsdottir was supported by the project 001-ABEL-CM-2013 of the NILS Science and Sustainability Programme and the project Nominal SOS (nr. 141558-051) of the Icelandic Research Fund.

Syntax

$$p, q, r \in \textsc{Proc} ::= \mathsf{nil} \quad \text{(inaction)} \quad \mid \alpha.p \quad \text{(prefixing)} \quad \mid p + q \quad \text{(choice)}$$
$$\mid \mathsf{rec}\, x.p \quad \text{(recursion)} \quad \mid x \quad \text{(rec. variable)}$$

Dynamics

$$\textsc{Act}\,\frac{}{\alpha.p \xrightarrow{\alpha} p} \qquad \textsc{Rec}\,\frac{}{\mathsf{rec}\, x.p \xrightarrow{\tau} p[\mathsf{rec}\, x.p/x]} \qquad \textsc{SelL}\,\frac{p \xrightarrow{\mu} p'}{p + q \xrightarrow{\mu} p'}$$

Fig. 1. A Model for describing Systems

where parts of a property are verified using RV and other parts are checked using other techniques. 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.

This paper follows this second approach. In particular, it revisits Hennessy-Milner Logic with recursion [3], μHML, a reformulation of the expressive modal μ-calculus [19], used to describe correctness properties of reactive system; subsets of the logic have already been adapted for detectEr [25], an RV tool for runtime-verifying actor-based systems [16, 8], whereas constructs from the modal μ-calculus have been used in other RV tools such as Eagle [4]. In this study we consider the logic in its entirety, and investigate the monitorability of the logic *wrt.* an operational definition of a general class of monitors that employ both acceptance and rejection verdicts [7, 14]. In particular, our results extend the class of monitorable μHML properties used in [16] and establish monitorability upper bounds for this logic. We also present new results that relate the utility of multi-verdict monitors *wrt.* logics defined over programs (as opposed to traces). To the best of our knowledge, this is one of the first bodies of work investigating the limits of RV *wrt.* a *branching-time* logic that specifies properties about the *execution graph* of a program; other work pertaining to the aforementioned second approach has focussed on *linear-time* logics defined over *execution traces*, and has explored RV's limits along the linear-time dimension, *e.g.*, [11].

In the rest of the paper, Sec. 2 introduces our model for reactive systems and Sec. 3 presents the logic μHML defined over this model. Sec. 4 formalises our abstract RV operational setup in terms of monitors and our instrumentation relation. In Sec. 5 we argue for a particular correspondence between monitors and μHML properties within this setup. Sec. 6 identifies monitorability limits for the logic but also establishes a monitorable logical subset that satisfies the correspondence of Sec. 5. Sec. 7 shows that this subset is maximally expressive, using a result about multi-verdict monitors. Sec. 8 concludes.

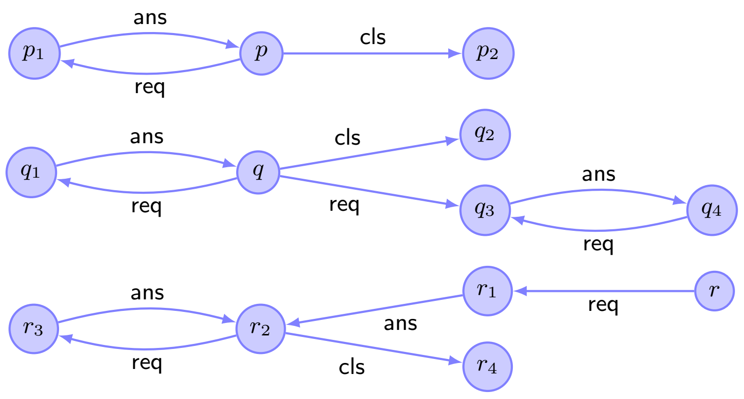

Fig. 2. A depiction of the system in Example 1

2 The Model

We describe systems abstractly as Labelled Transition Systems (LTSs) [3, 19]. An LTS is a triple $\langle \textsc{Proc}, (\textsc{Act} \cup \{\tau\}), \longrightarrow \rangle$ consisting of a set of states, $\textsc{Proc}$, a set of actions, $\textsc{Act}$ with distinguished silent action τ (we assume $\mu \in \textsc{Act} \cup \{\tau\}$ and $\tau \notin \textsc{Act}$), and a transition relation, $\longrightarrow \subseteq (\textsc{Proc} \times (\textsc{Act} \cup \{\tau\}) \times \textsc{Proc})$. LTS states can be expressed as processes, $\textsc{Proc}$, from the regular fragment of CCS [22] as defined in Fig. 1. Assuming a set of (visible) actions, $\alpha, \beta \in \textsc{Act}$ and a set of (recursion) variables $x, y, z \in \textsc{Vars}$, processes may be either inactive, prefixed by an action, a mutually-exclusive choice amongst two processes, or recursive; $\mathsf{rec}\, x.p$ acts as a *binder* for x in p and we work up to alpha-conversion of bound variables. All recursive processes are assumed to be guarded.

The dynamic behaviour is then described by the transition rules of Fig. 1, defined over the closed and guarded terms in $\textsc{Proc}$ (we elide the symmetric rule $\textsc{SelR}$). The suggestive notation $p \xrightarrow{\mu} p'$ denotes $(p, \mu, p') \in\longrightarrow$; we also write $p \not\xrightarrow{\alpha}$ to denote $\neg(\exists p'.\, p \xrightarrow{\alpha} p')$. For example, $p_1 + p_2 \xrightarrow{\mu} q$ if either $p_1 \xrightarrow{\mu} q$ or $p_2 \xrightarrow{\mu} q$. As usual, we 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 $t, u \in \textsc{Act}^*$ range over sequences of visible actions and write $p \overset{\alpha_1}{\Longrightarrow} \ldots \overset{\alpha_n}{\Longrightarrow} p_n$ as $p \overset{t}{\Longrightarrow} p_n$, where $t = \alpha_1, \ldots, \alpha_n$. See [22, 3] for more details.

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.

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

Syntax

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

Semantics

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

Fig. 3. μHML Syntax and Semantics

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.

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

Pictorially, the *resp.* LTSs denoted by processes p, q and r are shown in Fig. 2, where the arcs correspond to weak transitions, $\stackrel{\mu}{\Longrightarrow}$. ■

3 The Logic

The logic μHML 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 Fig. 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 equality up to alpha-conversion.

Formulae are interpreted over the process powerset domain, $S \in \mathcal{P}(\textsc{Proc})$. The semantic definition of Fig. 3 is given for *both* open and closed formulae and employs an environment from variables to sets of processes, $\rho \in \textsc{LVar} \rightharpoonup \mathcal{P}(\textsc{Proc})$; this permits an inductive definition on the structure of the formula. For instance, in Fig. 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 formula φ whenever $p \in [\![\varphi]\!]$, and violates a formula whenever $p \notin [\![\varphi]\!]$.

Example 2. 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.

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

Formula φ_1 denotes a *liveness* property describing processes that *eventually* stop offering the action cls after any number of serviced request, $(\mathsf{req}.\mathsf{ans})^*$—processes q and r from Ex. 1 satisfy this property but p does not. Changing the fixpoint into a maximal one, *i.e.,* φ_2, would include p in the property as well. 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 (p satisfies this property but q and r do not).

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

Formula φ_5 is satisfied by processes that, after one serviced request, $\mathsf{req}.\mathsf{ans}$, exhibit *a* complete[3] transition sequence $(\mathsf{req}.\mathsf{ans})^*$ that never offers action cls (q from Ex. 1 satisfies this property whereas p and r do not). Formula φ_6 describes processes that along *all* serviced request sequences, $(\mathsf{req}.\mathsf{ans})^*$, eventually reach a stage where they offer action cls (processes p and q satisfy the criteria immediately, whereas r satisfies it for $(\mathsf{req}.\mathsf{ans})^*$ sequences longer than 1). ■

4 Monitors and instrumentation

Monitors may also be viewed as LTSs, through the syntax of Fig. 4; this is similar to that of processes, with the exception that 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 rules in Fig. 4. The only new transition concerns verdicts, mVer, stating that a verdict may transition with *any* $\alpha \in \textsc{Act}$ and go back to the same state, modelling the requirement that verdicts are *irrevocable.*

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

Syntax

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

Dynamics

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

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

Instrumentation

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

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

Fig. 4. Monitors and Instrumentation

Fig. 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 [22, 18]. 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, $m \not\xrightarrow{\alpha}$, even after any number of internal actions, $m \not\xrightarrow{\tau}$, instrumentation forces it to terminate with an inconclusive verdict, end, while the process is allowed to proceed unaffected; see rule iTer. Rules iAsyP and iAsyM allow monitors and processes to transition independently *wrt.* internal moves.[4]

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

- *either $m \overset{t}{\Longrightarrow} m'$*
- *or $m' = \mathsf{end}$ and $\exists t', \alpha, t'', m''.\ t = t'\alpha t''$, $m'' \not\xrightarrow{\tau}$ and $m \overset{t'}{\Longrightarrow} m'' \not\xrightarrow{\alpha}$.*

Remark 1. Since we strive towards a general theory, the syntax in Fig. 4 allows for non-deterministic monitors such as $\alpha.\mathsf{yes} + \alpha.\mathsf{no}$ or $\alpha.\mathsf{nil} + \alpha.\beta.\mathsf{yes}$. There are

[4] If a monitor cannot match a process action, but can transition silently, it is allowed to do so, and the matching check is applied again to the τ-derivative monitor.

settings where determinism is unattainable (*e.g.,* distributed monitoring [15]) or desirable (*e.g.,* testers [23]), and others where non-determinism expresses under-specification (*e.g.,* program refinement [1]). Thus, expressing non-determinism allows us to study the cases where it is tolerated or considered erroneous.

Example 3. Monitor m_1 (defined below) monitors for executions of the form $(\mathsf{req}.\mathsf{ans})^*.\mathsf{cls}$ returning the acceptance verdict yes, whereas m_2 dually rejects executions of that form. When composed with process p from Ex. 1, the monitored system $m_1 \triangleleft p$ may either service requests forever, $m_1 \triangleleft p \overset{\mathsf{req}}{\Longrightarrow} \cdot \xrightarrow{\mathsf{ans}} m_1 \triangleleft p$, or else terminate with a yes verdict, $m_1 \triangleleft p \overset{\mathsf{cls}}{\Longrightarrow} \mathsf{yes} \triangleleft \mathsf{nil}$. By contrast, when instrumented over a process capable of the transition $p' \xrightarrow{\mathsf{ans}} p''$, m_1 may terminate its process observation after one transition, *i.e.,* $m_1 \triangleleft p' \overset{\mathsf{ans}}{\Longrightarrow} \mathsf{end} \triangleleft p''$.

$$m_1 = \mathsf{rec}\, x.\big(\mathsf{req}.\mathsf{ans}.x + \mathsf{cls}.\mathsf{yes}\big) \qquad m_3 = \mathsf{rec}\, x.\big(\mathsf{req}.\mathsf{ans}.x + \mathsf{cls}.\mathsf{yes} + \mathsf{req}.\mathsf{req}.x\big)$$

$$m_2 = \mathsf{rec}\, x.\big(\mathsf{req}.\mathsf{ans}.x + \mathsf{cls}.\mathsf{no}\big) \qquad m_4 = \mathsf{rec}\, x.\big(\mathsf{req}.\mathsf{ans}.x + \mathsf{cls}.\mathsf{yes} + \mathsf{cls}.\mathsf{no}\big)$$

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

5 Correspondence

Our goal is to establish a correspondence between the verdicts reached by monitors over an instrumented system from Sec. 4 and the properties specified using the logic of Sec. 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 μHML formula, φ. This will, in turn, allow us to determine when a monitor m *represents* (in some precise sense) a property φ.

Example 4. Monitor m_1 from Ex. 3 monitors for *satisfactions* of the property

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

describing processes that can perform a cls action after a number of serviced requests. Stated otherwise, m_1 produces a yes verdict for a computation of the form $(\mathsf{req}.\mathsf{ans})^*.\mathsf{cls}$ from a process p, attesting that $p \in [\![\varphi_7]\!]$. Similarly, m_2 from Ex. 3 monitors for *violations* of the property φ_4 from Ex. 2. The same *cannot* be said for m_4 from Ex. 3 and φ_7 above: for some processes, *e.g.,* p from Ex. 1, it may produce both verdicts yes and no for witness computations $(\mathsf{req}.\mathsf{ans})^*.\mathsf{cls}$, which leads to contradictions at a logical level *i.e.,* we cannot have both $p \in [\![\varphi_7]\!]$ and $p \notin [\![\varphi_7]\!]$. A similar argument applies to m_4 and m_2 from Ex. 3. ■

Remark 2. A monitor may behave non-deterministically in other ways *wrt.* a process. For instance, m_3 from Ex. 3 may sometimes flag an acceptance but at other times may not when monitoring p from Ex. 1, *even when* p produces the same trace: *e.g.*, for $t = \mathsf{req.ans.cls}$ we have $m_3 \triangleleft p \overset{t}{\Longrightarrow} \mathsf{yes} \triangleleft \mathsf{nil}$ but also $m_3 \triangleleft p \overset{t}{\Longrightarrow} \mathsf{end} \triangleleft \mathsf{nil}$. However, since any other terminal monitor state apart from yes and no (*i.e.*, end and any other non-verdict state) is of no consequence from a logical satisfaction/violation point of view, we abstract from such outcomes.

We investigate conditions that one could require for establishing the correspondence between monitors and formulae. We start with Def. 2 (below): it defines when m is able to *monitor 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/Rejection). *$\boldsymbol{acc}(p, m) \overset{def}{=} \exists t, p'.\ m \triangleleft p \overset{t}{\Longrightarrow} \mathsf{yes} \triangleleft p'$ and $\boldsymbol{rej}(p, m) \overset{def}{=} \exists t, p'.\ m \triangleleft p \overset{t}{\Longrightarrow} \mathsf{no} \triangleleft p'$.*

Definition 2 (Sound Monitoring).

$$\boldsymbol{smon}(m, \varphi) \overset{def}{=} \forall p.\big(\boldsymbol{acc}(p, m) \textit{ implies } p \in [\![\varphi]\!]\big) \textit{ and } \big(\boldsymbol{rej}(p, m) \textit{ implies } p \notin [\![\varphi]\!]\big)$$

Note that, if $\mathbf{smon}(m, \varphi)$ and $\exists p.\mathbf{acc}(p, m)$, by Def. 2 we know $p \in [\![\varphi]\!]$; thus, $\neg(p \notin [\![\varphi]\!])$ and by the contrapositive of Def. 2, we must also have $\neg\mathbf{rej}(p, m)$.

Example 5. From Ex. 4, we formally have $\mathbf{smon}(m_1, \varphi_7)$, $\mathbf{smon}(m_2, \varphi_2)$ and $\mathbf{smon}(m_3, \varphi_7)$. We can also show that $\neg\mathbf{smon}(m_4, \varphi_7)$ and $\neg\mathbf{smon}(m_4, \varphi_2)$. ■

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 Def. 2, *i.e.*, *complete monitoring* for m and φ, stating 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 Fig. 3.

Example 6. Consider the basic formula $\langle\alpha\rangle\mathsf{tt}$. One could ascertain that the simple monitor $\alpha.\mathsf{yes}$ satisfies the condition that $p \in [\![\varphi]\!]$ implies $\mathbf{acc}(p, m)$ for all p. However, there does not exist a sound monitor that can satisfy $\forall p.p \notin [\![\varphi]\!]$ implies $\mathbf{rej}(p, m)$ for $\langle\alpha\rangle\mathsf{tt}$. Arguing by contradiction, assume that one such monitor m exists. Since $\mathsf{nil} \notin [\![\langle\alpha\rangle\mathsf{tt}]\!]$ then we should have $\mathbf{rej}(\mathsf{nil}, m)$. By Def. 1 and Prop. 1, this means $m \Longrightarrow \mathsf{no}$ which, in turn, implies that $\mathbf{rej}(\alpha.\mathsf{nil}, m)$ 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, $[\alpha]\mathsf{ff}$: 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 $\mathbf{acc}(p, m)$. ■

Concretely, requiring complete monitoring would limit correspondence to a trivial subset of the logic, namely tt and ff. We therefore define the weaker forms of completeness that are stated below.

Definition 3 (Satisfaction/Violation/Partially-Complete Monitoring).

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

We can now formalise monitor-formula correspondence: m *monitors* for 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.

Definition 4 (Monitoring). $\textbf{\textit{mon}}(m,\varphi) \overset{\textit{def}}{=} \textbf{\textit{smon}}(m,\varphi)$ *and* $\textbf{\textit{cmon}}(m,\varphi)$.

6 Monitorability

Using Def. 4, we can define what it means for a formula to be monitorable.

Definition 5 (Monitorability). *Formula* φ *is* monitorable *iff* $\exists m.\textbf{\textit{mon}}(m,\varphi)$. *A language* $\mathcal{L} \subseteq \mu$HML *is monitorable iff every* $\varphi \in \mathcal{L}$ *is monitorable.*

We immediately note that not all logical formulae are monitorable.

Example 7. Through the witness outlined in Ex. 6, we can show that formulae $\langle\alpha\rangle\mathsf{tt}$ and $\langle\beta\rangle\mathsf{tt}$ are monitorable with a satisfaction complete monitor. However, $\varphi_8 = \langle\alpha\rangle\mathsf{tt}\wedge\langle\beta\rangle\mathsf{tt}$ (their conjunction), is *not.* Intuitively, this is so because once a monitor observes one of the actions, it cannot "go back" to check for the other. Formally, we argue towards a contradiction by assuming that $\exists m.\mathbf{mon}(m,\varphi_8)$. There are two subcases 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 Prop. 1, m reaches verdict yes along one of the traces ϵ,α or β. If the trace is ϵ, then m also accepts nil, which is unsound (since $\mathsf{nil} \notin [\![\varphi_8]\!]$) whereas if the trace is α, m must also accept $\alpha.\mathsf{nil}$, 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 Prop. 1, we either have $m \overset{\epsilon}{\Longrightarrow} \mathsf{no}$ or $m \overset{\beta}{\Longrightarrow} \mathsf{no}$ 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 a 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, it consists of the safe and co-safe syntactic subsets of μHML, sHML and cHML respectively.

Definition 6 (Monitorable Logic). $\psi,\chi \in$ MHML $\overset{\textit{def}}{=}$ sHML$\cup$cHML *where:*

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

To prove monitorability for mHML, we define a monitor synthesis function $(\!|-|\!)$ generating a monitor for each $\psi \in$ mHML. We then show that $(\!|\psi|\!)$ is the witness monitor required by Def. 5 to demonstrate the monitorability of ψ.

Definition 7 (Monitor Synthesis).

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

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

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

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

A few comments are in order. We first note that Def. 7 is *compositional*; see *e.g.*, [23] for reasons why this is desirable. It also assumes a bijective mapping between the denumerable sets LVar and 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. Although Def. 7 covers both sHML and cHML, the syntactic constraints of Def. 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). The conditional cases handle logically equivalent formulae, *e.g.*, since $[\![\mathsf{ff}]\!] = [\![\mathsf{min}\, X.\langle\alpha\rangle\mathsf{ff}]\!]$ we have $(\!|\mathsf{ff}|\!) = (\!|\mathsf{min}\, X.\langle\alpha\rangle\mathsf{ff}|\!) = \mathsf{no}$. In the case of conjunctions and disjunctions, these are essential to be able to generate sound monitors.

Example 8. 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 would generate the monitor $\alpha.\mathsf{yes} + \mathsf{no}$ which is not only redundant, but *unsound* *e.g.*, for $p = \alpha.\mathsf{nil}$ 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, *e.g.*, $\langle\alpha\rangle\mathsf{tt}\vee(\mathsf{min}\, X.\langle\alpha\rangle\mathsf{ff})\vee(\langle\alpha\rangle\mathsf{min}\, X.\mathsf{ff})$. However, in all of these cases, our monitor synthesis of Def. 7 generates $\alpha.\mathsf{yes}$. ■

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

Proof. We show that for all $\varphi \in$ mHML, $\mathbf{mon}((\!|\varphi|\!), \varphi)$ holds. □

Thm. 1 provides us with a simple syntactic check to determine whether a formula is monitorable; as shown earlier in Ex. 7, determining whether a formula is monitorable is in general non-trivial. Moreover, the proof of Thm. 1 (through Def. 7) provides us with an automatic monitor synthesis algorithm that is correct according to Def. 4.

Example 9. Since φ_4 from Ex. 2 is in mHML, we know it is monitorable. Moreover, we can generate the *correct* monitor $(\!|\varphi_4|\!) = \mathsf{rec}\, x.\big(\mathsf{req}.\mathsf{ans}.x + \mathsf{cls}.\mathsf{no}\big) = m_2$ (from Ex. 3). Using similar reasoning, φ_7 (Ex. 4) is also monitorable, and a correct monitor for it is m_1 from Ex. 3. ■

7 Expressiveness

The results obtained in Sec. 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 Def. 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 Def. 4. To this end, we prove a general (and perhaps surprising) result.

Theorem 2 (Multi-verdict Monitors and Monitoring). $\forall m \in \text{Mon}$

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

Proof. By contradiction. Assume that $\exists\varphi.\textbf{mon}(m,\varphi)$. Using t and u, we can construct the obvious process $p = t + u$, where $m \triangleleft p \overset{t}{\Longrightarrow} \mathsf{yes} \triangleleft \mathsf{nil}$ and $m \triangleleft p \overset{u}{\Longrightarrow} \mathsf{no} \triangleleft \mathsf{nil}$. We therefore have $\textbf{acc}(p,m)$ and $\textbf{rej}(p,m)$, and by monitor soundness (Def. 2), that $p \in [\![\varphi]\!]$ and $p \notin [\![\varphi]\!]$. This is clearly a contradiction. $\square$

Stated otherwise, Thm. 2 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.*, [11, 7, 14]). This also implies that, in order to answer the aforementioned question, 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 Def. 7 reveals that all the monitors generated using it are, in fact, uni-verdict.

We partition 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* $\mathcal{L} \subseteq \mu$HML *is* expressive-complete *wrt. rejection monitors iff*

$$\forall m \in \text{RMon}.\ \exists\ \varphi \in \mathcal{L} \quad \textit{such that } \textbf{\textit{mon}}(m, \varphi).$$

We show that the language sHML (Def. 6) is rejection expressive-complete. We do so with the aid of a mapping function from a rejection monitor to a corresponding formula in sHML defined below. Def. 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 Def. 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 Def. 7.

Definition 9 (Rejection Monitors to sHML Formulae).

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

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

Proposition 2. sHML *is Rejection Expressive-Complete.*

Proof. We show that for all $m \in \text{RMon}$, $\textbf{mon}(m, \langle\!\langle m \rangle\!\rangle)$ holds. $\square$

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

Proposition 3. cHML *is Acceptance Expressive-Complete.*

Equipped with Prop. 2 and Prop. 3, 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. uni-verdict monitors, iff* $\forall m \in \text{AMon} \cup \text{RMon}.\ \exists\ \varphi \in \mathcal{L}.\ \textbf{\textit{mon}}(m, \varphi)$.

Theorem 3. mHML *is Expressive-Complete.*

Proof. Follows from Prop. 2 and Prop. 3 $\square$

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.*, Thm. 4. First, however, we define what we understand by language inclusion up to formula semantic equivalence, Def. 12.

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

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

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

Proposition 4. *For any* $\mathcal{L} \subseteq \mu\text{HML}$*:*

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

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

$$\exists m \in \text{AMon}.\ \big(\forall p.(\textbf{acc}(p, m) \text{ iff } p \in [\![\varphi]\!])\big). \tag{1}$$

By Prop. 3, for the monitor m used in (1), we also know

$$\exists \pi \in \text{cHML}.\ \big(\forall p.(\textbf{acc}(p, m) \text{ iff } p \in [\![\pi]\!])\big). \tag{2}$$

Assume an arbitrary $p \in [\![\varphi]\!]$. By (1) we obtain $\textbf{acc}(p, m)$, and by (2) we obtain $p \in [\![\pi]\!]$. Thus $[\![\varphi]\!] \subseteq [\![\pi]\!]$. Dually, we can also reason that $[\![\pi]\!] \subseteq [\![\varphi]\!]$. $\square$

Theorem 4 (Completeness). *($\mathcal{L} \subseteq \mu\text{HML}$ is monitorable) implies* $\mathcal{L} \sqsubseteq \text{mHML}$.

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

$$\forall \varphi \in \mathcal{L}.\ \exists m \text{ such that } \mathbf{smon}(m, \varphi) \text{ and } \mathbf{cmon}(m, \varphi) \tag{3}$$

By Thm. 2, we know that every m used in (3) 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:

$$\big(\forall \varphi \in \mathcal{L}_{\text{acc}}.\ \exists m \in \text{AM{\scriptsize ON}}.\ \forall p.(\mathbf{acc}(p, m) \text{ iff } p \in [\![\varphi]\!])\big) \tag{4}$$

$$\big(\forall \varphi \in \mathcal{L}_{\text{rej}}.\ \exists m \in \text{RM{\scriptsize ON}}.\ \forall p.(\mathbf{rej}(p, m) \text{ iff } p \notin [\![\varphi]\!])\big) \tag{5}$$

By (4), (5) and Prop. 4 we obtain $\mathcal{L}_{\text{acc}} \sqsubseteq$ cHML and $\mathcal{L}_{\text{rej}} \sqsubseteq$ sHML *resp.*, from which the required result follows. □

Thm. 2 and Thm. 4 constitute powerful results *wrt.* the monitorability of our branching-time logic. Completeness, Thm. 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.[5] 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, Thm. 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 [7], generally carry *higher overheads* than uni-verdict monitors.

Example 10. By virtue of Thm. 4, we can conclude that properties φ_1, φ_2, φ_3, φ_5 and φ_6 from Ex. 2 are all non-monitorable properties according to Def. 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 Def. 5 (Monitorability) — through Def. 2 and Def. 3 — universally quantifies over all processes, which generally poses problems for automation.

For instance, in the case of φ_1, we could use Thm. 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 Ex. 6). 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 (Def. 3) it should reject the process $\mathsf{req.nil} + \mathsf{cls.nil}$ since this process does not satisfy φ_1: by Prop. 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

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

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. ■

8 Conclusion

We have investigated monitorability aspects of a branching-time logic called μHML, which impinges on what properties can be verified at runtime. It extends and generalises prior work carried out in the context of reactive systems modelled as LTSs [16]. The concrete contributions of the paper are:

1. An operational definition of monitorability, Def. 4, specified over an instrumentation relation unifying the individual behaviour of processes and monitors, Fig. 4, which is used to define monitorable subsets of μHML, Def. 5.
2. The identification of a subset of μHML, Def. 6, that is shown to be monitorable, Thm. 1, and also maximally expressive, Thm. 4, *wrt.* Def. 5.
3. A result asserting that, *wrt.* Def. 4, uni-verdict monitors suffice for monitoring branching-time properties, Thm. 2.

Future Work: It is worth exploring other definitions of monitorability apart from that of Def. 4, and determining how this affects the monitorable subset of μHML identified in this work. For instance, one could relax the conditions of Def. 4 by only requiring soundness (Def. 2), or require more stringent conditions *wrt.* verdicts and monitor non-determinism; see [16] for a practical motivation of this. Moreover, monitorability is also largely dependent on the underlying instrumentation relation used; there may be other sensible relations apart from the one defined in Fig. 4 that are worth investigating within this setting.

A separate line of research could investigate manipulation techniques that decompose formulae into monitorable components. For instance, reformulating a generic formula φ as the disjunction $\phi \vee \pi$ (recall $\pi \in$ cHML) could allow for a *hybrid* verification approach that distributes the load between the pre-deployment phase and the runtime phase whereby we model-check for the satisfaction of a system *wrt.* ϕ and, if this fails, runtime verify the system *wrt.* π.

Related Work: In [23], 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. Falcone *et al.* [14] revisit this classical definition, extending 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 the different definitions.

The logic μHML has been previously studied from a linear-time perspective in [2, 9], in order to find subsets that characterise may/must testing equivalences. Although tests are substantially different from our monitor instrumentations, the logic subsets identified in [2, 9] are related to (albeit different from) mHML.

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. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
5. 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.
6. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *Logic and Comput.*, 20(3):651–674, 2010.
7. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *TOSEM*, 20(4):14, 2011.
8. I. Cassar and A. Francalanza. On Synchronous and Asynchronous Monitor Instrumentation for Actor Systems. In *FOCLASA*, volume 175, pages 54–68, 2014.
9. A. Cerone and M. Hennessy. Process behaviour: Formulae vs. tests. In *EXPRESS*, volume 41 of *EPTCS*, pages 31–45, 2010.
10. E. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In *ALP LNCS*, pages 474–486. Springer-Verlag, 1992.
11. C. Cini and A. Francalanza. An LTL Proof System for Runtime Verification. In *TACAS*, volume 9035, pages 581–595. Springer, 2015.
12. C. Colombo, G. Pace, and G. Schneider. LARVA — Safer monitoring of Real-Time Java programs (Tool paper). In *SEFM*, pages 33–37, 2009.
13. 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.
14. Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.
15. A. Francalanza, A. Gauci, and G. J. Pace. Distributed System Contract Monitoring. *JLAP*, 82(5-7):186–215, 2013.
16. A. Francalanza and A. Seychell. Synthesising Correct concurrent Runtime Monitors. *Formal Methods in System Design (FMSD)*, pages 1–36, 2014.
17. M. Geilen. On the Construction of Monitors for Temporal Logic Properties. In *RV*, volume 55 of *ENTCS*, pages 181–199, 2001.
18. C. A. R. Hoare. *Communicating sequential processes.* Prentice-Hall, 1985.
19. D. Kozen. Results on the propositional μ-calculus. *TCS*, 27:333–354, 1983.
20. M. Leucker and C. Schallhart. A brief account of Runtime Verification. *JLAP*, 78(5):293 – 303, 2009.
21. Z. Manna and A. Pnueli. Completing the Temporal Picture. *TCS*, 83(1):97–130, 1991.
22. R. Milner. *A Calculus of Communicating Systems.* Springer, 1982.
23. A. Pnueli and A. Zaks. Psl model checking and run-time verification via testers. In *FM*, pages 573–586. Springer, 2006.
24. K. Sen, G. Rosu, and G. Agha. Generating optimal linear temporal logic monitors by coinduction. In *ASIAN*, LNCS, pages 260–275. Springer, 2004.
25. detectEr Project. `http://www.cs.um.edu.mt/svrg/Tools/detectEr/`.