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

Synthesising Correct Concurrent Runtime Monitors

Adrian Francalanza · Aldrin Seychell

the date of receipt and acceptance should be inserted later

Abstract This paper studies the correctness of automated synthesis for concurrent monitors. We adapt a subset of the Hennessy-Milner logic with recursion (a reformulation of the modal μ-calculus) to specify safety properties for Erlang programs. We also define an automated translation from formulas in this sub-logic to concurrent Erlang monitors that detect formula violations at runtime. Subsequently, we formalise a novel definition for monitor correctness that incorporates monitor behaviour when instrumented with the program being monitored. Finally, we devise a sound technique that allows us to prove monitor correctness in stages; this technique is used to prove the correctness of our automated monitor synthesis.

Keywords runtime verification · automated monitor synthesis · monitor correctness · mu-calculus · concurrency · actors

PACS PACS code1 · PACS code2 · more

Mathematics Subject Classification (2000) MSC code1 · MSC code2 · more

The research work disclosed in this publication is partially funded by the Strategic Educational Pathways Scholarship Scheme (Malta). The scholarship is part nanced by the European Union European Social Fund.

Adrian Francalanza
Dept. of Computer Science,
ICT, University of Malta.
Tel.: +356-2340-2745
Fax: +356-2132-0539
E-mail: afra1@um.edu.mt

Aldrin Seychell
Dept. of Computer Science,
ICT, University of Malta.
E-mail: aldrin.seychell.08@um.edu.mt

1 Introduction

Runtime Verification (RV), is a lightweight verification technique that may be used to determine whether the *current system run* respects a correctness property. Two requirements are crucial for the adoption of this technique. First, runtime *monitor overheads* need to be kept to a minimum so as not to degrade system performance. Second, instrumented monitors need to form part of the trusted computing base of the verification setup by adhering to an agreed notion of *monitor correctness*; amongst other things, this normally includes a guarantee that runtime checking corresponds (in some sense) to the property being checked for. Monitor overheads and correctness are occasionally conflicting concerns. For instance, in order to lower monitoring overheads, engineers increasingly use *concurrent monitors* [16, 29,35] so as to benefit from the underlying parallel and distributed architectures. However concurrent monitors are also more susceptible to elusive errors such as non-deterministic behaviour, deadlocks or livelocks which may, in turn, affect their correctness.

Ensuring monitor correctness is, in general, non-trivial. One prominent obstacle is the fact that system properties are typically specified using one formalism, *e.g.*, a high-level logic, whereas the respective monitors that check these properties are described using another formalism, *e.g.*, a programming language. This, in turn, makes it hard to ascertain the semantic correspondence between the two descriptions. *Automated monitor synthesis* can mitigate this problem by standardising the translation from the property logic to the monitor formalism. It also gives more scope for a formal treatment of monitor correctness.

In this work, we investigate the correctness of *synthesised* monitors in a *concurrent setting*, whereby (*i*) the system being verified executes concurrently with the synthesised monitor (*ii*) the system and the monitor themselves consist of concurrent sub-components. The correctness of monitor synthesis has been studied previously by the seminal work of Geilen, [23], and (more formally) by subsequent work such as that of Sen *et al.*, [34], and Bauer *et al.*, [5]. Our approach differs from these studies in a number of respects. First, the aforementioned work abstracts away from the internal working of a system, representing it as a string of events/states (execution trace); this complicates reasoning about how monitors are instrumented *wrt.* an executing system. It also focusses on a logic that is readily amenable to runtime analysis, namely Linear Temporal Logic (LTL) [13]. Moreover, it expresses synthesis in terms of *abstract* or *single-threaded* monitors—using either pseudocode [23] or automata [34,5] —executing *wrt.* such trace. By contrast, we strive towards a more intensional formal definition of online correctness for synthesised concurrent monitors whereby, for arbitrary property φ, the synthesised monitor M_φ running concurrently *wrt.* some system S (denoted as $S \parallel M_\varphi$) respects the following condition:

$$
\begin{array}{l}
\text{Whenever } S \parallel M_\varphi \text{ executes to some } S' \parallel M' \quad \text{then} \\
\left(\begin{array}{l}
\text{If the execution from } S \text{ to } S' \text{ violates } \varphi \text{ then} \\
\qquad\qquad M' \text{ should consistently flag the violation} \\
\qquad\qquad\qquad \text{and} \\
\text{If } M' \text{ flags a violation then the execution from } S \text{ to } S' \text{ should violate } \varphi
\end{array}\right)
\end{array}
\qquad (1)
$$

The setting of (1) brings to the fore additional issues concerning monitor correctness:

(i) A property logic semantics may be defined over other computational entities apart from traces, *i.e.*, the current *execution* in (1), such as the entire computational tree of a program. As a result, the logic semantics may not readily lend itself to the formulation of

monitor correctness outlined in condition (1) above, which *only* requires monitor detection *whenever a violation occurs*. In general, a monitored system that violates a property according to the original logic semantics, may do so along one computational path but *not* along another; this is often the case for concurrent systems with multiple execution paths as a result of different thread interleavings scheduled at runtime.

(ii) Concurrent monitors may also have multiple execution paths. Condition (1) thus requires stronger guarantees than those for single-threaded monitors so as to ensure that *all* these paths correspond to an appropriate runtime check of system property being monitored. Stated otherwise, correct concurrent monitors must *always/consistently* detect violations and flag them, irrespective of their runtime interleaving.

(iii) Apart from the formal semantics of the source logic (specifying property φ), we also require a formal semantics for the target languages of *both* the system and the monitor executing in *side-by-side*, *i.e.*, $S \parallel M_\varphi$. In most cases, the latter may not be available.

(iv) Online monitor correctness needs to ensure that monitor execution cannot be interfered with by the system, and viceversa. Whereas adequate monitor instrumentation typically prevents direct interferences, condition (1) must consider *indirect* interferences such as system divergences [32,25], *i.e.*, infinite internal looping making the system unresponsive, which may prevent the monitors from progressing.

(v) Ensuring correctness along the lines of (1) can be quite onerous because *every* execution path of the monitor running concurrently with the monitored system, $S \parallel M_\varphi$, needs to be analysed to ensure consistent detections along every thread interleaving. Consequently, one needs to devise scalable techniques facilitating monitor correctness analysis.

We conduct our study in terms of actor-based [26] concurrent monitors written in Erlang [11,3], an industry-strength language for constructing fault-tolerant systems. To alleviate our technical development, we also restrict monitoring to systems written in the same language. We limit ourselves to a logic that describes *reactive properties*, *i.e.*, *system interactions with the environment*, and focus on the synthesis of *asynchronous* monitors, performing runtime analysis through the Erlang Virtual Machine (EVM)'s tracing mechanism. Despite the typical drawbacks associated with asynchrony, *e.g.*, late detections, our monitoring setup is in line with the asynchrony advocated by the actor concurrency model, which facilitates scalable coding techniques such as fail-fast design patterns [11]. Asynchronous monitoring has also been used in other RV tools, *e.g.*, [14,17], and has proved to be less intrusive and easier to instrument than synchronous monitoring setups. It is also the more feasible alternative when monitoring distributed systems [19,16]. More importantly, though, it still allows us to investigate the main issues arising from the correctness setup outlined in (1), and we expect most of the issues investigated to carry over in a straightforward fashion to purely synchronous settings.

As an expository logic for describing reactive properties, we consider an adaptation of sHML [1] — a syntactic subset of the more expressive μ-calculus logic — describing safety, *i.e.*, monitorable [28], properties. Our choice for this logic was, in part, motivated by the fact that the full μ-calculus had already been adapted to describe concurrent Erlang program behaviour in [22], albeit for model-checking purposes. Given the usual drawbacks associated with full-blown model checking, our work contributes towards an investigation of lightweight verification techniques for μ-calculus properties of Erlang programs. It also allows us to investigate how to extend runtime verification techniques to logics that were not originally intended for this verification setup. More precisely, sHML is a syntactic subset of a larger logic used to specify *branching-time* properties for concurrent systems [2]; our work can thus be used as a first step towards a broader investigation of which other subsets of this

Actor Systems, Expressions, Values and Patterns

$$\begin{array}{rcl}
A,B,C \in \textsc{Actr} & ::= & i[e \triangleleft q]^m \mid A \parallel B \mid (\nu\, i)A \\
e,d \in \textsc{Exp} & ::= & v \mid \mathsf{self} \mid e!d \mid \mathsf{rcv}\, g\, \mathsf{end} \mid e(d) \mid \mathsf{spw}\, e \mid \mathsf{case}\, e\, \mathsf{of}\, g\, \mathsf{end} \mid x = e, d \mid \ldots \\
v,u \in \textsc{Val} & ::= & x \mid i \mid a \mid \mu y.\lambda x.e \mid \{v,\ldots,v\} \mid l \mid \mathsf{exit} \mid \ldots \\
p,o \in \textsc{Pat} & ::= & x \mid i \mid a \mid \{p,\ldots,p\} \mid \mathsf{nil} \mid p : x \mid \ldots
\end{array}
\qquad
\begin{array}{rcl}
q,r \in \text{MB}\textsc{ox} & ::= & \epsilon \mid v : q \\
l,k \in \textsc{Lst} & ::= & \mathsf{nil} \mid v : l \\
g,f \in \text{PL}\textsc{st} & ::= & \epsilon \mid p \rightarrow e; g
\end{array}$$

Fig. 1: Erlang Syntax

branching-time logic can actually be verified at runtime. Crucially, however, sHML acts as an adequate vehicle to study the monitor correctness issues set out in (1) above.

The rest of the paper is structured as follows. Sec. 2 discusses the formal semantics of our systems and monitor target language. Sec. 3 discusses reformulations to the logic facilitating the formulation of monitor correctness, discussed later in Sec. 4. Sec. 5 describes a synthesis algorithm for the logic and a tool built using the algorithm. Subsequently, Sec. 6 proves the correctness of this monitor synthesis. Sec. 8 concludes.

2 The Language

We require a formal semantics for both our monitor-synthesis target language, and the systems we intend to monitor. We partially address this problem by expressing both monitors and systems in terms of the same language, *i.e.*, Erlang, thus only requiring one semantics. However, we still need to describe the Erlang tracing semantics we intend to use for our asynchronous monitoring. Although Erlang semantic formalisations exist, *e.g.*, [36,22,8], none describe this tracing mechanism. We therefore define a calculus—following [36,22]—for modelling the *tracing semantics* of a (Turing-complete) subset of the Erlang language (we leave out distribution, process linking and fault-trapping mechanisms).

Figure 1 outlines the language syntax, assuming disjoint denumerable sets of process/actor identifiers $i, j, h \in \textsc{Pid}$, atoms $a, b \in \textsc{Atom}$, and variables $x, y, z \in \textsc{Var}$. An executing Erlang program is made up of a *system of actors*, $\textsc{Actr}$, composed in *parallel*, $A \parallel B$, where some identifiers are local (scoped) to subsystems of actors, and thus not known to the environment, *e.g.*, i in a system $A \parallel (\nu\, i)B$. Individual actors, denoted as $i[e \triangleleft q]^m$, are uniquely identified by an identifier, i, and consist of an expression, e, executing *wrt.* a local mailbox, q (denoted as a list of values); as we explain later, m denotes the actor monitoring modality. Actor *expressions* typically consist of a sequence of variable binding $x_i = e_i,$ terminated by an expression, e_{final}:

$$x_1 = e_1,\ \ldots,\ x_n = e_n,\ e_{\text{final}}$$

An expression e_i in a binding $x_i = e_i, e_{i+1}$ is expected to evaluate to a value, v, which is then bound to x_i in the continuation expression e_{i+1}. When instead e_i generates an exception, exit, it aborts subsequent computations[1] in e_{i+k} for $1 \leq k \leq (n - i)$. Apart from bindings, expressions may also consist of self references (to the actor's own identifier), self, outputs to other actors, $e_1!e_2$, pattern-matching inputs from the mailbox, $\mathsf{rcv}\, g\, \mathsf{end}$, or pattern-matching for case-branches, $\mathsf{case}\, e\, \mathsf{of}\, g\, \mathsf{end}$ (where g is a list of expressions guarded by patterns, $p_i \rightarrow e_i$), function applications, $e_1(e_2)$, and actor-spawning, $\mathsf{spw}\, e$, amongst others. Values consist of variables, x, process ids, i, recursive functions, $\mu y.\lambda x.e$ (where the preceding μy denotes the binder for function self-reference), tuples $\{v_1, \ldots, v_n\}$ and lists, l, amongst others.

[1] Due to exit exceptions, variable bindings, $x = e, d$ cannot be encoded as function applications, $\lambda x.d(e)$.

Remark 1 The functions $\mathrm{fv}(A)$ and $\mathrm{fId}(A)$ return the free variables and free process identifiers of A *resp.* and are defined in standard fashion. We write $\lambda x.e$ and d, e for $\mu y.\lambda x.e$ and $y = d, e$ *resp.* when $y \notin \mathrm{fv}(e)$. In $p \to e$, we replace x in p with $\_$ whenever $x \notin \mathrm{fv}(e)$. We use standard shorthand for lists of binders and write $\mu y.\lambda(x_1, \dots x_n).e$ and $(\nu\, i_1, \dots, i_n)\, A$ *resp.* for $\mu y.\lambda x_1 \dots \lambda x_n.e$ and $(\nu\, i_1) \dots (\nu\, i_n)\, A$. We sometimes elide mailboxes and write $i[e]$, when these are empty, $i[e \triangleleft \epsilon]$, or when they do not change in the transition rules that follow.

Specific to our formalisation, we also subject each individual actor, $i[e \triangleleft q]^m$, to a *monitoring-modality*, $m, n \in \{\circ, \bullet, *\}$, where $\circ$, $\bullet$ and $*$ denote *monitored*, *unmonitored* and *tracing* actors *resp.* Modalities play a crucial role in our language semantics, defined as a labelled transition system over systems, $A \xrightarrow{\gamma} B$, where actions $\gamma \in \mathrm{Act}_\tau$, include bound *output* labels, $(\tilde{h})i!v$, and *input* labels, $i?v$ and a distinguished *internal* label, τ. In line with the reactive properties we consider later, our formalisation only traces system interactions with the environment (send and receive messages) from *monitored* actors. Thus, whereas unmonitored, $\bullet$, and tracing, $*$, actors have standard input and output transition rules

$$\textsc{SndU}\ \frac{m \in \{\bullet, *\}}{j[i!v \triangleleft q]^m \xrightarrow{i!v} j[v \triangleleft q]^m} \qquad \textsc{RcvU}\ \frac{m \in \{\bullet, *\}}{i[e \triangleleft q]^m \xrightarrow{i?v} i[e \triangleleft q{:}v]^m}$$

actors with a monitored modality, $\circ$, *i.e.*, actors j and i in rules SndM and RcvM below, produce a residual message reporting the send and receive interactions ($\{\mathsf{sd}, i, v\}$ and $\{\mathsf{rv}, i, v\}$ *resp.*) at the tracer's mailbox *i.e.*, actor h with modality $*$ in the rules below; this models closely the tracing mechanism offered by the Erlang Virtual Machine (EVM) [11]. In our target language, the list of report messages at the tracer's mailbox constitutes the system trace to be used for asynchronous monitoring.

$$\textsc{SndM}\ \frac{}{j[i!v \triangleleft q]^\circ \parallel h[d \triangleleft r]^* \xrightarrow{i!v} j[v \triangleleft q]^\circ \parallel h[d \triangleleft r{:}\{\mathsf{sd}, i, v\}]^*}$$

$$\textsc{RcvM}\ \frac{}{i[e \triangleleft q]^\circ \parallel h[d \triangleleft r]^* \xrightarrow{i?v} i[e \triangleleft q{:}v]^\circ \parallel h[d \triangleleft r{:}\{\mathsf{rv}, i, v\}]^*}$$

Our LTS semantics assumes *well-formed* actor systems, whereby every actor identifier is *unique*; it is termed to be a *tracing semantics* because a distinguished *tracer* actor, identified by the monitoring modality $*$, receives messages recording external communication events by *monitored* actors. Formally, we write $A \xrightarrow{\gamma} B$ in lieu of $\langle A, \gamma, B\rangle \in \longrightarrow$, the least ternary relation satisfying the rules in Fig. 2. These rules employ *evaluation contexts*, denoted as C (described below) specifying which sub-expressions are active. For instance, an expression is only evaluated when at the top level variable binding, $x = C, e$ or when the expression denoting the destination of an output has evaluated to a value, $v!C$; the other cases are also fairly standard.[2] We denote the application of a context C to an expression e as $C[e]$.

$$C \ ::= \quad [-] \mid C!e \mid v!C \mid C(e) \mid v(C) \mid \mathsf{case}\, C\, \mathsf{of}\, g\, \mathsf{end} \mid x = C, e \mid \dots$$

Communication in actor systems happens in two stages: an actor receives messages, keeping them in order in its mailbox, and then selectively reads them at a later stage using pattern-matching—rules Rd1 and Rd2 describe how mailbox messages are traversed in order to find

[2] In our formalisation, expressions are not allowed to evaluate under a spawn context, $\mathsf{spw}\,[-]$; this aspect differs from standard Erlang semantics but allows a lightweight description of function application spawning. An adjustment in line with the actual Erlang spawning would be straightforward.

$$\textsc{SndM}\;\frac{}{j[C[i!v] \triangleleft q]^{\circ} \parallel h[d \triangleleft r]^{*} \xrightarrow{i!v} j[C[v] \triangleleft q]^{\circ} \parallel h[d \triangleleft (r{:}\{\mathsf{sd}, i, v\})]^{*}}$$

$$\textsc{RcvM}\;\frac{\mathrm{fv}(v) = \emptyset}{i[e \triangleleft q]^{\circ} \parallel h[d \triangleleft r]^{*} \xrightarrow{i?v} i[e \triangleleft q{:}v]^{\circ} \parallel h[d \triangleleft (r{:}\{\mathsf{rv}, i, v\})]^{*}}$$

$$\textsc{SndU}\;\frac{m \in \{\bullet, *\}}{j[C[i!v] \triangleleft q]^{m} \xrightarrow{i!v} j[C[v] \triangleleft q]^{m}} \qquad \textsc{RcvU}\;\frac{m \in \{\bullet, *\} \quad \mathrm{fv}(v) = \emptyset}{i[e \triangleleft q]^{m} \xrightarrow{i?v} i[e \triangleleft q{:}v]^{m}}$$

$$\textsc{Scp}\;\frac{A \xrightarrow{\gamma} B}{(\nu\, j)A \xrightarrow{\gamma} (\nu\, j)B}\; j \notin (\mathrm{obj}(\gamma) \cup \mathrm{sbj}(\gamma)) \qquad \textsc{Opn}\;\frac{A \xrightarrow{(\tilde{h})i!v} B}{(\nu\, j)A \xrightarrow{(j,\tilde{h})i!v} B}\; i \neq j,\, j \in \mathrm{sbj}((\tilde{h})i!v)$$

$$\textsc{Com}\;\frac{}{j[C[i!v] \triangleleft q]^{m} \parallel i[e \triangleleft q]^{n} \xrightarrow{\tau} j[C[v] \triangleleft q]^{m} \parallel i[e \triangleleft q{:}v]^{n}}$$

$$\textsc{Par}\;\frac{A \xrightarrow{\gamma} A'}{A \parallel B \xrightarrow{\gamma} A' \parallel B}\;\mathrm{obj}(\gamma) \cap \mathrm{fId}(B) = \emptyset \qquad \textsc{Rd1}\;\frac{\mathrm{mtch}(g, v) = e}{i[C[\mathsf{rcv}\, g\, \mathsf{end}] \triangleleft (v : q)]^{m} \xrightarrow{\tau} i[C[e] \triangleleft q]^{m}}$$

$$\textsc{Rd2}\;\frac{\mathrm{mtch}(g, v) = \bot \qquad i[C[\mathsf{rcv}\, g\, \mathsf{end}] \triangleleft q]^{m} \xrightarrow{\tau} i[C[e] \triangleleft r]^{m}}{i[C[\mathsf{rcv}\, g\, \mathsf{end}] \triangleleft (v : q)]^{m} \xrightarrow{\tau} i[C[e] \triangleleft (v : r)]^{m}}$$

$$\textsc{Cs1}\;\frac{\mathrm{mtch}(g, v) = e}{i[C[\mathsf{case}\, v\, \mathsf{of}\, g\, \mathsf{end}]]^{m} \xrightarrow{\tau} i[C[e]]^{m}} \qquad \textsc{Cs2}\;\frac{\mathrm{mtch}(g, v) = \bot}{i[C[\mathsf{case}\, v\, \mathsf{of}\, g\, \mathsf{end}]]^{m} \xrightarrow{\tau} i[C[\mathsf{exit}]]^{m}}$$

$$\textsc{Ass}\;\frac{v \neq \mathsf{exit}}{i[C[x = v, e]]^{m} \xrightarrow{\tau} i[C[e\{^{v}/_{x}\}]]^{m}} \qquad \textsc{Ext}\;\frac{}{i[C[x = \mathsf{exit}, e]]^{m} \xrightarrow{\tau} i[C[\mathsf{exit}]]^{m}}$$

$$\textsc{App}\;\frac{}{i[C[\mu y.\lambda x.e\,(v)]]^{m} \xrightarrow{\tau} i[C[e\{^{\mu y.\lambda x.e}/_{y}\}\{^{v}/_{x}\}]]^{m}} \qquad \textsc{Slf}\;\frac{}{i[C[\mathsf{self}]]^{m} \xrightarrow{\tau} i[C[i]]^{m}}$$

$$\textsc{Spw}\;\frac{(m = \circ = n) \text{ or } (n = \bullet)}{i[C[\mathsf{spw}\, e] \triangleleft q]^{m} \xrightarrow{\tau} (\nu\, j)(i[C[j] \triangleleft q]^{m} \parallel j[e \triangleleft \epsilon]^{n})} \qquad \textsc{Str}\;\frac{A \equiv A' \xrightarrow{\gamma} B' \equiv B}{A \xrightarrow{\gamma} B}$$

$$\textsc{sCom}\;\frac{}{A \parallel B \equiv B \parallel A} \qquad \textsc{sAss}\;\frac{}{(A \parallel B) \parallel C \equiv A \parallel (B \parallel C)} \qquad \textsc{sCtxP}\;\frac{A \equiv B}{A \parallel C \equiv B \parallel C}$$

$$\textsc{sExt}\;\frac{i \notin \mathrm{fId}(A)}{A \parallel (\nu\, i)B \equiv (\nu\, i)(B \parallel A)} \qquad \textsc{sSwp}\;\frac{}{(\nu\, i)(\nu\, j)A \equiv (\nu\, j)(\nu\, i)A} \qquad \textsc{sCtxS}\;\frac{A \equiv B}{(\nu i)A \equiv (\nu i)B}$$

Fig. 2: Erlang Semantics for Actor Systems

the first one matching a pattern in the pattern list g, releasing the respective guarded expression e as a result. We choose only to record *external* communication at tracer processes, *i.e.*, between the system and the environment, and do *not* trace internally communication between actors within the system, irrespective of their modality (see Com); structural equivalence rules, $A \equiv B$, are employed to simplify the presentation of our rules—see rule Str and the corresponding structural rules. In Par, the side-condition enforces the *single-receiver* property, inherent to actor systems; for instance, it prevents a transition with an action $j!v$ when actor j is part of the actor system B. We note that in rule Spw, spawned actors inherit monitorability when launched by a monitored actor, but are launched as unmonitored otherwise.

Mailbox reading—defined by the rules Rd1 and Rd2 in Fig. 2—includes pattern-matching functionality, allowing the actor to selectively choose which messages to read first from its mailbox whenever the first pattern $p \rightarrow e$ from the pattern list g is matched, returning $e\sigma$, where σ substitutes free variables in e for value binding resulting from the pattern-match; when no pattern is matched, mailbox reading *blocks* - see Definition 1.

Definition 1 (Pattern-Matching) We define mtch : PLst $\times$ Val $\rightarrow$ Exp$_\perp$ and vmtch : Pat $\times$ Val $\rightarrow$ Sub$_\perp$ as follows:

$$\mathrm{mtch}(g, v) \overset{\mathbf{def}}{=} \begin{cases} e\sigma & \text{if } g = p \rightarrow e : f, \mathrm{vmtch}(p, v) = \sigma \\ d & \text{if } g = p \rightarrow e : f, \mathrm{vmtch}(p, v) = \perp, \mathrm{mtch}(f, v) = d \\ \perp & \text{otherwise} \end{cases}$$

$$\mathrm{vmtch}(p, v) \overset{\mathbf{def}}{=} \begin{cases} \emptyset & \text{if } p = v \text{ (whenever } p \text{ is } a, i \text{ or } \mathsf{nil}) \\ \{v/x\} & \text{if } p = x \\ \biguplus_{i=1}^{n} \sigma_i & \text{if } p = \{p_1, \ldots, p_n\}, v = \{v_1, \ldots, v_n\} \text{ where } \mathrm{vmtch}(p_i, v_i) = \sigma_i \\ \sigma \uplus \{l/x\} & \text{if } p = o : x, v = u : l \text{ where } \mathrm{vmtch}(o, u) = \sigma \\ \perp & \text{otherwise} \end{cases}$$

$$\sigma_1 \uplus \sigma_2 \overset{\mathbf{def}}{=} \begin{cases} \sigma_1 \cup \sigma_2 & \text{if } \mathrm{dom}(\sigma_1) \cap \mathrm{dom}(\sigma_2) = \emptyset \\ \sigma_1 \cup \sigma_2 & \text{if } \forall v \in \mathrm{dom}(\sigma_1) \cap \mathrm{dom}(\sigma_2). \sigma_1(v) = \sigma_2(v) \\ \perp & \text{if } \sigma_1 = \perp \text{ or } \sigma_2 = \perp \\ \perp & \text{otherwise} \end{cases}$$

Branching for mailbox pattern-matching differs from pattern-match branching for the case construct, described by the rules Cs1 and Cs2 in Fig. 2: similar to the mailbox read construct, it matches a value to the first appropriate pattern in the pattern list, launching the respective guarded expression with the appropriate variable bindings resulting from the pattern-match; if, however, no match is found it generates an exception, exit, which aborts subsequent computation, Ext. The rest of the transition rules, such as App for function application and Slf for retreiving the actor name, are fairly standard.

Remark 2 Our tracing semantics sits at a higher level of abstraction than that offered by the EVM [11] because trace entries typically contain more information. For instance, the EVM records internal communication between monitored actors, as an output trace entry *immediately followed by* the corresponding input trace entry; we here describe *sanitised* traces whereby consecutive matching trace entries are filtered out.

Example 1 (Non-deterministic behaviour) Our systems exhibit non-deterministic behaviour through either internal or external choices [30,25]. Consider the actor system:

$$A \triangleq (\nu\, j_1, j_2, h)(\, i[\mathsf{rcv}\; x \rightarrow \mathsf{obs}!x\, \mathsf{end} \triangleleft \epsilon]^\circ \;\parallel\; j_1[i!v]^\circ \;\parallel\; j_2[i!u]^\circ \;\parallel\; h[e \triangleleft q]^*\,)$$

Actors j_1, j_2 and h are local, *i.e.*, scoped through the construct $(\nu\, j_1, j_2, h)(\ldots)$, thus not visible to the environment. The monitored actor i may receive value v internally from actor j_1,

(2) by rule Com, read it from its mailbox, (3) by Rd1, and then output it to some environment actor obs, (4) by SndM, while recording this *external* output at h's mailbox (the tracer).

$$
\begin{aligned}
A &\xrightarrow{\tau} (\nu\, j_1, j_2, h)(\, i[\mathsf{rcv}\; x \rightarrow \mathsf{obs}!x\,\mathsf{end} \triangleleft v]^{\circ} \parallel j_1[v]^{\circ} \parallel j_2[i!u]^{\circ} \parallel h[e \triangleleft q]^{*}\,) && (2)\\
&\xrightarrow{\tau} (\nu\, j_1, j_2, h)(\, i[\mathsf{obs}!v \triangleleft \epsilon]^{\circ} \parallel j_1[v]^{\circ} \parallel j_2[i!u]^{\circ} \parallel h[e \triangleleft q]^{*}\,) && (3)\\
&\xrightarrow{\mathsf{obs}!v} (\nu\, j_1, j_2, h)(\, i[v \triangleleft \epsilon]^{\circ} \parallel j_1[v] \parallel j_2[i!u] \parallel h[e \triangleleft q : \{\mathsf{sd}, \mathsf{obs}, v\}]^{*}\,) && (4)
\end{aligned}
$$

But if actor j_2 sends its value to i before j_1, we observe a different external behaviour

$$
A \xrightarrow{\tau} \cdot \xrightarrow{\tau} \cdot \xrightarrow{\mathsf{obs}!u} (\nu\, j_1, j_2, h)(\, i[u \triangleleft \epsilon]^{\circ} \parallel j_1[i!v] \parallel j_2[u] \parallel h[e \triangleleft q : \{\mathsf{sd}, \mathsf{obs}, u\}]^{*}\,)
$$

i.e., A eventually outputs u instead of v to obs (accordingly monitor h would hold the entry $\{\mathsf{sd}, \mathsf{obs}, u\}$ instead); these behaviours amount to an *internal choice*.

External choice results when A receives different external inputs: we can derive

$$
A \xrightarrow{i?v_1} B_1 \triangleq (\nu j_1, j_2, h)(i[\mathsf{rcv}\; x \rightarrow \mathsf{obs}!x\,\mathsf{end} \triangleleft v_1]^{\circ} \parallel j_1[i!v]^{\circ} \parallel j_2[i!u]^{\circ} \parallel h[e \triangleleft q : \{\mathsf{rv}, i, v_1\}]^{*})
$$

but also $A \xrightarrow{i?v_2} B_2$ for some appropriate B_2. Subsequently, B_1 can only produce the output $B_1 \xrightarrow{\tau}{}^{*} \cdot \xrightarrow{\mathsf{obs}!v_1} B_3$ whereas from B_2 can only produce $B_2 \xrightarrow{\tau}{}^{*} \cdot \xrightarrow{\mathsf{obs}!v_2} B_4$. Note that, in the first case, h's mailbox in B_3 is appended by entries $\{\mathsf{rv}, i, v_1\} : \{\mathsf{sd}, \mathsf{obs}, v_1\}$ whereas, in the second case, h's mailbox in B_4 is appended by $\{\mathsf{rv}, i, v_2\} : \{\mathsf{sd}, \mathsf{obs}, v_2\}$. ■

Example 2 (Infinite Behaviour) Our systems may exhibit infinite behaviour. Actor A (below) may produce an infinite number of output actions $j!v$ (for $v \neq \mathsf{exit}$) through repeated sequences of function applications, (5) by rule App, outputs, (6) by rule SndU and variable assignments, (7) by rule Ass.

$$
\begin{aligned}
A \triangleq i[(\mu y.\lambda x.z = j!x, y(z))(v) \triangleleft q]^{\bullet} &\xrightarrow{\tau} i[z = j!v, (\mu y.\lambda x.z' = j!x, y(z'))(z) \triangleleft q]^{\bullet} && (5)\\
&\xrightarrow{j!v} i[z = v, (\mu y.\lambda x.z' = j!x, y(z'))(z) \triangleleft q]^{\bullet} && (6)\\
&\xrightarrow{\tau} i[(\mu y.\lambda x.z' = j!x, y(z'))(v) \triangleleft q]^{\bullet} \;=\; A && (7)
\end{aligned}
$$

Systems with infinite behaviour may also transition to a different system with each computational step. System B (below) is a slight modification to A that delegates the output to a newly spawned actor at each iteration. Using a similar sequence of transitions to (5), (6) and (7) together with rule Spw we are able to obtain the computation below, whereby the number of actors grow with each iteration.

$$
\begin{aligned}
B \;\triangleq i[(\mu y.\lambda x.z = \mathsf{spw}\; j!x, y(x))(v) \triangleleft q]^{\bullet}\; (\xrightarrow{\tau})^{3} \cdot \xrightarrow{j!v} (\nu\, h)(B \parallel h[v \triangleleft \epsilon]^{\bullet})\\
(\xrightarrow{\tau})^{3} \cdot \xrightarrow{j!v} (\nu\, h, h')(B \parallel h[v \triangleleft \epsilon]^{\bullet} \parallel h'[v \triangleleft \epsilon]^{\bullet})\; (\xrightarrow{\tau})^{3} \cdot \xrightarrow{j!v} \ldots
\end{aligned}
$$

Such infinite behaviour together with the non-deterministic behaviour discussed in Example 1 typically lead to state-explosion problems when systems are verified exhaustively. ■

3 The Logic

In order to specify reactive properties for the actor systems discussed in Sec. 2, we consider an adaptation of SafeHML [1] (sHML), a sub-logic of the Hennessy-Milner Logic (HML) with recursion — the latter logic has been shown to be a reformulation of the expressive μ-calculus [27]. It assumes a denumerable set of formula variables, $X, Y \in \text{LVar}$, and is inductively defined by the following grammar:

$$\varphi, \psi \in \text{sHML} \;::=\; \mathsf{ff} \;\mid\; \varphi\wedge\psi \;\mid\; [\alpha]\varphi \;\mid\; X \;\mid\; \mathsf{max}(X, \varphi)$$

The formulas for falsity, ff, conjunction, $\varphi\wedge\psi$, and action necessity, $[\alpha]\varphi$, are inherited from HML[25], whereas variables X and the recursion construct $\mathsf{max}(X, \varphi)$ are used to define *maximal* fixpoints; as expected, $\mathsf{max}(X, \varphi)$ is a binder for the free variables X in φ, inducing standard notions of open and closed formulas. We only depart from the logic of [1] by limiting formulas to *basic actions* $\alpha, \beta \in \text{BAct}$, including just input, $i?v$, and *unbound* outputs, $i!v$, so as to keep our technical development manageable.

Remark 3 The handling of *bound* output actions, $(\tilde{h})i!v$, is well understood [31] and does not pose problems to monitoring, apart from making action pattern-matching cumbersome (consult [24] for an example of how bound values can be matched); it also complicates instrumentation (see Sec. 4 and Sec. 5). Certain (silent) τ labels can also be monitored using minor adaptations (see, for instance, Remark 2); they however increase substantially the size of the traces recorded, unnecessarily cluttering the tracing semantics of Sec. 2.

The semantics of our logic is defined for closed formulas, using the operation $\varphi\{\psi/X\}$, which substitutes free occurrences of X in φ with ψ without introducing any variable capture. It is specified as the satisfaction relation of Definition 2 (adapted from [1]). In what follows, we write *weak* transitions $A \Longrightarrow B$ and $A \overset{\alpha}{\Longrightarrow} B$, for $A \overset{\tau}{\rightarrow}^{*} B$ and $A \overset{\tau}{\rightarrow}^{*} \cdot \overset{\alpha}{\rightarrow} \cdot \overset{\tau}{\rightarrow}^{*} B$ *resp.* We let $s, t \in (\text{BAct})^*$ range over *lists of basic actions* and write sequences of weak actions $A \overset{\alpha_1}{\Longrightarrow} \cdots \overset{\alpha_n}{\Longrightarrow} B$, where $s = \alpha_1, \ldots, \alpha_n$, as $A \overset{s}{\Longrightarrow} B$ (or as $A \overset{s}{\Longrightarrow}$ when B is unimportant).

Definition 2 (Satisfiability) A relation $\mathcal{R} \in \text{Actr} \times \text{sHML}$ is a satisfaction relation iff:

$$\begin{array}{rl}
(A, \mathsf{ff}) \in \mathcal{R} & \text{never} \\
(A, \varphi\wedge\psi) \in \mathcal{R} & \text{implies } (A, \varphi) \in \mathcal{R} \text{ and } (A, \psi) \in \mathcal{R} \\
(A, [\alpha]\varphi) \in \mathcal{R} & \text{implies } (B, \varphi) \in \mathcal{R} \text{ whenever } A \overset{\alpha}{\Longrightarrow} B \\
(A, \mathsf{max}(X, \varphi)) \in \mathcal{R} & \text{implies } (A, \varphi\{\mathsf{max}(X,\varphi)/X\}) \in \mathcal{R}
\end{array}$$

Satisfiability, $\models_s$, is the *largest* satisfaction relation; we write $A \models_s \varphi$ for $(A, \varphi) \in \models_s$. It follows from standard fixed-point theory that the implications of satisfaction relation are bi-implications for Satisfiability.

Example 3 (Satisfiability) Consider the safety formula

$$\varphi_{\text{safe}} \triangleq \mathsf{max}(X,\ [\alpha][\alpha][\beta]\mathsf{ff} \wedge [\alpha]X\,) \tag{8}$$

stating that a satisfying actor system should never perform a sequence of two external actions α followed by the external action β (through the subformula $[\alpha][\alpha][\beta]\mathsf{ff}$), and that this needs to hold after every α action (through $[\alpha]X$); effectively the formula states that trace sequences of α-actions *greater than two* cannot be followed by a β-action.

A system A_1 exhibiting *just* the external behaviour $A_1 \overset{\alpha\beta}{\Longrightarrow}$ satisfies φ_{safe}, as would a system A_2 with just the *infinite* behaviour $A_2 \overset{\alpha}{\Longrightarrow} A_2$. System A_3, with a trace $A_3 \overset{\alpha\alpha\beta}{\Longrightarrow}$, *does not* satisfy this property, $A_3 \not\models_{\text{s}} \varphi_{\text{safe}}$, according to Definition 2. However, this system may be capable of producing other external traces at runtime (*e.g.*, as a result of some internal choice). In fact, if A_3 exhibits the alternate behaviour $A_3 \overset{\beta}{\Longrightarrow}$, one would be *unable* to observe any violation from A_3 at runtime since the external trace β is allowed by φ_{safe}.

Since actors may violate a property along one execution but satisfy it along another, the inverse of $\models_{\text{s}}$, *i.e.*, $A \not\models_{\text{s}} \varphi$, is too coarse to be used for a definition of monitor correctness along the lines of (1) discussed earlier. We thus define a *violation relation*, Definition 3, characterising actors violating a property along a specific execution trace.

Definition 3 (Violation) The violation relation, denoted as $\models_{\text{v}}$, is the *least* relation of the form $(\text{ACTR} \times \text{BACT}^* \times \text{sHML})$ satisfying the following rules:

$$\begin{array}{rl} A, s \models_{\text{v}} \mathsf{ff} & \text{always} \\ A, s \models_{\text{v}} \varphi \wedge \psi & \text{if } A, s \models_{\text{v}} \varphi \text{ or } A, s \models_{\text{v}} \psi \\ A, \alpha s \models_{\text{v}} [\alpha]\varphi & \text{if } A \overset{\alpha}{\Longrightarrow} B \text{ and } B, s \models_{\text{v}} \varphi \\ A, s \models_{\text{v}} \mathsf{max}(X, \varphi) & \text{if } A, s \models_{\text{v}} \varphi\{^{\mathsf{max}(X,\varphi)}/_X\} \end{array}$$

We write $A, s \models_{\text{v}} \varphi$ in lieu of $(A, s, \varphi) \in \models_{\text{v}}$. It also follows from standard fixed-point theory that the constraints of the violation relation are bi-implications.

Example 4 (Violation) Recall the safety formula φ_{safe} defined in (8). Actor A_3, from Example 3, together with the witness violating trace $\alpha\alpha\beta$ violate φ_{safe}, *i.e.*, $(A_3, \alpha\alpha\beta) \models_{\text{v}} \varphi_{\text{safe}}$. However, A_3 together with trace β do not violate φ_{safe}, *i.e.*, $(A_3, \beta) \not\models_{\text{v}} \varphi_{\text{safe}}$. Definition 3 relates a violating trace with an actor *only when* that trace leads the actor to a violation: if A_3 cannot perform the trace $\alpha\alpha\alpha\beta$, by Definition 3, we have $(A_3, \alpha\alpha\alpha\beta) \not\models_{\text{v}} \varphi_{\text{safe}}$, even though the trace is prohibited by φ_{safe}. A violating trace may also lead a system to a violation before its end, *e.g.*, $(A_3, \alpha\alpha\beta\alpha) \models_{\text{v}} \varphi_{\text{safe}}$ according to Definition 3. ■

Despite the technical discrepancies between Definition 2 and Definition 3 — *e.g.*, the use of maximal versus minimal fixpoints and a differing model — we show that Definition 3 corresponds, in some sense, to the dual of Definition 2.

Theorem 1 (Correspondence) $\exists s.(A, s) \models_{v} \varphi$ *iff* $A \not\models_{s} \varphi$

Proof For the *if* case we prove the contrapositive, *i.e.*, that $\forall s.A, s \not\models_{\text{v}} \varphi$ implies $A \models_{\text{s}} \varphi$ by co-inductively showing that $\mathcal{R} = \{(A, \varphi) \mid \forall s.A, s \not\models_{\text{v}} \varphi\}$ is a satisfaction relation. The proof is by induction on the structure of φ. We outline two cases (see [20] for the rest):

$\varphi \wedge \psi$: From the definition of $\mathcal{R}$ we know that $\forall s.A, s \not\models_{\text{v}} \varphi \wedge \psi$ and, by Definition 3, this implies that $\forall s.(A, s \not\models_{\text{v}} \varphi$ and $A, s \not\models_{\text{v}} \psi)$. Distributing the universal quantification yields $\forall s.A, s \not\models_{\text{v}} \varphi$ and $\forall s.A, s \not\models_{\text{v}} \psi$, and by the definition of $\mathcal{R}$ we obtain $(A, \varphi) \in \mathcal{R}$ and $(A, \psi) \in \mathcal{R}$, as required for satisfiability relations by Definition 2.

$[\alpha]\varphi$: From the definition of $\mathcal{R}$ we know that $\forall s.A, s \not\models_{\text{v}} [\alpha]\varphi$. In particular, for all $s = \alpha t$, we know that $\forall t.A, \alpha t \not\models_{\text{v}} [\alpha]\varphi$. From Definition 3 it must be the case that whenever $A \overset{\alpha}{\Longrightarrow} B$ we have that $\forall t.B, t \not\models_{\text{v}} \varphi$, which in turn implies that $(B, \varphi) \in \mathcal{R}$ (from the definition of $\mathcal{R}$); this is the implication required by Definition 2.

For the *only-if* case we prove $\exists s.A, s \models_{\mathrm{v}} \varphi$ implies $A \not\models_{\mathrm{s}} \varphi$ by rule induction on $A, s \models_{\mathrm{v}} \varphi$. Note that $A \not\models_{\mathrm{s}} \varphi$ means that there does not exist *any* satisfiability relation including the pair (A, φ). Again we outline the main cases (see [20] for the rest):

$A, s \models_{\mathrm{v}} [\alpha]\varphi$ because $s = \alpha s', A \overset{\alpha}{\Longrightarrow} B$ and $B, s' \models_{\mathrm{v}} \varphi$: By $B, s' \models_{\mathrm{v}} \varphi$ and I.H. we obtain $B \not\models_{\mathrm{s}} \varphi$, and subsequently, by $A \overset{\alpha}{\Longrightarrow} B$, we conclude that $A \not\models_{\mathrm{s}} [\alpha]\varphi$.

$A, s \models_{\mathrm{v}} \mathsf{max}(X, \varphi)$ because $A, s \models_{\mathrm{v}} \varphi\{^{\mathsf{max}(X,\varphi)}/_X\}$: By $A, s \models_{\mathrm{v}} \varphi\{^{\mathsf{max}(X,\varphi)}/_X\}$ and I.H. we obtain $A \not\models_{\mathrm{s}} \varphi\{^{\mathsf{max}(X,\varphi)}/_X\}$ which, in turn, implies that $A \not\models_{\mathrm{s}} \mathsf{max}(X, \varphi)$. ■

Definition 3 and Theorem 1 allows us to show that *every* formula $\varphi \in$ sHML denotes a *safety language*, as defined in [28, 12].[3] Informally, this states that whenever we determine that $A \not\models_{\mathrm{s}} \varphi$ along a particular trace s, such a judgement is preserved *for all extensions* of that trace. This property, in turn, implies that all formulas in sHML are *monitorable* [5]. We formally prove this result in Theorem 2, assuming the standard notion of trace prefixes *i.e.,* $s \leq t$ iff $\exists.s'$ such that $t = ss'$.

Theorem 2 (Safety Properties and sHML) *$A, s \models_{v} \varphi$ and $s \leq t$ implies $A, t \models_{v} \varphi$*

Proof By rule induction on $A, s \models_{\mathrm{v}} \varphi$. We outline the main cases leaving the rest for [20]:

$A, s \models_{\mathrm{v}} \varphi \wedge \psi$ because $A, s \models_{\mathrm{v}} \varphi$: By $A, s \models_{\mathrm{v}} \varphi$, $s \leq t$ and I.H. we obtain $A, t \models_{\mathrm{v}} \varphi$ which, by the same rule, implies $A, t \models_{\mathrm{v}} \varphi \wedge \psi$.

$A, s \models_{\mathrm{v}} [\alpha]\varphi$ because $s = \alpha s', A \overset{\alpha}{\Longrightarrow} B$ and $B, s' \models_{\mathrm{v}} \varphi$: From $s = \alpha s'$ and $s \leq t$ we know that $s' \leq t'$ for some t' where $t = \alpha t'$. By $s' \leq t'$, $B, s' \models_{\mathrm{v}} \varphi$ and I.H. we obtain $B, t' \models_{\mathrm{v}} \varphi$ and by $A \overset{\alpha}{\Longrightarrow} B$ and $t = \alpha t'$ we derive $A, t \models_{\mathrm{v}} [\alpha]\varphi$. ■

4 Correctness

Specifying online monitor correctness is complicated by the fact that, in general, we have limited control over the behaviour of the systems being monitored. For starters, a system that does not satisfy a property may still exhibit runtime behaviour that does not violate it, as discussed earlier in the case of system A_3 of Example 3 and Example 4. We deal with system non-determinism by only requiring monitor detection when the system performs a violating execution: this can be expressed through the violation relation of Definition 3.

At runtime, a system may also interfere with the execution of monitors. Appropriate *instrumentation* can limit system effects on the monitors. In our asynchronous actor setting, *direct* interferences from the system to the monitors can be precluded by (*i*) locating the monitors at process identifiers *not* known to the system (*ii*) preventing the monitors from communicating these identifiers to the system. These measures inhibit the system's ability to send messages to the monitors.

A system may also interfere with monitor executions indirectly by *diverging*, *i.e.,* infinite internal computation (τ-transitions) without external actions. This can prevent the monitors from progressing during their execution and thus postpone indefinitely violation detections [32]. In our case, divergence is handled, in part, by the EVM itself, which guarantees fair executions for concurrent actors [11]. In settings where fair executions may be assumed, it suffices to require a weaker property for monitors, reminiscent of the condition

[3] Note that we do *not* show that sHML captures all the safety properties expressible in HML with recursion, and there are infact other formulas that specify safety properties such as tt.

in fair/should-testing [33]. Definition 4 states that, for an arbitrary basic action α, an actor system A satisfies the predicate *should-α* if, for *all* sequences of internal actions leading to some system B, there always *exists* an execution from B that can produce the action α; in the case of monitors, the external should-action is set to a reserved violation-detection action, *e.g.*, fail!.

Definition 4 (Should-α) $A \Downarrow_\alpha \stackrel{\text{def}}{=} (\, A \Longrightarrow B \quad \text{implies} \quad B \overset{\alpha}{\Longrightarrow}\,)$

We limit monitoring to *monitorable* systems, where all actors are subject to a monitorable modality, *i.e.*, modality $\circ$.

$$A \equiv (\nu \tilde{h})(i[e \triangleleft q]^m \parallel B) \quad \text{implies} \quad m = \circ$$

This guarantees that (*i*) they can be composed with a tracer actor (*ii*) all the basic actions produced by the system are recorded as trace entries at the tracer's mailbox.[4] Monitor correctness is defined for (unmonitored) *basic* systems, satisfying the condition:

$$A \equiv (\nu \tilde{h})(i[e \triangleleft q]^m \parallel B) \quad \text{implies} \quad m = \bullet$$

which are instrumented to execute in parallel with the monitor. Our instrumentation is defined through the operation $\lceil - \rceil$, Definition 5, converting basic systems to monitorable ones using `trace/2` and `set_on_spawn` Erlang commands [11]; see Lemma 1. Importantly, instrumentation does not affect the visible behaviour of a basic system; see Lemma 2.

Definition 5 (Instrumentation) $\lceil - \rceil :: \text{Actr} \rightarrow \text{Actr}$ is defined inductively as:

$$\lceil i[e \triangleleft q]^m \rceil \stackrel{\text{def}}{=} i[e \triangleleft q]^\circ \qquad \lceil B \parallel C \rceil \stackrel{\text{def}}{=} \lceil B \rceil \parallel \lceil C \rceil \qquad \lceil (\nu\, i) B \rceil \stackrel{\text{def}}{=} (\nu\, i)\lceil B \rceil$$

Lemma 1 *If A is a basic system then $\lceil A \rceil$ is monitorable.*

Lemma 2 *For all basic actors A where $i \notin \text{fId}(A)$:*

$$A \xrightarrow{\alpha} B \textit{ iff } \begin{cases} (\nu\, i)(\lceil A \rceil \parallel i[e \triangleleft q]^*) \xrightarrow{j!v} (\nu\, i)(\lceil B \rceil \parallel i[e \triangleleft q : \{\mathsf{sd}, j, v\}]^*) & \textit{if } \alpha = j!v \\ (\nu\, i)(\lceil A \rceil \parallel i[e \triangleleft q]^*) \xrightarrow{j?v} (\nu\, i)(\lceil B \rceil \parallel i[e \triangleleft q : \{\mathsf{rv}, j, v\}]^*) & \textit{if } \alpha = j?v \\ (\nu\, i)(\lceil A \rceil \parallel i[e \triangleleft q]^*) \xrightarrow{\tau} (\nu\, i)(\lceil B \rceil \parallel i[e \triangleleft q]^*) & \textit{if } \alpha = \tau \end{cases}$$

We are now in a position to state monitor correctness, for some predefined violation-detection monitor action fail!, Definition 6; in what follows, fail is always assumed to be fresh. We restrict our definition to expressions e located at a fresh scoped location i (not used by the system, *i.e.*, $i \notin \text{fId}(A)$) with an empty mailbox, ϵ; expression e may then spawn concurrent submonitors while executing. The definition can be extended to generic concurrent monitors, *i.e.*, multiple expressions, in straightforward fashion.

Definition 6 (Correctness) $e \in \text{Exp}$ is a correct monitor for $\varphi \in \text{sHML}$ iff for any basic actors $A \in \text{Actr}$, $i \notin \text{fId}(A)$, and execution traces $s \in (\text{Act} \setminus \{\mathsf{fail!}\})^*$:

$$(\nu\, i)(\lceil A \rceil \parallel i[e]^*) \overset{s}{\Longrightarrow} B \quad \text{implies} \quad (A, s \models_{\text{v}} \varphi \quad \text{iff} \quad B \Downarrow_{\mathsf{fail!}})$$

Definition 6 states that e correctly monitors property φ whenever, for any trace of environment interactions, s, of a monitored system, $(\nu\, i)(\lceil A \rceil \parallel i[e \triangleleft \epsilon]^*)$, yielding system B, if s leads A to a violation of φ, then system B should always detect it, and viceversa. It formalises the definition of monitor correctness outlined earlier in (1) from Sec. 1.

[4] Due to asynchronous communication, even scoped actors can produce visible actions by sending messages to environment actors.

5 Automated Monitor Synthesis

We define a translation from sHML formulas to Erlang *monitors* that asynchronously analyse a system and flag an alert whenever they detect violations by the current system execution (for the respective sHML formula). This translation describes the core algorithm for a tool automating monitor synthesis from sHML formulas [21].

Despite its relative simplicity, sHML still provides opportunities for performing concurrent monitoring. The most obvious case is the translation of conjunction formulas, $\varphi_1 \wedge \varphi_2$, whereby the resulting code needs to check *both* sub-formulas φ_1 and φ_2 so as to ensure that *neither* is violated; since conjunctions are prevalent in many monitoring logics, we conjecture that the concepts discussed here extend in straightforward fashion to similar runtime verification settings with asynchronous concurrent monitors. More specifically, a translation in terms of two *concurrent* (sub)monitors, each analysing different parts of the trace so as to ensure the observation of its respective sub-formula, constitutes a natural synthesis of the conjunction formula in our target language: it adheres to recommended Erlang practices advocating for concurrency wherever possible [11], but also allows us to benefit from the advantages of concurrent monitors discussed in Sec. 1.

Example 5 (Conjunction Formulas) Consider the two sHML formulas

$$\varphi_{\text{no\_dup\_ans}} \triangleq [\alpha_{\text{call}}]\,(\,\mathsf{max}(X,\ [\beta_{\text{ans}}]\,[\beta_{\text{ans}}]\,\mathsf{ff} \wedge [\beta_{\text{ans}}]\,[\alpha_{\text{call}}]\,X)\,)$$
$$\varphi_{\text{react\_ans}} \triangleq \mathsf{max}(Y,\ [\beta_{\text{ans}}]\,\mathsf{ff} \wedge [\alpha_{\text{call}}]\,[\beta_{\text{ans}}]\,Y\,)$$

Formula $\varphi_{\text{no\_dup\_ans}}$ requires that call actions α_{call} are at most serviced by a *single* answer action β_{ans}, and that this condition is invariant for any sequence of $(\alpha_{\text{call}},\beta_{\text{ans}})$ pairs. On the other hand, formula $\varphi_{\text{react\_ans}}$ requires that answer actions are only produced *in response to* call actions; again this is required to be invariant for sequences of $(\alpha_{\text{call}},\beta_{\text{ans}})$ pairs. Although one can rephrase the conjunction of the two formulas as a formula without a top-level conjunction, it is more straightforward to use two concurrent monitors executing in parallel (one for each sub-formula in $\varphi_{\text{no\_dup\_ans}} \wedge \varphi_{\text{react\_ans}}$). There are also other reasons why it would be beneficial to keep the sub-formulas separate: for instance, keeping the formulas disentangled improves maintainability and separation of concerns when subformulas originate from distinct requirement specifications.[5] ■

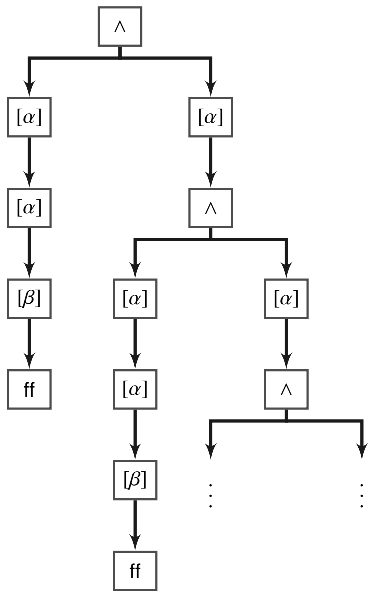
Multiple conjunctions also arise indirectly when used under fix-point operators. When synthesising concurrent monitors analysing separate branches of such recursive properties, it is important to generate monitors that can dynamic spawn further submonitors themselves as required at runtime, so as to keep the monitoring overheads to a minimum.

Example 6 (Conjunctions and Fixpoints) Recall φ_{safe}, from (8) in Ex. 3.

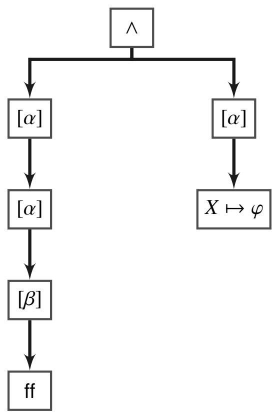
$$\varphi_{\text{safe}} \triangleq \mathsf{max}(X,\ [\alpha][\alpha][\beta]\mathsf{ff} \wedge [\alpha]X\,)$$

Semantically, the formula represents the infinite-depth tree with an infinite number of conjunctions, as depicted in Fig. 3(a). Although in practice, we cannot generate an infinite number of concurrent monitors, φ_{safe} will translate into possibly more than two concurrent monitors executing in parallel. The same applies for the conjunctions used in formulas $\varphi_{\text{no\_dup\_ans}}$ and $\varphi_{\text{react\_ans}}$ from Example 5. ■

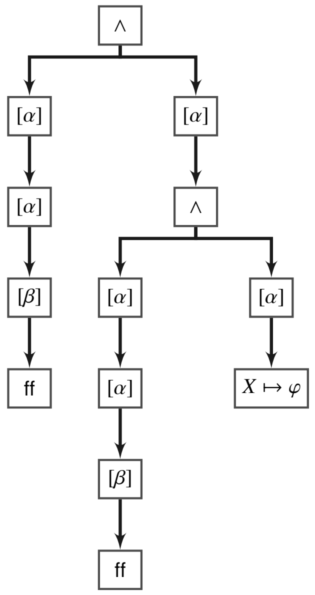
[5] One potential disadvantage of splitting formulas is that of increasing communication amongst monitors.



(a) Denotation of φ_{safe} defined in (8)



(b) Constructed concurrent monitors for φ_{safe} where $\varphi = [\alpha][\alpha][\beta]\mathsf{ff}\wedge[\alpha]X$



(c) First expansion of the constructed monitor for φ_{safe}

Fig. 3: Monitor Combinator generation for φ_{safe} of Ex. 3

Our monitor synthesis, $[\![-]\!]^{\mathbf{m}} :: \text{sHML} \rightarrow \textsc{Exp}$, given in Definition 7, takes a *closed, guarded*[6] sHML formula and returns an Erlang function. This function takes a *map* as an argument (encoded as a list of tuples from formula variables to other synthesised monitors of the same form) and releases an expression that performs the monitoring.

The map encodes an *environment* that maps formula variables to logical formula, introduced by the binding in the construct $\mathsf{max}(X,\varphi)$; it is used for *lazy* recursive unrolling of formulas so as to minimize monitoring overhead. For instance, when synthesising formula φ_{safe} from Ex. 3, the algorithm initially spawns only two concurrent submonitors, one checking for the subformula $[\alpha][\alpha][\beta]\mathsf{ff}$, and another one checking for the formula $[\alpha]X$, as is depicted in Fig. 3(b). Whenever the rightmost submonitor in Fig. 3(b) observes the action α and reaches X, it consults its environment and retrieves the respective formula bound to X; this allows it to unfolds X and spawns an additonal submonitor as depicted in Fig. 3(c), thereby increasing monitor overheads *incrementally* as needed.

As discussed earlier in Sec. 2, traces are encoded as messages ordered inside the respective mailbox of the monitor. Monitoring thus involves reading these messages from the mailbox *in order*, analysing them, and determining what action to take. In a system of hierarchically-organised concurrent monitors, monitor actions can either flag a violation, forward messages to other sub-monitors, or spawn new monitors.

Definition 7 (Synthesis) $[\![-]\!]^{\mathbf{m}}$ is defined on the structure of the sHML formula:

$$
\begin{aligned}
[\![\mathsf{ff}]\!]^{\mathbf{m}} &\overset{\text{def}}{=} \lambda x_{\text{env}}.\mathsf{fail}!\\
[\![\varphi_1\wedge\varphi_2]\!]^{\mathbf{m}} &\overset{\text{def}}{=} \begin{cases}\lambda x_{\text{env}}.\, y_{\text{pid1}} = \mathsf{spw}\,([\![\varphi_1]\!]^{\mathbf{m}}(x_{\text{env}})),\\ \qquad\quad y_{\text{pid2}} = \mathsf{spw}\,([\![\varphi_2]\!]^{\mathbf{m}}(x_{\text{env}})),\\ \qquad\quad \mathsf{fork}(y_{\text{pid1}}, y_{\text{pid2}})\end{cases}\\
[\![[\alpha]\varphi]\!]^{\mathbf{m}} &\overset{\text{def}}{=} \begin{cases}\lambda x_{\text{env}}.\mathsf{rcv}\\ \qquad\qquad \mathsf{tr}(\alpha) \rightarrow [\![\varphi]\!]^{\mathbf{m}}(x_{\text{env}});\\ \qquad\qquad \_ \rightarrow \mathsf{stop}\\ \qquad\quad \mathsf{end}\end{cases}\\
[\![\mathsf{max}(X,\varphi)]\!]^{\mathbf{m}} &\overset{\text{def}}{=} \lambda x_{\text{env}}.\, y_{\text{mon}} = [\![\varphi]\!]^{\mathbf{m}}, y_{\text{mon}}(\{'X', y_{\text{mon}}\} : x_{\text{env}})\\
[\![X]\!]^{\mathbf{m}} &\overset{\text{def}}{=} \lambda x_{\text{env}}.\, y_{\text{mon}} = \mathsf{lookUp}('X', x_{\text{env}}),\ y_{\text{mon}}(x_{\text{env}})
\end{aligned}
$$

Auxiliary Function definitions and meta-operators:

$$
\begin{aligned}
\mathsf{fork} &\overset{\text{def}}{=} \mu y_{\text{rec}}.\lambda(x_{\text{pid1}}, x_{\text{pid2}}).\mathsf{rcv}\, z \rightarrow (x_{\text{pid1}}!z,\ x_{\text{pid2}}!z)\,\mathsf{end},\ y_{\text{rec}}(x_{\text{pid1}}, x_{\text{pid2}})\\
\mathsf{lookUp} &\overset{\text{def}}{=} \begin{cases}\mu y_{\text{rec}}.\lambda(x_{\text{var}}, x_{\text{map}}).\mathsf{case}\ x_{\text{map}}\ \mathsf{of}\ (\{x_{\text{var}}, z_{\text{mon}}\} : \_) \rightarrow z_{\text{mon}}\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \_ : z_{\text{tl}} \rightarrow y_{\text{rec}}(x_{\text{var}}, z_{\text{tl}})\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{nil} \rightarrow \mathsf{exit}\\ \qquad\qquad\qquad\qquad \mathsf{end}\end{cases}
\end{aligned}
$$

In Definition 7, the synthesised monitor for formula ff immediately reports a violation to some supervisor actor handling the violation; we assume that the supervisor actor is identified by the name fail. Conjunction formulas, $\varphi_1\wedge\varphi_2$, translate into the spawning of the

[6] In *guarded* sHML formulas, variables appear only as a sub-formula of a necessity formula.

respective monitors for φ_1 and φ_2, *i.e.*, the command $y_{\text{pid}i} = \mathsf{spw}\,([\![\varphi_i]\!]^{\mathbf{m}}(x_{\text{env}}))$ in Definition 7, and the subsequent forwarding of trace messages to these spawned monitors through the auxiliary function fork. The translated monitor for $[\alpha]\varphi$ behaves as the monitor translation for φ once it receives a trace message encoding the occurrence of action α, *i.e.*, the guarded expression $\text{tr}(\alpha) \rightarrow [\![\varphi]\!]^{\mathbf{m}}(x_{\text{env}})$ in Definition 7, which uses the meta-function tr(−) defined below:

$$\text{tr}(i?v) \stackrel{\text{def}}{=} \{\mathsf{rv}, i, v\} \qquad\qquad \text{tr}(i!v) \stackrel{\text{def}}{=} \{\mathsf{sd}, i, v\}$$

Importantly, the monitor for $[\alpha]\varphi$ terminates if the trace message does not correspond to α, *i.e.*, the guarded expression $\_ \rightarrow \mathsf{stop}$ in Definition 7.

The translations for formulas $\mathsf{max}(X, \varphi)$ and X are best understood together. The monitor for $\mathsf{max}(X, \varphi)$ behaves like that for φ, under the *extended* environment where X is mapped to the monitor for φ,*i.e.*, the operation $\{'X', y_{\text{mon}}\} : x_{\text{env}}$ in Definition 7; this effectively models the formula unrolling $\varphi\{^{\mathsf{max}(X,\varphi)}/_X\}$ from Definition 3. The monitor for X retrieves the respective monitor translation bound to X in the map using the auxiliary function lookUp, and then behaves like the monitor retrieved from the environment; the retrieved monitor is launched by applying it to the environment itself, *i.e.*, expression $y_{\text{mon}}(x_{\text{env}})$ in Definition 7. *Closed* formulas ensure that map entries for the formula variables used are always present in the respective environment generated, whereas *guarded* formulas guarantee that formula variables, X, are guarded by necessity conditions, $[\alpha]\varphi$; this implements the *lazy* recursive unrolling of formulas and prevents infinite bound-variable expansions.

Monitor instrumentation, performed through the function Mon (defined below), spawns the synthesised function initialised to the empty environment, *i.e.*, a nil list, and then acts as a message forwarder to the spawned process, through the function mLoop (defined below), for any trace messages it receives through the tracing semantics discussed in Sec. 2.

$$\begin{aligned}\mathsf{Mon} &\stackrel{\text{def}}{=} \lambda x_{\text{frm}}.\ z_{\text{pid}} = \mathsf{spw}\,([\![x_{\text{frm}}]\!]^{\mathbf{m}}(\mathsf{nil})),\ \mathsf{mLoop}(z_{\text{pid}})\\ \mathsf{mLoop} &\stackrel{\text{def}}{=} \mu y_{\text{rec}}.\lambda x_{\text{pid}}.\ \mathsf{rcv}\ z_{\text{msg}} \rightarrow (x_{\text{pid}}!z_{\text{msg}})\ \mathsf{end},\ y_{\text{rec}}(x_{\text{pid}})\end{aligned}$$

Example 7 (Synthesised monitor) Recall $\varphi_{\text{react\_ans}}$ from Example 5:

$$\varphi_{\text{react\_ans}} \triangleq \mathsf{max}(Y,\ [\beta_{\text{ans}}]\,\mathsf{ff} \wedge [\alpha_{\text{call}}]\,[\beta_{\text{ans}}]\,Y)$$

According to Definition 7, its respective monitor translation is the one described below.

Once the translated function above is applied to the function Mon (defined above), it is spawned as a concurrent actor where variable x_{env} (below) is instantiated to the empty environment, nil. In turn, the spawned function launches two further concurrent actors that monitor for the subformulas $[\beta_{\text{ans}}]\,\mathsf{ff}$ and $[\alpha_{\text{call}}]\,[\beta_{\text{ans}}]\,Y$, binding their respective pIds to variables y_{pid1} and y_{pid2} below. Note that these two actors are instantiated with the extended environment $\{'Y', e_{\text{unf}}\} : \mathsf{nil}$, mapping the formula variable Y to the unfolding function e_{unf} (labelled below). This function is retrieved and executed by the actor monitoring for the subformula $[\alpha_{\text{call}}]\,[\beta_{\text{ans}}]\,Y$, after reading two consecutive messages from its mailbox of the form α_{call} and β_{ans}. In this setup, the residual expressions of Mon and $[\![\varphi_{\text{react\_ans}}]\!]^{\mathbf{m}}$ (after their respective actor spawnings) behave as trace forwarders to the two concurrent actors monitoring for subformula $[\beta_{\text{ans}}]\,\mathsf{ff}$ and subformula $[\alpha_{\text{call}}]\,[\beta_{\text{ans}}]\,Y$.

$$
\begin{array}{l}
[\![\varphi_{\text{react\_ans}}]\!]^{\mathbf{m}} = [\![\mathsf{max}(Y,\ [\beta_{\text{ans}}]\,\mathsf{ff} \wedge [\alpha_{\text{call}}]\,[\beta_{\text{ans}}]\,Y)]\!] = \\
\lambda x_{\text{env}}. \\
\quad y_{\text{mon}} = \\
\quad \left.\left(\begin{array}{l}
\lambda x'_{\text{env}}. \\
\quad y_{\text{pid1}} = \mathsf{spw}\left(\left(\begin{array}{l}
\lambda x''_{\text{env}}. \\
\quad \mathsf{rcv}\ \mathsf{tr}(\beta_{\text{ans}}) \rightarrow \lambda x'''_{\text{env}}.\mathsf{fail}!(x''_{\text{env}}); \\
\qquad \_ \rightarrow \mathsf{stop} \\
\quad \mathsf{end}
\end{array}\right)(x'_{\text{env}})\right), \\
\quad y_{\text{pid2}} = \mathsf{spw}\left(\left(\begin{array}{l}
\lambda z_{\text{env}}. \\
\quad \mathsf{rcv}\ \mathsf{tr}(\alpha_{\text{call}}) \rightarrow \\
\qquad \left(\begin{array}{l}
\lambda z'_{\text{env}}. \\
\quad \mathsf{rcv}\ \mathsf{tr}(\beta_{\text{ans}}) \rightarrow \\
\qquad \lambda z''_{\text{env}}.\left(\begin{array}{l} y_{\text{mon}} = \mathsf{lookUp}('Y', z''_{\text{env}}), \\ y_{\text{mon}}(z''_{\text{env}}) \end{array}\right)(z'_{\text{env}}) \\
\qquad \_ \rightarrow \mathsf{stop} \\
\quad \mathsf{end}
\end{array}\right)(z_{\text{env}})\,; \\
\qquad \_ \rightarrow \mathsf{stop} \\
\quad \mathsf{end}
\end{array}\right)(x'_{\text{env}})\right), \\
\quad \mathsf{fork}(y_{\text{pid1}}, y_{\text{pid2}})
\end{array}\right)\right\} e_{\text{unf}} \\
y_{\text{mon}}(\{'Y', y_{\text{mon}}\} : x_{\text{env}}) \qquad \blacksquare
\end{array}
$$

5.1 Tool Implementation

We have constructed a tool called detectEr [21], that implements the monitor synthesis of Definition 7: given an sHML formula it generates a monitor that can be instrumented with minimal changes to the system (actual Erlang code), as discussed earlier in Sec. 4. Despite not being the main focus of this work, we conducted preliminary empirical experiments evaluating the performance of our synthesised monitors. This was carried out using a simulated server that launches individual workers to handle a series of requests from individual clients; we also injected faults making certain workers non-deterministically behave erratically. We synthesised monitors to check that each worker respects the *no-duplicate-reply* property from Example 5:

$$\varphi_{wrkr} \triangleq [wrk?req]\,(\,\mathsf{max}(X,\ [clnt!rply]\,[clnt!rply]\,\mathsf{ff} \wedge [clnt!rply]\,[wrk?req]\,X)\,)$$

and calculated the overheads incurred for varying number of client requests (*i.e.,* concurrent workers); we also compared this with the performance a monitor that checks for property violations in sequential fashion. Tests were carried out on an Intel Core i7 processor with 8GB of RAM, running Microsoft Windows 8 and EVM version R15B02. The results, summarised in the table below, show that our synthesised concurrent monitoring yields acceptable overheads that are consistently lower than those of a sequential monitor. We conjecture that this discrepancy can be increased further when monitoring for recursive properties with longer chains of necessity formulas. For more extensive empirical results relating to the tool detectEr, see [9].

	Unmonitored	Sequential		Concurrent		
No of. Reqs.	**Time** (μs)	**Time** (μs)	**Ovhd**(%)	**Time**(μs)	**Ovhd.**(%)	**Improv.**(%)
250	117.813	121.667	3.27	118.293	0.40	2.86
350	185.232	202.500	9.32	194.793	5.16	4.16
450	237.606	248.333	4.51	242.380	2.01	2.51
550	286.461	319.167	11.42	308.853	7.82	3.60
650	345.543	372.232	7.72	354.333	2.54	5.18

6 Proving Correctness

The preliminary results obtained in Sec. 5 advocate for the feasibility of using concurrent monitors. We however still need to show that the monitors synthesised are correct. Definition 6 allows us to state one of the main results of the paper, Theorem 3.

Theorem 3 (Correctness) *For all $\varphi \in$ sHML, $\mathsf{Mon}(\varphi)$ is a correct monitor for φ.*

Proving Theorem 3 directly can be an arduous task: for *any* sHML formula, it requires reasoning about *all* the possible execution paths of *any* monitored system in parallel with the instrumented monitor. We propose a formal technique for alleviating the task of ascertaining the monitor correctness of Definition 6 by teasing apart *three* separate (weaker) monitor-conditions: they are referred to as *Violation Detectability*, *Detection Preservation* and *Monitor Separability*.

These conditions are important properties in their own right—for instance, Detection Preservation requires the monitor to behave *deterministically wrt.* violation detections, whereas Monitor Separability requires that the monitor computation *does not affect* the execution of the monitored system. Moreover, the three conditions pose advantages to the checking of monitor correctness: since these conditions are independent to one another, they can be checked in parallel by distinct analysing entities; alternatively, the conditions that are inexpensive to check may be carried out before the more expensive ones, thus acting as vetting phases that abort early and keep the analysis cost to a minimum. More importantly though, the three conditions together imply our original monitor-correctness criteria.

The first sub-property is *Violation Detectability*, Lemma 3, guaranteeing that every violating trace s of formula φ is detectable by the respective synthesised monitor,[7] (the *only-if* case) and that there are no false detections (the *if* case). This property is easier to verify than Theorem 3 since it requires us to consider the execution of the monitor in isolation and, more importantly, requires us to verify the *existence of an execution path* that detects the violation; concurrent monitors typically have multiple execution paths and Theorem 3 requires us to prove this property for *all* of the possible monitor execution paths.

Lemma 3 (Violation Detectability) *For basic $A \in$ ACTR and $i \notin \mathrm{fId}(A)$, $A \overset{s}{\Longrightarrow}$ implies:*

$$A, s \models_v \varphi \quad \textit{iff} \quad i[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(s)]^* \xRightarrow{\mathsf{fail!}}$$

Detection Preservation (Lemma 4), the second sub-property, is not concerned with relating detections to the violations specified by our logic semantics, $A, s \models_v \varphi$. Instead it guarantees that if a monitor can potentially detect a violation, further reductions do not exclude the possibility of this detection. In the case where monitors always have a finite

[7] We elevate tr to basic action sequences s in pointwise fashion, $\mathrm{tr}(s)$, where $\mathrm{tr}(\epsilon) = \epsilon$.

reduction *wrt.* their mailbox contents (as it turns out to be the case for monitors synthesised by Definition 7) this condition guarantees that the monitor will detect violations *deterministically*. More generally, however, in a setting that guarantees fair actor executions, Lemma 4 ensures that detection will always eventually occur, even when monitors execute in parallel with other, potentially divergent, systems.

Lemma 4 (Detection Preservation) *For all* $\varphi \in$ sHML, $q \in \textsc{Val}^*$

$$\Big(i[\mathit{Mon}(\varphi) \triangleleft q]^* \overset{\mathit{fail!}}{\Longrightarrow} \;\; \textit{and} \;\; i[\mathit{Mon}(\varphi) \triangleleft q]^* \Longrightarrow B\Big) \quad \textit{implies} \quad B \overset{\mathit{fail!}}{\Longrightarrow}$$

The third sub-property is Separability, Lemma 5, which implies that the behaviour of a (monitored) system *is independent of* the monitor and, dually, the behaviour of the monitor depends, *at most*, on the trace generated by the system.

Lemma 5 (Monitor Separability) *For all basic* $A \in \textsc{Actr}, i \notin \mathrm{fId}(A), \varphi \in$ sHML, *and* $s \in (\textsc{Act} \setminus \{\mathit{fail!}\})^*$,

$$(\nu\, i)(\lceil A \rceil \parallel i[\mathit{Mon}(\varphi)]^*) \overset{s}{\Longrightarrow} B \textit{ implies } \exists B', B'' \textit{s.t.}$$
$$B \equiv (\nu\, i)(B' \parallel B'') \quad \textit{and} \quad A \overset{s}{\Rightarrow} A' \textit{ s.t. } B' = \lceil A' \rceil \quad \textit{and} \quad i[\mathit{Mon}(\varphi) \triangleleft \mathrm{tr}(s)]^* \Longrightarrow B''$$

These three properties suffice to show monitor correctness.

Theorem 3 (Correctness). *For all* $\varphi \in$ sHML, $\mathit{Mon}(\varphi)$ *is a correct monitor for* φ.

Proof According to Definition 6 we have to show:

$$(\nu\, i)(\lceil A \rceil \parallel i[\mathsf{Mon}(\varphi)]^*) \overset{s}{\Longrightarrow} B \quad \text{implies} \quad (A, s \models_{\mathrm{V}} \varphi \quad \text{iff} \quad B \Downarrow_{\mathsf{fail!}})$$

We assume that $(\nu\, i)(\lceil A \rceil \parallel i[\mathsf{Mon}(\varphi)]^*) \overset{s}{\Longrightarrow} B$ holds and consider the two sides of the bi-implication separately. For the *only-if* case, we assume

$$(\nu\, i)(A \parallel i[\mathsf{Mon}(\varphi)]^*) \overset{s}{\Longrightarrow} B \tag{9}$$
$$A, s \models_{\mathrm{V}} \varphi \tag{10}$$

In order to show $B \Downarrow_{\mathsf{fail!}}$, we use Definition 4 *(Should-α)* to expand $B \Downarrow_{\mathsf{fail!}}$. We thus also assume $B \Longrightarrow B'$, for arbitrary B', and then be required to prove that $B' \overset{\mathsf{fail!}}{\Longrightarrow}$. From (9), $B \Longrightarrow B'$ and Lemma 5 *(Monitor Separability)* we know

$$\exists B'', B''' \textit{s.t. } B' \equiv (\nu\, i)(B'' \parallel B''') \tag{11}$$
$$A \overset{s}{\Longrightarrow} A' \text{ for some } A' \text{ where } \lceil A' \rceil = B'' \tag{12}$$
$$i[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(s)]^* \Longrightarrow B''' \tag{13}$$

From (12), (10) and Lemma 3 *(Violation Detectability)* we obtain

$$i[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(s)]^* \overset{\mathsf{fail!}}{\Longrightarrow} \tag{14}$$

and from (13) , (14) and Lemma 4 *(Detection Preservation)* we get $B''' \overset{\mathsf{fail!}}{\Longrightarrow}$. Hence, by (11), and standard transition rules for parallel composition and scoping, Par and Scp, we can reconstruct $B' \overset{\mathsf{fail!}}{\Longrightarrow}$, as required.

For the *if* case we assume:

$$(\nu\, i)(\lceil A \rceil \parallel i[\mathsf{Mon}(\varphi)]^{*}) \overset{s}{\Longrightarrow} B \tag{15}$$

$$B \Downarrow_{\mathsf{fail!}} \tag{16}$$

and have to prove $A, s \models_{\mathrm{v}} \varphi$. From (16) we know $B \overset{\mathsf{fail!}}{\Longrightarrow}$. Together with (15), this implies that there exists a sequence of reductions (τ-transitions) from $(\nu\, i)(\lceil A \rceil \parallel i[\mathsf{Mon}(\varphi)]^{*})$ leading to a (monitored) system that flags a violation:

$$\exists B' \; s.t. \; (\nu\, i)(\lceil A \rceil \parallel i[\mathsf{Mon}(\varphi)]^{*}) \overset{s}{\Longrightarrow} B' \xrightarrow{\mathsf{fail!}} \tag{17}$$

From Lemma 5 *(Monitor Separability)* and (17) we obtain

$$\exists B'', B''' s.t. \; B' = (\nu\, i)(B'' \parallel B''') \tag{18}$$

$$A \overset{s}{\Longrightarrow} A' \text{ for some } A' \text{ where } \lceil A' \rceil = B'' \tag{19}$$

$$i[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(s)]^{*} \Longrightarrow B''' \tag{20}$$

From (17), (18) and the assumption that action $\mathsf{fail!}$ is fresh to A, we deduce that it can only be B''' that is capable of flagging the violation:

$$B''' \xrightarrow{\mathsf{fail!}} . \tag{21}$$

Thus, by (20) and (21), we get

$$i[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(s)] \overset{\mathsf{fail!}}{\Longrightarrow} \tag{22}$$

Therefore, by (19) , (22) and Lemma 3 *(Violation Detectability)* we obtain $A, s \models_{\mathrm{v}} \varphi$. ■

7 Proving the Monitor Sub-properties

The proof of monitor correctness given in Sec. 6 hinges on the three monitor sub-properties discussed in the same section. We now consider the proofs for these three monitor properties for the monitor synthesis presented in Sec. 5. In what follows, we assume that the tracer, *i.e.*, where the synthesised monitor is placed in the instrumentation of Definition 6, is located at the process identifier i_{mtr}. Moreover sequences of identifiers are denoted as $\tilde{h}$; for instance, $(\nu\, \tilde{j})A$ is used to denote the system $(\nu\, j_1)\dots(\nu\, j_n)A$ whenever the scoped identifiers are unimportant.

These proofs rely on an encoding of formula substitutions, $\theta :: \textsc{LVar} \rightharpoonup \mathrm{sHML}$, partial maps from formula variables to (possibly open) formulas, to lists of tuples containing a string representation of the variable and the respective monitor translation of the formula as defined in Definition 7. Formula substitutions are denoted as lists of individual substitutions, $\{^{\varphi_1}/_{X_1}\}\dots\{^{\varphi_n}/_{X_n}\}$ where every X_i is distinct, and empty substitutions are denoted as ϵ.

Definition 8 (Formula Substitution Encoding)

$$\mathrm{enc}(\theta) \overset{\mathbf{def}}{=} \begin{cases} \mathsf{nil} & \text{when } \theta = \epsilon \\ \{'X', [\![\varphi]\!]^{\mathbf{m}}\} : \mathrm{enc}(\theta') & \text{if } \theta = \{^{\mathsf{max}(X,\varphi)}/_{X}\}\theta' \end{cases}$$

Our monitor lookup function of Definition 7 models variable substitution, Lemma 6. We can also show that different representations of the same formula substitution do not affect the outcome of the execution of lookUp on the respective encoding, Lemma 7, which justifies the abuse of notation that assume a unique representation of a formula substitution.

Lemma 6 *If* $\theta(X) = \varphi$ *then* $i[\mathsf{lookUp}('X', \mathrm{enc}(\theta)) \triangleleft q]^m \Longrightarrow i[[\![\varphi]\!]^{\mathbf{m}} \triangleleft q]^m$

Proof By induction on the number of mappings $\{\varphi_1/X_1\}\ldots\{\varphi_n/X_n\}$ in θ. The base case, *i.e.*, when we have zero mappings, is trivial since it contradicts the left side of the implication. The inductive case has two subcases and follows from the pattern-matching branches of the lookUp code, defined in Definition 7. ■

Lemma 7 *If* $\theta(X) = \varphi$ *then* $i[\mathsf{lookUp}('X', \mathrm{enc}(\theta')) \triangleleft q]^m \Longrightarrow i[[\![\varphi]\!]^{\mathbf{m}} \triangleleft q]^m$ *whenever* θ *and* θ' *denote the same substitution.*

Proof By induction on the number of mappings $\{\varphi_1/X_1\}\ldots\{\varphi_n/X_n\}$ in θ, analogously to the proof for Lemma 6. ■

Our proofs use another technical result, Lemma 8, stating that silent actions are, in some sense, preserved when actor-mailbox contents of a free actor are increased; note that the lemma only applies for cases where the mailbox at this free actor decreases in size or remains unaffected by the τ-action, specified through the sublist condition $q' \leq q$.

Lemma 8 (Mailbox Increase) $(\nu\,\tilde{h})(i[e \triangleleft q]^m \parallel A) \xrightarrow{\tau} (\nu\,\tilde{j})(i[e' \triangleleft q']^m \parallel B)$ *where* $i \notin \tilde{h}$ *and* $q' \leq q$ *implies* $(\nu\,\tilde{h})(i[e \triangleleft q : v]^m \parallel A) \xrightarrow{\tau} (\nu\,\tilde{j})(i[e' \triangleleft q' : v]^m \parallel B)$

Proof By rule induction on $(\nu\,\tilde{h})(i[e \triangleleft q]^m \parallel A) \xrightarrow{\tau} (\nu\,\tilde{j})(i[e' \triangleleft q']^m \parallel B)$. ■

7.1 Violation Detection

In one direction, Lemma 3 *(Violation Detection)* from Sec. 6 relies on Lemma 13 (see Appendix A.1) in order to establish the correspondence between violations and the possibility of detections. In the other direction, Lemma 3 relies on Lemma 16 (see Appendix A.1), which establishes a correspondence between violation detections and actual violations, as stated in Definition 3. We recall that Lemma 3 was stated *wrt. closed* sHML formulas.

Lemma 3 (Violation Detection). *Whenever* $A \overset{s}{\Longrightarrow}$ *then :*

$$A, s \models_v \varphi \quad \textit{iff} \quad i_{mtr}[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(s)]^* \overset{\mathsf{fail!}}{\Longrightarrow}$$

Proof For the *only-if* case, we assume $A \overset{s}{\Longrightarrow}$ and $A, s \models_v \varphi$ and are required to prove $i_{\mathrm{mtr}}[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(s)]^* \overset{\mathsf{fail!}}{\Longrightarrow}$. We recall from Sec. 5 that Mon was defined as

$$\lambda x_{\mathrm{frm}}.z_{\mathrm{pid}} = \mathsf{spw}\,([\![x_{\mathrm{frm}}]\!]^{\mathbf{m}}(\mathsf{nil})), \mathsf{mLoop}(z_{\mathrm{pid}}). \tag{23}$$

and as a result we can deduce (using rules such as App, Spw and Par) that

$$i_{\mathrm{mtr}}[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(s)]^* \Longrightarrow (\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^* \parallel i[[\![\varphi]\!]^{\mathbf{m}}(\mathsf{nil})]^{\bullet}) \tag{24}$$

Assumption $A, s \models_{\mathrm{v}} \varphi$ can be rewritten as $A, s \models_{\mathrm{v}} \varphi\theta$ for $\theta = \epsilon$, and thus, by Definition 8 we know $\mathsf{nil} = \mathrm{enc}(\theta)$. By Lemma 13 we obtain

$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[\llbracket\varphi\rrbracket^{\mathbf{m}}(\mathsf{nil})]^{\bullet}) \overset{\mathsf{fail!}}{\Longrightarrow} \tag{25}$$

and the result thus follows from (24) and (25).

For the *if* case, we assume $A \overset{s}{\Longrightarrow}$ and $i_{\mathrm{mtr}}[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(s)]^{*} \overset{\mathsf{fail!}}{\Longrightarrow}$ and are required to prove $A, s \models_{\mathrm{v}} \varphi$.

Since φ is closed, we can assume the empty list of substitutions $\theta = \epsilon$ where, by default, $\mathrm{fv}(\varphi) \subseteq \mathrm{dom}(\theta)$ and, by Definition 8, $\mathsf{nil} = \mathrm{enc}(\theta)$. By (23) we can decompose the transition sequence $i_{\mathrm{mtr}}[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(s)]^{*} \overset{\mathsf{fail!}}{\Longrightarrow}$ as

$$i_{\mathrm{mtr}}[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(s)]^{*}(\xrightarrow{\tau})^{3}(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[\llbracket\varphi\rrbracket^{\mathbf{m}}(\mathsf{nil})]^{\bullet}) \overset{\mathsf{fail!}}{\Longrightarrow} \tag{26}$$

The result, *i.e.*, $A, s \models_{\mathrm{v}} \varphi$, follows from (26) and Lemma 16. ■

7.2 Detection Preservation

In order to prove Lemma 4 from Sec. 6, we are able to require a stronger guarantee, *i.e.*, confluence (Definition 9) under weak transitions for the concurrent monitors described in Definition 7; this is formalised as Lemma 18 in Appendix A.2.

Definition 9 (Confluence modulo Inputs with Identical Recipients)

$$\mathrm{cnf}(A) \overset{\mathbf{def}}{=} \; A \xrightarrow{\gamma_1} A' \text{ and } A \xrightarrow{\gamma_2} A'' \;\text{ implies }\; \begin{cases} \gamma_1 = i?v_1, \gamma_2 = i?v_2 \;\text{ or;} \\ \gamma_1 = \gamma_2, A' = A'' \;\text{ or;} \\ A' \xrightarrow{\gamma_2} A''', A'' \xrightarrow{\gamma_1} A''' \text{ for some } A''' \end{cases}$$

In Definition 9, a system is deemed *confluent* if, whenever it can perform two *separate* actions γ_1 and γ_2 that are *not* input actions at the same actor, both residual systems can *resp.* still perform the other action to reach the same system.

Lemma 18 (Appendix A.2) allows us to prove Lemma 9, and subsequently Lemma 10; the latter Lemma implies Detection Preservation, Lemma 4, used by Theorem 3 of Sec. 6.

Lemma 9 *For all* $\varphi \in \mathrm{sHML}$, $q \in \mathrm{VAL}^{*}$

$$i_{mtr}[\mathit{Mon}(\varphi) \triangleleft q]^{*} \Longrightarrow A, A \overset{\mathit{fail!}}{\Longrightarrow} \;\textit{and}\; A \xrightarrow{\tau} B \;\textit{ implies }\; B \overset{\mathit{fail!}}{\Longrightarrow}$$

Proof From $i_{\mathrm{mtr}}[\mathsf{Mon}(\varphi) \triangleleft q]^{*} \Longrightarrow A$ and Lemma 18 we know that $\mathrm{cnf}(A)$. The proof is by induction on $A(\xrightarrow{\tau})^{n} \cdot \xrightarrow{\mathsf{fail!}}$.

$n = 0$: We have $A \xrightarrow{\mathsf{fail!}} A'$ (for some A'). By $A \xrightarrow{\tau} B$ and $\mathrm{cnf}(A)$ we obtain $B \xrightarrow{\mathsf{fail!}} B'$ for some B' where $A' \xrightarrow{\tau} B'$.

$n = k+1$: We have $A \xrightarrow{\tau} A'(\xrightarrow{\tau})^k \cdot \xrightarrow{\textsf{fail!}}$ (for some A'). By $A \xrightarrow{\tau} A'$, $A \xrightarrow{\tau} B$ and $\text{cnf}(A)$ we either know that $B = A'$, in which case the result follows immediately, or else obtain

$$B \xrightarrow{\tau} A'' \tag{27}$$

$$A' \xrightarrow{\tau} A'' \quad \text{for some } A'' \tag{28}$$

In such a case, by $A \xrightarrow{\tau} A'$ and $i_{\text{mtr}}[\textsf{Mon}(\varphi) \triangleleft q]^* \Longrightarrow A$ we deduce that

$$i_{\text{mtr}}[\textsf{Mon}(\varphi) \triangleleft q]^* \Longrightarrow A',$$

and subsequently, by (28), $A'(\xrightarrow{\tau})^k \cdot \xrightarrow{\textsf{fail!}}$ and I.H. we obtain $A'' \overset{\textsf{fail!}}{\Longrightarrow}$; the required result then follows from (27). ■

Lemma 10 (Detection Confluence) *For all $\varphi \in$ sHML, $q \in \textsc{Val}^*$*

$$i_{mtr}[\textsf{Mon}(\varphi) \triangleleft q]^* \Longrightarrow A, A \overset{\textsf{fail!}}{\Longrightarrow} \text{ and } A \Longrightarrow B \text{ implies } B \overset{\textsf{fail!}}{\Longrightarrow}$$

Proof By induction on $A(\xrightarrow{\tau})^n B$ and Lemma 9. ■

We are now in a position to prove Lemma 4 of Sec. 6.

Lemma 4 (Detection Preservation). *For all $\varphi \in$ sHML, $q \in \textsc{Val}^*$*

$$i_{mtr}[\textsf{Mon}(\varphi) \triangleleft q]^* \overset{\textsf{fail!}}{\Longrightarrow} \text{ and } i_{mtr}[\textsf{Mon}(\varphi) \triangleleft q]^* \Longrightarrow B \text{ implies } B \overset{\textsf{fail!}}{\Longrightarrow}$$

Proof From Lemma 10, for the case where $i_{\text{mtr}}[\textsf{Mon}(\varphi) \triangleleft q]^* \Longrightarrow i_{\text{mtr}}[\textsf{Mon}(\varphi) \triangleleft q]^*$. ■

7.3 Monitor Separability

For the proof for Lemma 5 of Sec. 6, we make use of Lemma 2, relating the behaviour of a monitored system to the same system when unmonitored, Lemma 8 delineating behaviour preservation after extending mailbox contents at specific actors, and Lemma 11(Appendix A), so as to reason about the structure and generic behaviour of synthesised monitors.

Lemma 5 (Monitor Separability). *For all basic actors $\varphi \in$ sHML, $A \in \textsc{Actr}$ where i_{mtr} is fresh to A, and $s \in (\textsc{Act} \setminus \{\textsf{fail!}\})^*$,*

$$(\nu\, i_{mtr})(\lceil A \rceil \parallel i_{mtr}[\textsf{Mon}(\varphi)]^*) \overset{s}{\Rightarrow} B \text{ implies } \exists B', B'' s.t. \begin{cases} B \equiv (\nu\, i_{mtr})(B' \parallel B'') \\ A \overset{s}{\Longrightarrow} A' \ s.t.\ B' = \lceil A' \rceil \\ i_{mtr}[\textsf{Mon}(\varphi) \triangleleft \text{tr}(s)]^* \Longrightarrow B'' \end{cases}$$

Proof By induction on n in $(\nu\, i_{\text{mtr}})(\lceil A \rceil \parallel i_{\text{mtr}}[\textsf{Mon}(\varphi)]^*)(\xrightarrow{\gamma_n})^n B$, the length of the sequence of actions:

$n = 0$: Since $s = \epsilon$ and $A = (\nu\, i_{\text{mtr}})(\lceil A \rceil \parallel i_{\text{mtr}}[\textsf{Mon}(\varphi)]^*)$, the conditions hold trivially.

$n = k+1$: We have $(\nu\, i_{\mathrm{mtr}})(\lceil A\rceil \parallel i_{\mathrm{mtr}}[\mathsf{Mon}(\varphi)]^{*})(\xrightarrow{\gamma_k})^{k} C \xrightarrow{\gamma} B$. By I.H. we know that

$$C \equiv (\nu\, i_{\mathrm{mtr}})(C' \parallel C'') \tag{29}$$

$$A \overset{t}{\Longrightarrow} A'' \text{ s.t. } C' = \lceil A''\rceil \tag{30}$$

$$i_{\mathrm{mtr}}[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(t)]^{*} \Longrightarrow C'' \tag{31}$$

$$\gamma = \tau \text{ implies } t = s \quad\text{and}\quad \gamma = \alpha \text{ implies } t\alpha = s \tag{32}$$

and by (31) and Lemma 11 we know that

$$C'' \equiv (\nu\,\mathbf{h})(i_{\mathrm{mtr}}[e \triangleleft q]^{*} \parallel C''') \tag{33}$$

$$\mathrm{fId}(C'') = \{i_{\mathrm{mtr}}\} \tag{34}$$

We proceed by considering the two possible subcases for the structure of γ:

$\gamma = \alpha$: By (32) we know that $s = t\alpha$. By (34) and (33), it must be the case that $C \equiv (\nu\, i_{\mathrm{mtr}})(C' \parallel C'') \xrightarrow{\alpha} B$ happens because

$$\text{for some } B'\ C' \xrightarrow{\alpha} B' \tag{35}$$

$$B \equiv (\nu\, i_{\mathrm{mtr}})(B' \parallel (\nu\,\mathbf{h})(i_{\mathrm{mtr}}[e \triangleleft q : \mathrm{tr}(\alpha)]^{*} \parallel C''')) \tag{36}$$

By (35), (30) and Lemma 2 we know that $\exists A'$ such that $\lceil A'\rceil = B'$ and that $A'' \xrightarrow{\alpha} A'$. Thus by (30) and $s = t\alpha$ we obtain

$$A \overset{s}{\Longrightarrow} A' \text{ s.t. } B' = \lceil A'\rceil$$

By (31), (33) and repeated applications of Lemma 8 we also know that

$$i_{\mathrm{mtr}}[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(t) : \mathrm{tr}(\alpha)]^{*} = i_{\mathrm{mtr}}[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(s)]^{*} \Longrightarrow \\ (\nu\,\mathbf{h})(i_{\mathrm{mtr}}[e \triangleleft q : \mathrm{tr}(\alpha)]^{*} \parallel C''') = B''$$

The result then follows from (36).

$\gamma = \tau$: Analogous to the other case, where we also have the case that the reduction is instigated by C'', in which case the results follows immediately. ■

8 Conclusion

We have studied a more intensional notion of correctness for monitor synthesis in a concurrent online setting; we worked close to the actual implementation level of abstraction so as to enhance our confidence in the correctness of our instrumented monitors. More precisely, we have identified a number of additional issues raised when proving monitor correctness in this concurrent setting, and built a tool [21], automating monitor synthesis from a reactive property logic (sHML) to asynchronous monitors in a concurrent language (Erlang), illustrating these issues. The specific contributions of the paper, in order of importance, are:

1. A novel formal *definition of monitor correctness*, Definition 6, dealing with issues such as system non-determinism and system interference.
2. A *proof technique* teasing apart aspects of the monitor correctness definition, Lemma 3, Lemma 4 and Lemma 5, allowing us to prove correctness in stages. We subsequently apply this technique to prove the *correctness of our tool*, Theorem 3.

3. An alternative *violation* characterisation of the logic, sHML, that is more amenable to runtime analysis and reasoning about monitor correctness, together with a proof of correspondence for this reformulation, Theorem 1.
4. An extension of a formalisation for Erlang describing its tracing semantics, Sec. 2.
5. A formal monitor synthesis definition from sHML formulas to Erlang code, Definition 7.

Related Work: The aforementioned work, [23,34,5], discusses monitor synthesis from a different logic, namely LTL, to either pseudocode, automata or Büchi automata; none of this work considers online concurrent monitoring, circumventing issues associated with concurrency and system interference. There is considerable work on runtime monitoring of web services, *e.g.*, [18,7] verifying the correctness of reactive (communication) properties, similar to those expressed through sHML; to the best of our knowledge, none of this work tackles correct monitor synthesis from a specified logic. In [14], Colombo *et al.* develop an Erlang RV tool using the EVM tracing mechanism but do not consider the issue of correct monitor generation. Fredlund [22] adapted a variant of HML to specify correctness properties in Erlang, albeit for model checking purposes.

There is also work relating HML formulas with tests, namely [1], but also [10]. Our monitors differ from tests, as in [1], in a number of ways: (*i*) they are defined in terms of *concurrent* actors, as opposed to *sequential* CCS processes; (*ii*) they analyse systems *asynchronously*, acting on traces, whereas tests interact with the system *directly*, forcing certain system behaviour; (*iii*) they are expected to *always* detect violations when they occur whereas tests are only required to have *one* possible execution that detects violations.

In recent work, Bocchi *et al.* [6] studied monitors that enforce multi-party session types at runtime for high-level specifications of message-passing programs, expressed using a distributed π-calculus. Their methodology differs from ours in a number of ways. In particular, (*i*) they give a direct operational semantics to their session specifications in terms of an LTS, which allows them to interact directly with the processes that they monitor; by contrast, we synthesise monitors from our specification formulas as programs in the host language, and focus on proving the correctness of this synthesis; (*ii*) the parallelisation criteria for their session projections is based on the participants being monitored whereas we parallelise on the basis of the structure of the formula, namely across formula conjunctions; (*iii*) they work at the level of an abstract model, namely a distributed π-calculus, whereas we work at a level of abstraction that is close to actual Erlang code that can be compiled and executed.

There is other work on the decomposition of monitor synthesis. In [4], they synthesise a dedicated monitor from an LTL specification for each *synchronous* component executing in a system, thereby *localising* monitoring to a component level. By contrast, our monitor synthesis is agnostic to the internal structure of the monitored system, and decomposition is purely based on the structure of the correctness formulas. In [35], they define a distributed logic for specifying correctness properties of distributed systems and provide a distributed monitor synthesis algorithm for the logic, implemented as actor-based tool called DiAna. Their setting is however different from ours: their systems do not assume a global clock and monitoring work over partially ordered traces (one for each location). Monitoring in the absence of global clocks is also considered in [19], where they develop bisimulation-based coinductive techniques to reason about monitored systems. Crucially, none of these works considers issues relating to the correctness of monitor synthesis studied in this paper.

Future Work: The monitoring semantics of Sec. 2 can be used as a basis to prove the correctness of existing Erlang monitoring tools such as [14,15]. sHML can also be extended to handle limited, monitorable forms of liveness properties (often termed co-safety properties

[28]); the work carried out in [10] provides an ideal starting point. It is also worth exploring mechanisms for synchronous monitoring, as opposed to asynchronous variant studied in this paper. Erlang also facilitates monitor *distribution* which can be used to lower monitoring overheads [35, 16]. Distributed monitoring can also be used to increase the expressivity of our tool so as to handle correctness properties for distributed programs. However, this poses a departure from our setting because the unique trace described by our framework would be replaced by separate independent traces at each location, where the lack of a total ordering of events may prohibit the detection of certain violations [19].

References

1. L. Aceto and A. Ingólfsdóttir. Testing Hennessy-Milner Logic with Recursion. In *FoSSaCS'99*, pages 41–55. Springer, 1999.
2. L. Aceto, A. Ingólfsdóttir, K. G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, New York, NY, USA, 2007.
3. J. Armstrong. *Programming Erlang*. The Pragmatic Bookshelf, 2007.
4. A. Bauer and Y. Falcone. Decentralised LTL monitoring. In D. Giannakopoulou and D. Mry, editors, *FM*, volume 7436 of *LNCS*, pages 85–100. Springer, 2012.
5. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20:14:1–14:64, September 2011.
6. L. Bocchi, T.-C. Chen, R. Demangeon, K. Honda, and N. Yoshida. Monitoring networks through multiparty session types. In *FMOODS/FORTE 2013*, volume 7892 of *LNCS*, pages 50–65, 2013.
7. T.-D. Cao, T.-T. Phan-Quang, P. Felix, and R. Castanet. Automated runtime verification for web services. In *ICWS*, pages 76–82. IEEE, 2010.
8. R. Carlsson. An introduction to Core Erlang. In *PLI'01 (Erlang Workshop)*, 2001.
9. I. Cassar and A. Francalanza. On synchronous and asynchronous monitor instrumentation for actor-based systems. In *FOCLASA*, EPTCS, 2014. (to appear).
10. A. Cerone and M. Hennessy. Process behaviour: Formulae vs. tests. In *EXPRESS'10*, volume 41 of *EPTCS*, pages 31–45, 2010.
11. F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly, 2009.
12. E. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In *ALP*, volume 623 of *LNCS*, pages 474–486. Springer-Verlag, 1992.
13. E. Clarke, Jr., O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.
14. C. Colombo, A. Francalanza, and R. Gatt. Elarva: A monitoring tool for Erlang. In *RV*, volume 7186 of *LNCS*, pages 370–374. Springer, 2011.
15. C. Colombo, A. Francalanza, and I. Grima. Simplifying contract-violating traces. In *FLACOS*, volume 94 of *EPTCS*, pages 11–20, 2012.
16. C. Colombo, A. Francalanza, R. Mizzi, and G. J. Pace. polylarva: Runtime verification with configurable resource-aware monitoring boundaries. In *SEFM*, pages 218–232, 2012.
17. B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. Lola: Runtime monitoring of synchronous systems. In *TIME*. IEEE, 2005.
18. Y. Falcone, M. Jaber, T.-H. Nguyen, M. Bozga, and S. Bensalem. Runtime verification of component-based systems. In *SEFM*, volume 7041 of *LNCS*, pages 204–220. Springer, 2011.
19. A. Francalanza, A. Gauci, and G. J. Pace. Distributed System Contract Monitoring. *JLAP*, 82(5-7):186–215, 2013.
20. A. Francalanza and A. Seychell. Synthesising correct concurrent runtime monitors in Erlang. Technical Report CS2013-01, University of Malta, Jan 2013. Accessible at `https://www.cs.um.edu.mt/svrg/papers.html`.
21. A. Francalanza, A. Seychell, and I. Cassar. DetectEr. Accessible at `https://bitbucket.org/casian/detecter2.0`.
22. L.-Å. Fredlund. *A Framework for Reasoning about Erlang Code*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2001.
23. M. Geilen. On the construction of monitors for temporal logic properties. *ENTCS*, 55(2):181–199, 2001.
24. M. Hennessy. *A Distributed Picalculus*. Cambridge University Proess, Cambridge, UK., 2008.
25. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, Jan. 1985.
26. C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245. Morgan Kaufmann, 1973.

27. D. Kozen. Results on the propositional μ-calculus. *TCS*, 27:333–354, 1983.
28. Z. Manna and A. Pnueli. A hierarchy of temporal properties (invited paper, 1989). In *PODC*, pages 377–410. ACM, 1990.
29. P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Rosu. An overview of the MOP runtime verification framework. *STTT*, 14(3):249–289, 2012.
30. R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
31. R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *TCS*, 114:149–171, 1993.
32. R. D. Nicola and M. C. B. Hennessy. Testing equivalences for processes. *TCS*, pages 83–133, 1984.
33. A. Rensink and W. Vogler. Fair testing. *Inf. Comput.*, 205(2):125–198, Feb. 2007.
34. K. Sen, G. Rosu, and G. Agha. Generating optimal linear temporal logic monitors by coinduction. In *ASIAN*, volume 2896 of *LNCS*, pages 260–275. Springer-Verlag, 2004.
35. K. Sen, A. Vardhan, G. Agha, and G. Roşu. Efficient decentralized monitoring of safety in distributed systems. *ICSE*, pages 418–427, 2004.
36. H. Svensson, L.-Å. Fredlund, and C. Benac Earle. A unified semantics for future erlang. In *Erlang Workshop*, pages 23–32. ACM, 2010.

A Auxiliary Proofs

For the proofs in Sec. 7, we find it convenient to prove a technical result, Lemma 11, identifying the possible structures a monitor can be in after an arbitrary number of silent actions; the lemma also establishes that the only possible external action that a synthesised monitors can perform is the *fail* action: this property helps us to reason about the possible interactions that concurrent monitors may engage in.

Lemma 11 (Monitor Transitions and Structure) *For all* $\varphi \in \text{sHML}, q \in (\text{Val})^*$, $\theta :: \text{LVar} \rightharpoonup \text{sHML}$, *if* $i[\llbracket\varphi\rrbracket^{\mathbf{m}}(\text{enc}(\theta)) \triangleleft q]^{\bullet}(\rightarrow)^{n}A$ *then*

1. $A \xrightarrow{\alpha} B$ *implies* $\alpha = \mathsf{fail}!$ *and;*
2. *A has the form* $i[\llbracket\varphi\rrbracket^{\mathbf{m}}(\text{enc}(\theta)) \triangleleft q]^{\bullet}$ *or, depending on* φ:

$\varphi = \mathsf{ff}$: $A \equiv i[\mathsf{fail!} \triangleleft q]^{\bullet}$ *or* $A \equiv i[\mathsf{fail} \triangleleft q]^{\bullet}$

$\varphi = [\alpha]\psi$: $A \equiv i[\mathsf{rcv}\ (\text{tr}(\alpha) \rightarrow \llbracket\psi\rrbracket^{\mathbf{m}}(\text{enc}(\theta))\,;\, \_ \rightarrow \mathsf{ok})\ \mathsf{end} \triangleleft q]^{\bullet}$ *or*

$(A \equiv B$ *where* $i[\llbracket\psi\rrbracket^{\mathbf{m}}(\text{enc}(\theta)) \triangleleft r]^{\bullet}(\xrightarrow{\tau})^{k}B$ *for some* $k < n$ *and* $q = \text{tr}(\alpha) : r)$ *or*
$A \equiv i[\mathsf{ok} \triangleleft r]^{\bullet}$ *where* $q = u : r$

$$\varphi = \varphi_1 \wedge \varphi_2\text{:}\ A \equiv i\left[\begin{array}{l} y_1 = \mathsf{spw}\,(\llbracket\varphi_1\rrbracket^{\mathbf{m}}(\text{enc}(\theta))),\\ \quad y_2 = \mathsf{spw}\,(\llbracket\varphi_2\rrbracket^{\mathbf{m}}(\text{enc}(\theta))),\, \mathsf{fork}(y_1, y_2) \end{array} \triangleleft q\right]^{\bullet}$$

or

$A \equiv (\nu\, j_1)\Big(\, i[e \triangleleft q]^{\bullet} \parallel (\nu \widetilde{h}_1)(j_1[e_1 \triangleleft q_1]^{\bullet} \parallel B)\,\Big)$ *where*

- *e is* $y_1 = j_1, y_2 = \mathsf{spw}\,(\llbracket\varphi_2\rrbracket^{\mathbf{m}}(\text{enc}(\theta))), \mathsf{fork}(y_1, y_2)$ *or*
  $y_2 = \mathsf{spw}\,(\llbracket\varphi_2\rrbracket^{\mathbf{m}}(\text{enc}(\theta)))\,, \mathsf{fork}(j_1, y_2)$
- $j_1[\llbracket\varphi_1\rrbracket^{\mathbf{m}}(\text{enc}(\theta))]^{\bullet}\ (\xrightarrow{\tau})^{k}\ (\nu \widetilde{h}_1)(j_1[e_1 \triangleleft q_1]^{\bullet} \parallel B)$ *for some* $k < n$

or

$$A \equiv (\nu\, j_1, j_2)\left(\begin{array}{l} i[y_2 = j_2, \mathsf{fork}(j_1, y_2) \triangleleft q]^{\bullet} \\ \parallel\ (\nu \widetilde{h}_1)(j_1[e_1 \triangleleft q_1]^{\bullet} \parallel B)\ \parallel\ (\nu \widetilde{h}_2)(j_2[e_2 \triangleleft q_2]^{\bullet} \parallel C) \end{array}\right)$$

where

- $j_1[\llbracket\varphi_1\rrbracket^{\mathbf{m}}(\text{enc}(\theta))]^{\bullet}\ (\xrightarrow{\tau})^{k}\ (\nu \widetilde{h}_1)(j_1[e_1 \triangleleft q_1]^{\bullet} \parallel B)$ *for some* $k < n$
- $j_2[\llbracket\varphi_2\rrbracket^{\mathbf{m}}(\text{enc}(\theta))]^{\bullet}\ (\xrightarrow{\tau})^{l}\ (\nu \widetilde{h}_2)(j_2[e_2 \triangleleft q_2]^{\bullet} \parallel C)$ *for some* $l < n$

or

$A \equiv (\nu\, j_1, j_2)\Big(\, i[e \triangleleft r]^{\bullet}\ \parallel\ (\nu \widetilde{h}_1)(j_1[e_1 \triangleleft q_1']^{\bullet} \parallel B)\ \parallel\ (\nu \widetilde{h}_2)(j_2[e_2 \triangleleft q_2']^{\bullet} \parallel C)\,\Big)$ *where*

- *e is either* $\mathsf{fork}(j_1, j_2)$ *or* $(\mathsf{rcv}\ z \rightarrow j_1!z, j_2!z\, \mathsf{end}, \mathsf{fork}(j_1, j_2))$
  or $j_1!u, i_2!u, \mathsf{fork}(j_1, j_2)$ *or* $j_2!u, \mathsf{fork}(j_1, j_2)$
- $j_1[\llbracket\varphi_1\rrbracket^{\mathbf{m}}(\text{enc}(\theta)) \triangleleft q_1]^{\bullet}\ (\xrightarrow{\tau})^{k}\ (\nu \widetilde{h}_1)(j_1[e_1 \triangleleft q_1']^{\bullet} \parallel B)$ *for* $k < n$, $q_1 < q$
- $j_2[\llbracket\varphi_2\rrbracket^{\mathbf{m}}(\text{enc}(\theta)) \triangleleft q_2]^{\bullet}\ (\xrightarrow{\tau})^{l}\ (\nu \widetilde{h}_2)(j_2[e_2 \triangleleft q_2']^{\bullet} \parallel C)$ *for* $l < n$, $q_2 < q$

$\varphi = X$: $A \equiv i[y = \mathsf{lookUp}('X', \text{enc}(\theta')), y(\text{enc}(\theta)) \triangleleft q]^{\bullet}$ *where* $\theta' < \theta$ *or*

$$A \equiv i\left[y = \left(\begin{array}{l} \mathsf{case}\ \text{enc}(\theta')\ \mathsf{of}\ \{'X', z_{mon}\} : \_ \rightarrow z_{mon};\\ \qquad\qquad\qquad \_ : z_{tl} \rightarrow \mathsf{lookUp}('X', z_{tl});\\ \qquad\qquad\qquad \mathit{nil} \rightarrow \mathit{exit};\\ \qquad\qquad \mathsf{end} \end{array}\right),\ \ y(\text{enc}(\theta)) \triangleleft q \right]^{\bullet}$$

where $\theta' < \theta$, *or*
$A \equiv B$ *where*

– $i[y = [\![\psi]\!]^{\mathbf{m}}, y(\mathrm{enc}(\theta)) \triangleleft q]^{\bullet} \ (\xrightarrow{\tau})^{k} \ B$
– $\theta(X) = \psi$

or $A \equiv i[y = \mathit{exit}, y(\mathrm{enc}(\theta)) \triangleleft q]^{\bullet}$ *or* $A \equiv i[\mathit{exit} \triangleleft q]^{\bullet}$

$\varphi = \mathsf{max}(X, \psi)$*:* $A \equiv B$ *where* $i[[\![\psi]\!]^{\mathbf{m}}(\{'X', [\![\psi]\!]^{\mathbf{m}}\} : \mathrm{enc}(\theta)) \triangleleft q]^{\bullet}(\xrightarrow{\tau})^{k}B$ *for* $k < n$.

Proof The proof is by strong induction on $i[[\![\varphi]\!]^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft q]^{\bullet}(\xrightarrow{\tau})^{n}A$. The inductive case involves a long and tedious list of case analysis exhausting all possibilities. ■

A.1 Proofs for establishing Violation Detection

Lemma 13 uses Lemma 12 which relates possible detections by monitors synthesised from subformulas to possible detections by monitors synthesised from conjunctions using these subformulas.

Lemma 12 *For an arbitrary* θ*,* $(\nu\, i)(i_{mtr}[\mathit{mLoop}(j_1) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta))]^{\bullet}) \overset{\mathit{fail!}}{\Longrightarrow}$ *implies* $(\nu\, i)(i_{mtr}[\mathit{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\varphi_1 \wedge \varphi_2]\!]^{\mathbf{m}}(\mathrm{enc}(\theta))]^{\bullet}) \overset{\mathit{fail!}}{\Longrightarrow}$ *for any* $\varphi_2 \in$ sHML.

Proof By Definition 7, we know that we can derive the sequence of reductions

$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\varphi_1 \wedge \varphi_2]\!]^{\mathbf{m}}(\mathrm{enc}(\theta))]^{\bullet}) \Longrightarrow$$
$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel (\nu\, j, h)(i[\mathsf{fork}(j, h)]^{\bullet} \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta))]^{\bullet} \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(\mathrm{enc}(\theta))]^{\bullet}))$$

We then prove, by induction on the structure of s, the following (see [20] for details):

$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{\bullet} \parallel i[[\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^{\bullet}) \overset{\mathsf{fail!}}{\Longrightarrow} \quad \text{implies}$$
$$(\nu\, i)\begin{pmatrix} i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel \\ (\nu\, j, h)(i[\mathsf{fork}(j, h)]^{\bullet} \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^{\bullet} \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^{\bullet}) \end{pmatrix} \overset{\mathsf{fail!}}{\Longrightarrow}$$

■

Lemma 13 *If* $A, s \models_{v} \varphi\theta$ *and* $l_{env} = \mathrm{enc}(\theta)$ *then*

$$(\nu\, i)(i_{mtr}[\mathit{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\varphi]\!]^{\mathbf{m}}(l_{env})]^{\bullet}) \overset{\mathit{fail!}}{\Longrightarrow} .$$

Proof Proof by rule induction on $A, s \models_{\mathrm{v}} \varphi\theta$:

$A, s \models_{\mathrm{v}} \mathsf{ff}\theta$: Using Definition 7 for the definition of $[\![\mathsf{ff}]\!]^{\mathbf{m}}$ and the rule App (and Par and Scp), we have

$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\mathsf{ff}]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet}) \Longrightarrow (\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[\mathsf{fail!}]^{\bullet})$$

The result follows trivially, since the process i can transition with a $\mathsf{fail!}$ action in a single step using the rule SndU.

$A, s \models_{\mathrm{v}} (\varphi_1 \wedge \varphi_2)\theta$ because $A, s \models_{\mathrm{v}} \varphi_1\theta$: By $A, s \models_{\mathrm{v}} \varphi_1\theta$ and I.H. we have

$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet}) \overset{\mathsf{fail!}}{\Longrightarrow}$$

The result thus follows from Lemma 12, which allows us to conclude that

$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\varphi_1 \wedge \varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet}) \overset{\mathsf{fail!}}{\Longrightarrow}$$

$A, s \models_{\mathrm{v}} (\varphi_1 \wedge \varphi_2)\theta$ because $A, s \models_{\mathrm{v}} \varphi_2\theta$: Analogous.

$A, s \models_{\mathrm{v}} ([\alpha]\varphi)\theta$ because $s = \alpha t, A \overset{\alpha}{\Longrightarrow} B$ and $B, t \models_{\mathrm{v}} \varphi\theta$: Using the rule App Scp and Definition 7 for the property $[\alpha]\varphi$ we derive (37), by executing mLoop— see Definition 7 — we obtain (38), and then by rule Rd1 we derive (39) below.

$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(\alpha t)]^{*} \parallel i[[\![\varphi]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet}) \xrightarrow{\tau} \tag{37}$$

$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(\alpha t)]^{*} \parallel i[\mathsf{rcv}\,(\mathrm{tr}(\alpha) \rightarrow [\![\varphi]\!]^{\mathbf{m}}(l_{\mathrm{env}})\,;\, \_ \rightarrow \mathsf{ok})\,\mathsf{end}]^{\bullet}) \Longrightarrow \tag{38}$$

$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(t)]^{*} \parallel i[\mathsf{rcv}\,(\mathrm{tr}(\alpha) \rightarrow [\![\varphi]\!]^{\mathbf{m}}(l_{\mathrm{env}})\,;\, \_ \rightarrow \mathsf{ok})\,\mathsf{end} \triangleleft \mathrm{tr}(\alpha)]^{\bullet}) \xrightarrow{\tau} \tag{39}$$
$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(t)]^{*} \parallel i[[\![\varphi]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet})$$

By $B, t \models_{\mathrm{v}} \varphi\theta$ and I.H. we obtain

$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(t)]^{*} \parallel i[[\![\varphi]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet}) \overset{\mathsf{fail!}}{\Longrightarrow}$$

and, thus, the result follows by (37), (38) and (39).

$A, s \models_{\mathrm{v}} (\mathsf{max}(X, \varphi))\theta$ because $A, s \models_{\mathrm{v}} \varphi\{^{\mathsf{max}(X,\varphi)}/_{X}\}\theta$: By Definition 7 and App for process i, we derive

$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\mathsf{max}(X, \varphi)]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet}) \Longrightarrow$$
$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\varphi]\!]^{\mathbf{m}}(\{'X', [\![\varphi]\!]^{\mathbf{m}}\} : l_{\mathrm{env}})]^{\bullet}) \tag{40}$$

Assuming the appropriate α-conversion for X in $\mathsf{max}(X, \varphi)$, we note that from $l_{\mathrm{env}} = \mathrm{enc}(\theta)$ and Definition 8 we obtain

$$\mathrm{enc}(\{^{\mathsf{max}(X,\varphi)}/_{X}\}\theta) = \{'X', [\![\varphi]\!]^{\mathbf{m}}\} : l_{\mathrm{env}} \tag{41}$$

By $A, s \models_{\mathrm{v}} \varphi\{^{\mathsf{max}(X,\varphi)}/_{X}\}\rho$, (41) and I.H. we obtain

$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\varphi]\!]^{\mathbf{m}}(\{'X', [\![\varphi]\!]^{\mathbf{m}}\} : l_{\mathrm{env}})]^{\bullet}) \overset{\mathsf{fail!}}{\Longrightarrow} \tag{42}$$

The result follows from (40) and (42). ■

Lemma 16 relies on a technical result, Lemma 15 which allows us to recover a violating reduction sequence for a subformula φ_1 or φ_2 from that of the synthesised monitor of a conjunction formula $\varphi_1 \wedge \varphi_2$. Lemma 15 relies on Lemma 14.

Lemma 14 *For some $l \leq n$:*

$$(\nu\, j, h)\Big(i\,\big[\mathsf{fork}(j, h) \triangleleft q_{frk}\big]^{\bullet} \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{env}) \triangleleft q]^{\bullet} \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{env}) \triangleleft r]^{\bullet}\Big)(\xrightarrow{\tau})^{n} \xrightarrow{\mathsf{fail!}}$$
$$\textit{implies}\quad (\nu\, j)(i_{mtr}[\mathsf{mLoop}(j) \triangleleft q_{frk}]^{*} \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{env}) \triangleleft q]^{\bullet})(\xrightarrow{\tau})^{l} \xrightarrow{\mathsf{fail!}}$$
$$\textit{or}\quad (\nu\, h)(i_{mtr}[\mathsf{mLoop}(h) \triangleleft q_{frk}]^{*} \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{env}) \triangleleft r]^{\bullet})(\xrightarrow{\tau})^{l} \xrightarrow{\mathsf{fail!}}$$

Proof By induction on the structure of the mailbox q_{frk} at actor i. ■

Lemma 15 *For some $l \leq n$*

$$(\nu\, i)\left(i_{mtr}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel (\nu\, j, h)\begin{pmatrix} i\,[\mathsf{fork}(j, h) \triangleleft \mathrm{tr}(t)]^{\bullet} \\ \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{env})]^{\bullet} \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{env})]^{\bullet} \end{pmatrix}\right)(\xrightarrow{\tau})^{k} \xrightarrow{\mathsf{fail!}}$$
$$\textit{implies}\quad (\nu\, i)(i_{mtr}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(ts)]^{*} \parallel i[[\![\varphi_1]\!]^{\mathbf{m}}(l_{env})]^{\bullet})(\xrightarrow{\tau})^{l} \xrightarrow{\mathsf{fail!}}$$
$$\textit{or}\quad (\nu\, i)(i_{mtr}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(ts)]^{*} \parallel i[[\![\varphi_2]\!]^{\mathbf{m}}(l_{env})]^{\bullet})(\xrightarrow{\tau})^{l} \xrightarrow{\mathsf{fail!}}$$

Proof Proof by induction on the structure of s.

$s = \epsilon$: From the structure of mLoop, we know that after the function application, the actor $i_{\mathrm{mtr}}[\mathsf{mLoop}(i)]^{*}$ is stuck. Thus we conclude that it must be the case that

$$(\nu\, j, h)\begin{pmatrix} i\,[\mathsf{fork}(j, h) \triangleleft \mathrm{tr}(t)]^{\bullet} \\ \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet} \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet} \end{pmatrix}(\xrightarrow{\tau})^{k} \xrightarrow{\mathsf{fail!}}$$

where $k = n$ or $k = n - 1$. In either case, the required result follows from Lemma 14.

$s = \alpha s'$: We have two subcases:

– If
$$(\nu\, j,h)\begin{pmatrix} i\,[\mathsf{fork}(j,h) \triangleleft \mathrm{tr}(t)]^{\bullet} \\ \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet} \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet}\end{pmatrix}(\xrightarrow{\tau})^k \xrightarrow{\mathsf{fail!}}$$
for some $k \leq n$ then, by Lemma 14 we obtain
$$\begin{aligned} &(\nu\, j)(i_{\mathrm{mtr}}[\mathsf{mLoop}(j) \triangleleft \mathrm{tr}(t)]^* \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet})(\xrightarrow{\tau})^l \xrightarrow{\mathsf{fail!}} \\ \text{or}\quad &(\nu\, h)(i_{\mathrm{mtr}}[\mathsf{mLoop}(h) \triangleleft \mathrm{tr}(t)]^* \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet})(\xrightarrow{\tau})^l \xrightarrow{\mathsf{fail!}} \end{aligned}$$
for some $l \leq k$. By Lemma 8 we thus obtain
$$\begin{aligned} &(\nu\, j)(i_{\mathrm{mtr}}[\mathsf{mLoop}(j) \triangleleft \mathrm{tr}(ts)]^* \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet})(\xrightarrow{\tau})^l \xrightarrow{\mathsf{fail!}} \\ \text{or}\quad &(\nu\, h)(i_{\mathrm{mtr}}[\mathsf{mLoop}(h) \triangleleft \mathrm{tr}(ts)]^* \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet})(\xrightarrow{\tau})^l \xrightarrow{\mathsf{fail!}} \end{aligned}$$
as required.

– Otherwise, it must be the case that
$$(\nu\, i)\begin{pmatrix} i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^* \\ \parallel (\nu\, j,h)\begin{pmatrix} i\,[\mathsf{fork}(j,h) \triangleleft \mathrm{tr}(t)]^{\bullet} \\ \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet} \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet}\end{pmatrix}\end{pmatrix}(\xrightarrow{\tau})^k \tag{43}$$
$$(\nu\, i)\begin{pmatrix} i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s')]^* \\ \parallel (\nu\, j,h)\,(i\,[e_{\mathrm{fork}} \triangleleft q : \mathrm{tr}(\alpha)]^{\bullet} \parallel A)\end{pmatrix}(\xrightarrow{\tau})^{n-k} \xrightarrow{\mathsf{fail!}} \tag{44}$$
For some $k = 3 + k_1$ where
$$\begin{aligned} &(\nu\, j,h)\begin{pmatrix} i\,[\mathsf{fork}(j,h) \triangleleft \mathrm{tr}(t)]^{\bullet} \\ \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet} \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet}\end{pmatrix}(\xrightarrow{\tau})^{k_1} \\ &\qquad\qquad (\nu\, j,h)\,(i\,[e_{\mathrm{fork}} \triangleleft q]^{\bullet} \parallel A) \end{aligned} \tag{45}$$
By (45) and Lemma 8 we obtain
$$\begin{aligned} &(\nu\, j,h)\begin{pmatrix} i\,[\mathsf{fork}(j,h) \triangleleft \mathrm{tr}(t) : \mathrm{tr}(\alpha)]^{\bullet} \\ \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet} \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet}\end{pmatrix}(\xrightarrow{\tau})^{k_1} \\ &\qquad\qquad (\nu\, j,h)\,(i\,[e_{\mathrm{fork}} \triangleleft q : \mathrm{tr}(\alpha)]^{\bullet} \parallel A) \end{aligned}$$
and by (44) we can construct the sequence of transitions:
$$(\nu\, i)\begin{pmatrix} i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s')]^* \\ \parallel (\nu\, j,h)\begin{pmatrix} i\,[\mathsf{fork}(j,h) \triangleleft \mathrm{tr}(t) : \alpha]^{\bullet} \\ \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet} \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet}\end{pmatrix}\end{pmatrix}(\xrightarrow{\tau})^{n-3} \xrightarrow{\mathsf{fail!}}$$
Thus, by I.H. we obtain, for some $l \leq n - 3$
$$\begin{aligned} &(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(t\alpha s')]^* \parallel i[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet})(\xrightarrow{\tau})^l \xrightarrow{\mathsf{fail!}} \\ \text{or}\quad &(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(t\alpha s')]^* \parallel i[[\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet})(\xrightarrow{\tau})^l \xrightarrow{\mathsf{fail!}} \end{aligned}$$
The result follows since $s = \alpha s'$. ■

Equipped with Lemma 15, we can now prove Lemma 16.

Lemma 16 *If* $A \overset{s}{\Longrightarrow}$, $l_{env} = \mathrm{enc}(\theta)$ *and* $(\nu\, i)(i_{mtr}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^* \parallel i[[\![\varphi]\!]^{\mathbf{m}}(l_{env})]^{\bullet}) \overset{\mathsf{fail!}}{\Longrightarrow}$ *then* $A, s \models_v \varphi\theta$, *whenever* $\mathrm{fv}(\varphi) \subseteq \mathrm{dom}(\theta)$.

Proof By strong induction on the number of transitions n, leading to the action $\mathsf{fail!}$
$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^* \parallel i[[\![\varphi]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet})(\xrightarrow{\tau})^n \xrightarrow{\mathsf{fail!}}$

$n = 0$: By inspection of the definition for mLoop, and by case analysis of $[\![\varphi]\!]^{\mathbf{m}}(l_{\text{env}})$ from Definition 7, it can never be the case that

$$(\nu\, i)(i_{\text{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^* \parallel i[[\![\varphi]\!]^{\mathbf{m}}(l_{\text{env}})]^{\bullet}) \xrightarrow{\mathsf{fail!}}$$

. Thus the result holds trivially.

$n = k + 1$: We proceed by case analysis on φ.

$\varphi = \mathsf{ff}$: The result holds immediately for any A and s by Definition 3.

$\varphi = [\alpha]\psi$: By Definition 7, we know that

$$(\nu\, i)(i_{\text{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^* \parallel i[[\![[\alpha]\psi]\!]^{\mathbf{m}}(l_{\text{env}})]^{\bullet})(\xrightarrow{\tau})^{k_1} \tag{46}$$

$$(\nu\, i)(i_{\text{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_2)]^* \parallel i[[\![[\alpha]\psi]\!]^{\mathbf{m}}(l_{\text{env}}) \triangleleft \mathrm{tr}(s_1)]^{\bullet}) \xrightarrow{\tau} \tag{47}$$

$$(\nu\, i)\left(\begin{array}{l} i_{\text{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_2)]^* \parallel \\ i\left[\mathsf{rcv}\left(\begin{array}{l}\mathrm{tr}(\alpha)\rightarrow [\![\psi]\!]^{\mathbf{m}}(l_{\text{env}})\,; \\ \_ \rightarrow \mathsf{ok}\end{array}\right) \mathsf{end} \triangleleft \mathrm{tr}(s_1)\right]^{\bullet}\end{array}\right)(\xrightarrow{\tau})^{k_2}\xrightarrow{\mathsf{fail!}} \tag{48}$$

$$\text{where } k + 1 = k_1 + k_2 + 1 \text{ and } s = s_1 s_2 \tag{49}$$

From the analysis of the code in (48), the only way for the action $\mathsf{fail!}$ to be triggered is by choosing the guarded branch $\mathrm{tr}(\alpha) \rightarrow [\![\varphi]\!]^{\mathbf{m}}(l_{\text{env}})$ in actor i. This means that (48) can be decomposed into the following reduction sequences.

$$(\nu\, i)\left(\begin{array}{l} i_{\text{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_2)]^* \parallel \\ i\,[\mathsf{rcv}\ (\mathrm{tr}(\alpha) \rightarrow [\![\psi]\!]^{\mathbf{m}}(l_{\text{env}})\,;\, \_ \rightarrow \mathsf{ok})\ \mathsf{end} \triangleleft \mathrm{tr}(s_1)]^{\bullet}\end{array}\right)(\xrightarrow{\tau})^{k_3} \tag{50}$$

$$(\nu\, i)\left(\begin{array}{l} i_{\text{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_4)]^* \parallel \\ i\,[\mathsf{rcv}\ (\mathrm{tr}(\alpha) \rightarrow [\![\psi]\!]^{\mathbf{m}}(l_{\text{env}})\,;\, \_ \rightarrow \mathsf{ok})\ \mathsf{end} \triangleleft \mathrm{tr}(s_1 s_3)]^{\bullet}\end{array}\right) \xrightarrow{\tau} \tag{51}$$

$$(\nu\, i) i_{\text{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_4)]^* \parallel i\ [[\![\psi]\!]^{\mathbf{m}}(l_{\text{env}}) \triangleleft \mathrm{tr}(s_5)]^{\bullet}(\xrightarrow{\tau})^{k_4} \xrightarrow{\mathsf{fail!}} \tag{52}$$

$$\text{where } k_2 = k_3 + k_4 + 1 \text{ and } s_1 s_3 = \alpha s_5 \text{ and } s_2 = s_3 s_4 \tag{53}$$

By (49) and (53) we derive

$$s = \alpha t \text{ where } t = s_5 s_4 \tag{54}$$

From the definition of mLoop we can derive

$$\begin{array}{l}(\nu\, i)(i_{\text{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(t)]^* \parallel i[[\![\psi]\!]^{\mathbf{m}}(l_{\text{env}})]^{\bullet})(\xrightarrow{\tau})^{k_5} \\ \qquad\qquad (\nu\, i)(i_{\text{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_4)]^* \parallel i\ [[\![\psi]\!]^{\mathbf{m}}(l_{\text{env}}) \triangleleft \mathrm{tr}(s_5)]^{\bullet})\end{array} \tag{55}$$

where $k_5 \leq k_1 + k_3$. From (54) we can split $A \overset{s}{\Longrightarrow}$ as $A \overset{\alpha}{\Longrightarrow} A' \overset{t}{\Longrightarrow}$ and from (55), (52), the fact that $k_5 + k_4 < k + 1 = n$ from (49) and (53), and I.H. we obtain

$$A', t \models_{\text{v}} \psi\theta \tag{56}$$

From (56), $A \overset{\alpha}{\Longrightarrow} A'$ and Definition 3 we thus conclude $A, s \models_{\text{v}} ([\alpha]\psi)\theta$.

$\varphi = \varphi_1 \wedge \varphi_2$ From Definition 7, we can decompose the transition sequence as follows

$$(\nu\, i)(i_{\text{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^* \parallel i[[\![\varphi_1 \wedge \varphi_2]\!]^{\mathbf{m}}(l_{\text{env}})]^{\bullet})(\xrightarrow{\tau})^{k_1} \tag{57}$$

$$(\nu\, i)(i_{\text{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_2)]^* \parallel i[[\![\varphi_1 \wedge \varphi_2]\!]^{\mathbf{m}}(l_{\text{env}}) \triangleleft \mathrm{tr}(s_1)]^{\bullet}) \xrightarrow{\tau} \tag{58}$$

$$(\nu\, i)\left(\begin{array}{l} i_{\text{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_2)]^* \\ \parallel i\left[\begin{array}{l} y_1 = \mathsf{spw}\,([\![\varphi_1]\!]^{\mathbf{m}}(l_{\text{env}})), \\ y_2 = \mathsf{spw}\,([\![\varphi_2]\!]^{\mathbf{m}}(l_{\text{env}})), \mathsf{fork}(y_1, y_2)\end{array} \triangleleft \mathrm{tr}(s_1)\right]^{\bullet}\end{array}\right)(\xrightarrow{\tau})^{k_2} \tag{59}$$

$$(\nu\, i)\left(\begin{array}{l} i_{\text{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_4)]^* \\ \parallel i\left[\begin{array}{l} y_1 = \mathsf{spw}\,([\![\varphi_1]\!]^{\mathbf{m}}(l_{\text{env}})), \\ y_2 = \mathsf{spw}\,([\![\varphi_2]\!]^{\mathbf{m}}(l_{\text{env}})), \mathsf{fork}(y_1, y_2)\end{array} \triangleleft \mathrm{tr}(s_1 s_3)\right]^{\bullet}\end{array}\right)(\xrightarrow{\tau})^{2} \tag{60}$$

$$(\nu\, i)\left(\begin{array}{l} i_{\text{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_4)]^* \\ \parallel (\nu\, j)\left(\begin{array}{l} i\left[\begin{array}{l} y_2 = \mathsf{spw}\,([\![\varphi_2]\!]^{\mathbf{m}}(l_{\text{env}})), \\ \mathsf{fork}(j, y_2)\end{array} \triangleleft \mathrm{tr}(s_1 s_3)\right]^{\bullet} \\ \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\text{env}})]^{\bullet}\end{array}\right)\end{array}\right)(\xrightarrow{\tau})^{k_3} \xrightarrow{\mathsf{fail!}} \tag{61}$$

$$\text{where } k + 1 = k_1 + 1 + k_2 + 2 + k_3,\ s = s_1 s_2 \text{ and } s_2 = s_3 s_4 \tag{62}$$

From (61) we can deduce that there are two possible transition sequences how action fail! was reached:

1. If fail! was reached because $j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\text{env}})]^{\bullet}(\xrightarrow{\tau})^{k_4}\xrightarrow{\textsf{fail!}}$ on its own, for some $k_4 \leq k_3$ then, by Par and Scp we deduce

$$(\nu\, i)(i_{\text{mtr}}[\textsf{mLoop}(i) \triangleleft \text{tr}(s)]^{*} \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\text{env}})]^{\bullet})(\xrightarrow{\tau})^{k_4}\xrightarrow{\textsf{fail!}}$$

   From (62) we know that $k_4 < k+1 = n$, and by the premise $A \overset{s}{\Longrightarrow}$ and I.H. we obtain $A, s \models_{\text{v}} \varphi_1\theta$. By Definition 3 we then obtain $A, s \models_{\text{v}} (\varphi_1 \wedge \varphi_2)\theta$
2. Alternatively, (61) can be decomposed further as

$$(\nu\, i)\left(\begin{array}{l} i_{\text{mtr}}[\textsf{mLoop}(i) \triangleleft \text{tr}(s_4)]^{*} \\ \parallel (\nu\, j)\left(\begin{array}{l} i\left[\begin{array}{l} y_2 = \textsf{spw}\,([\![\varphi_2]\!]^{\mathbf{m}}(l_{\text{env}})), \\ \textsf{fork}(j, y_2) \end{array} \triangleleft \text{tr}(s_1 s_3)\right]^{\bullet} \\ \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\text{env}})]^{\bullet} \end{array}\right) \end{array}\right)(\xrightarrow{\tau})^{k_4} \tag{63}$$

$$(\nu\, i)\left(\begin{array}{l} i_{\text{mtr}}[\textsf{mLoop}(i) \triangleleft \text{tr}(s_6)]^{*} \\ \parallel (\nu\, j)\left(\begin{array}{l} i\left[\begin{array}{l} y_2 = \textsf{spw}\,([\![\varphi_2]\!]^{\mathbf{m}}(l_{\text{env}})), \\ \textsf{fork}(j, y_2) \end{array} \triangleleft \text{tr}(s_1 s_3 s_5)\right]^{\bullet} \\ \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\text{env}})]^{\bullet} \end{array}\right) \end{array}\right)(\xrightarrow{\tau})^{2} \tag{64}$$

$$(\nu\, i)\left(\begin{array}{l} i_{\text{mtr}}[\textsf{mLoop}(i) \triangleleft \text{tr}(s_6)]^{*} \\ \parallel (\nu\, j, h)\left(\begin{array}{l} i\,[\textsf{fork}(j, h) \triangleleft \text{tr}(s_1 s_3 s_5)]^{\bullet} \\ \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\text{env}})]^{\bullet} \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{\text{env}})]^{\bullet} \end{array}\right) \end{array}\right)(\xrightarrow{\tau})^{k_5}\xrightarrow{\textsf{fail!}} \tag{65}$$

$$\text{where} k_3 = k_4 + 2 + k_5 \text{ and } s_4 = s_5 s_6 \tag{66}$$

   From (65) and Lemma 15 we know that, for some $k_6 \leq k_5$ either

$$(\nu\, i)(i_{\text{mtr}}[\textsf{mLoop}(i) \triangleleft \text{tr}(s_1 s_3 s_5 s_6)]^{*} \parallel i[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\text{env}})]^{\bullet})(\xrightarrow{\tau})^{k_6}\xrightarrow{\textsf{fail!}}$$
$$\text{or} \quad (\nu\, i)(i_{\text{mtr}}[\textsf{mLoop}(i) \triangleleft \text{tr}(s_1 s_3 s_5 s_6)]^{*} \parallel i[[\![\varphi_2]\!]^{\mathbf{m}}(l_{\text{env}})]^{\bullet})(\xrightarrow{\tau})^{k_6}\xrightarrow{\textsf{fail!}}$$

   From (62) and (66) we know that $s = s_1 s_3 s_5 s_6$ and that $k_6 < k{+}1 = n$. By I.H., we obtain either $A, s \models_{\text{v}} \varphi_1\theta$ or $A, s \models_{\text{v}} \varphi_2\theta$ and, in either case, by Definition 3 we deduce $A, s \models_{\text{v}} (\varphi_1 \wedge \varphi_2)\theta$.

$\varphi = X$ By Definition 7, we can deconstruct

$$(\nu\, i)(i_{\text{mtr}}[\textsf{mLoop}(i) \triangleleft \text{tr}(s)]^{*} \parallel i[[\![X]\!]^{\mathbf{m}}(l_{\text{env}})]^{\bullet})(\xrightarrow{\tau})^{k+1}\xrightarrow{\textsf{fail!}}$$

as

$$(\nu\, i)(i_{\text{mtr}}[\textsf{mLoop}(i) \triangleleft \text{tr}(s)]^{*} \parallel i[[\![X]\!]^{\mathbf{m}}(l_{\text{env}})]^{\bullet}) \Longrightarrow\xrightarrow{\tau} \tag{67}$$

$$(\nu\, i)(i_{\text{mtr}}[\textsf{mLoop}(i) \triangleleft \text{tr}(s_2)]^{*} \parallel i[y = \textsf{lookUp}('X', l_{\text{env}}), y(l_{\text{env}}) \triangleleft \text{tr}(s_1)]^{\bullet}) \Longrightarrow\xrightarrow{\tau} \tag{68}$$

$$(\nu\, i)(i_{\text{mtr}}[\textsf{mLoop}(i) \triangleleft \text{tr}(s_4)]^{*} \parallel i[y = v, y(l_{\text{env}}) \triangleleft \text{tr}(s_1 s_3)]^{\bullet}) \Longrightarrow\xrightarrow{\tau} \tag{69}$$

$$(\nu\, i)(i_{\text{mtr}}[\textsf{mLoop}(i) \triangleleft \text{tr}(s_6)]^{*} \parallel i[v(l_{\text{env}}) \triangleleft \text{tr}(s_1 s_3 s_5)]^{\bullet}) \Longrightarrow\xrightarrow{\textsf{fail!}} \tag{70}$$

where $s = s_1 s_2$, $s_2 = s_3 s_4$ and $s_4 = s_5 s_6$

Since $X \in \text{dom}(\theta)$, we know that $\theta(X) = \psi$ for some ψ. By the assumption $l_{\text{env}} = \text{enc}(\theta)$ and Lemma 6 we obtain that $v = [\![\psi]\!]^{\mathbf{m}}$. Hence, by (67), (68), (69) and (70) we can reconstruct

$$(\nu\, i)(i_{\text{mtr}}[\textsf{mLoop}(i) \triangleleft \text{tr}(s)]^{*} \parallel i[[\![\psi]\!]^{\mathbf{m}}(l_{\text{env}})]^{\bullet})(\xrightarrow{\tau})^{k_1}$$
$$(\nu\, i)(i_{\text{mtr}}[\textsf{mLoop}(i) \triangleleft \text{tr}(s_6)]^{*} \parallel i[[\![\psi]\!]^{\mathbf{m}}(l_{\text{env}}) \triangleleft \text{tr}(s_1 s_3 s_5)]^{\bullet})(\xrightarrow{\tau})^{k_2}\xrightarrow{\textsf{fail!}} \tag{71}$$

where $k_1 + k_2 < k + 1 = n$. By (71) and I.H. we obtain $A, s \models_{\text{v}} \psi$, which is the result required, since by $\theta(X) = \psi$ we know that $X\theta = \psi$.

$\varphi = \mathsf{max}(X, \psi)$ By Definition 7, we can deconstruct

$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[\llbracket \mathsf{max}(X,\psi) \rrbracket^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet})(\xrightarrow{\tau})^{k+1} \xrightarrow{\mathsf{fail!}}$$

as follows:

$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[\llbracket \mathsf{max}(X,\psi) \rrbracket^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet})(\xrightarrow{\tau})^{k_1} \xrightarrow{\tau}$$
$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_2)]^{*} \parallel i[\llbracket \psi \rrbracket^{\mathbf{m}}(\{'X', \psi\} : l_{\mathrm{env}}) \triangleleft \mathrm{tr}(s_1)]^{\bullet})(\xrightarrow{\tau})^{k_2} \xrightarrow{\mathsf{fail!}}$$

from which we can reconstruct the transition sequence

$$(\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[\llbracket \psi \rrbracket^{\mathbf{m}}(\{'X', \psi\} : l_{\mathrm{env}})]^{\bullet})(\xrightarrow{\tau})^{k_1+k_2} \xrightarrow{\mathsf{fail!}} \tag{72}$$

By the assumption $l_{\mathrm{env}} = \Gamma(\theta)$ we deduce that $\{'X', \psi\} : l_{\mathrm{env}} = \mathrm{enc}(\{^{\mathsf{max}(X,\psi)}/\}\theta)$ and, since $k_1 + k_2 < k + 1 = n$, we can use (72), $A \overset{s}{\Longrightarrow}$ and I.H. to obtain $A, s \models_{\mathrm{v}} \psi\{^{\mathsf{max}(X,\psi)}/_{X}\}\theta$. By Definition 3 we then conclude $A, s \models_{\mathrm{v}} \mathsf{max}(X, \psi)\theta$. ■

A.2 Proofs for establishing Detection Preservation

Lemma 18 relies heavily on Lemma 17.

Lemma 17 (Translation Confluence) *For all* $\varphi \in \mathrm{sHML}, q \in (\mathrm{VAL})^{*}$ *and* $\theta :: \mathrm{LVAR} \rightharpoonup \mathrm{sHML}$, $i[\llbracket \varphi \rrbracket^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^{\bullet} \Longrightarrow A$ *implies* $\mathrm{cnf}(A)$.

Proof Proof by strong numerical induction on n in $i[\llbracket \varphi \rrbracket^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^{\bullet}(\xrightarrow{\tau})^{n}A$.

$n = 0$: The only possible τ-action that can be performed by $i[\llbracket \varphi \rrbracket^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^{\bullet}$ is that for the function application of the monitor definition, *i.e.*,

$$i[\llbracket \varphi \rrbracket^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^{\bullet} \xrightarrow{\tau} i[e \triangleleft q]^{\bullet} \text{ for some } e. \tag{73}$$

Apart from that $i[\llbracket \varphi \rrbracket^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^{\bullet}$ can also only perform input action at i, *i.e.*,

$$i[\llbracket \varphi \rrbracket^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^{\bullet} \xrightarrow{i?v} i[\llbracket \varphi \rrbracket^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q : v]^{\bullet}$$

On the one hand, we can derive $i[e \triangleleft q]^{\bullet} \xrightarrow{i?v} i[e \triangleleft q : v]^{\bullet}$. Moreover, from (73) and Lemma 8 we can deduce $i[\llbracket \varphi \rrbracket^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q : v]^{\bullet} \xrightarrow{\tau} i[e \triangleleft q : v]^{\bullet}$ which allows us to close the confluence diamond.

$n = k + 1$: We proceed by case analysis on the property φ, using Lemma 11 to infer the possible structures of the resulting process. Again, most involving cases are those for conjunction translations, as they generate more than one concurrent actor; we discuss one of these below:

$\varphi = \varphi_1 \wedge \varphi_2$: By Lemma 11, A can have any of 4 general structures, one of which is

$$A \equiv (\nu\, j_1, j_2)\left(i[j_2!u, \mathsf{fork}(j_1, j_2) \triangleleft q]^{\bullet} \begin{array}{l} \parallel (\nu \widetilde{h_1})(j_1[e_1 \triangleleft q_1']^{\bullet} \parallel B) \\ \parallel (\nu \widetilde{h_2})(j_2[e_2 \triangleleft q_2']^{\bullet} \parallel C) \end{array} \right) \tag{74}$$

where

$$j_1[\llbracket \varphi_1 \rrbracket^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft q_1]^{\bullet}\ (\xrightarrow{\tau})^{k}\ (\nu \widetilde{h_1})(j_1[e_1 \triangleleft q_1']^{\bullet} \parallel B) \text{ for } k < n, q_1 < q \tag{75}$$
$$j_2[\llbracket \varphi_2 \rrbracket^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft q_2]^{\bullet}\ (\xrightarrow{\tau})^{l}\ (\nu \widetilde{h_2})(j_2[e_2 \triangleleft q_2']^{\bullet} \parallel C) \text{ for } l < n, q_2 < q \tag{76}$$

By Lemma 11, (75) and (76) we also infer that the only external action that can be performed by the processes $(\nu \widetilde{h_1})(j_1[e_1 \triangleleft q_1']^{\bullet} \parallel B)$ and $(\nu \widetilde{h_2})(j_2[e_2 \triangleleft q_2']^{\bullet} \parallel C)$ is $\mathsf{fail!}$. Moreover by (75) and (76) we can also show that

$$\mathrm{fId}\big((\nu \widetilde{h_1})(j_1[e_1 \triangleleft q_1']^{\bullet} \parallel B)\big) = \{j_1\} \qquad \mathrm{fId}\big((\nu \widetilde{h_2})(j_2[e_2 \triangleleft q_2']^{\bullet} \parallel C)\big) = \{j_2\}$$

Thus these two subactors cannot communicate with each other or send messages to actor i. This also means that the remaining possible actions that A can perform are:

$$A \xrightarrow{\tau} (\nu\, j_1, j_2)\left(i[u, \mathsf{fork}(j_1, j_2) \triangleleft q]^\bullet \begin{array}{l} \parallel (\nu \widetilde{h}_1)(j_1[e_1 \triangleleft q_1']^\bullet \parallel B) \\ \parallel (\nu \widetilde{h}_2)(j_2[e_2 \triangleleft q_2' : u]^\bullet \parallel C) \end{array}\right) \quad \text{or} \tag{77}$$

$$A \xrightarrow{\tau} (\nu\, j_1, j_2)\left(i[j_2!u, \mathsf{fork}(j_1, j_2) \triangleleft q]^\bullet \begin{array}{l} \parallel (\nu \widetilde{h}_1')(j_1[e_1' \triangleleft q_1'']^\bullet \parallel B') \\ \parallel (\nu \widetilde{h}_2)(j_2[e_2 \triangleleft q_2']^\bullet \parallel C) \end{array}\right) \tag{78}$$

because $(\nu \widetilde{h}_1)(j_1[e_1 \triangleleft q_1']^\bullet \parallel B) \xrightarrow{\tau} (\nu\ \widetilde{h}_1'\ )(j_1[e_1' \triangleleft q_1'']^\bullet \parallel B')$ or

$$A \xrightarrow{\tau} (\nu\, j_1, j_2)\left(\ i[j_2!u, \mathsf{fork}(j_1, j_2) \triangleleft q]^\bullet \begin{array}{l} \parallel (\nu \widetilde{h}_1)(j_1[e_1 \triangleleft q_1']^\bullet \parallel B) \\ \parallel (\nu \widetilde{h}_2')(j_2[e_2' \triangleleft q_2'']^\bullet \parallel C') \end{array}\right) \tag{79}$$

because $(\nu \widetilde{h}_2)(j_2[e_2 \triangleleft q_2']^\bullet \parallel C) \xrightarrow{\tau} (\nu\ \widetilde{h}_2'\ )(j_2[e_2' \triangleleft q_2'']^\bullet \parallel C')$ or

$$A \xrightarrow{i?v} (\nu\, j_1, j_2)\left(\begin{array}{l} i[j_2!u, \mathsf{fork}(j_1, j_2) \triangleleft q : v]^\bullet \\ \parallel (\nu \widetilde{h}_1)(j_1[e_1 \triangleleft q_1']^\bullet \parallel B) \parallel (\nu \widetilde{h}_2)(j_2[e_2 \triangleleft q_2']^\bullet \parallel C) \end{array}\right) \tag{80}$$

We prove confluence for the pair of actions (77) and (79) and leave the other combinations for the interested reader. From (79) and Lemma 8 we derive

$$(\nu \widetilde{h}_2)(j_2[e_2 \triangleleft q_2' : u]^\bullet \parallel C) \xrightarrow{\tau} (\nu\ \widetilde{h}_2'\ )(j_2[e_2' \triangleleft q_2'' : u]^\bullet \parallel C')$$

and by Par and Scp we obtain

$$\begin{array}{r} (\nu\, j_1, j_2)\left(\ i[u, \mathsf{fork}(j_1, j_2) \triangleleft q]^\bullet \begin{array}{l} \parallel (\nu \widetilde{h}_1)(j_1[e_1 \triangleleft q_1']^\bullet \parallel B) \\ \parallel (\nu \widetilde{h}_2)(j_2[e_2 \triangleleft q_2' : u]^\bullet \parallel C) \end{array}\right) \xrightarrow{\tau} \\ (\nu\, j_1, j_2)\left(\ i[u, \mathsf{fork}(j_1, j_2) \triangleleft q]^\bullet \begin{array}{l} \parallel (\nu \widetilde{h}_1)(j_1[e_1 \triangleleft q_1']^\bullet \parallel B) \\ \parallel (\nu \widetilde{h}_2')(j_2[e_2' \triangleleft q_2'' : u]^\bullet \parallel C') \end{array}\right) \end{array} \tag{81}$$

Using Com, Str, Par and Scp we can derive

$$\begin{array}{r} (\nu\, j_1, j_2)\left(\ i[j_2!u, \mathsf{fork}(j_1, j_2) \triangleleft q]^\bullet \begin{array}{l} \parallel (\nu \widetilde{h}_1)(j_1[e_1 \triangleleft q_1']^\bullet \parallel B) \\ \parallel (\nu \widetilde{h}_2')(j_2[e_2' \triangleleft q_2'']^\bullet \parallel C') \end{array}\right) \xrightarrow{\tau} \\ (\nu\, j_1, j_2)\left(\ i[u, \mathsf{fork}(j_1, j_2) \triangleleft q]^\bullet \begin{array}{l} \parallel (\nu \widetilde{h}_1)(j_1[e_1 \triangleleft q_1']^\bullet \parallel B) \\ \parallel (\nu \widetilde{h}_2')(j_2[e_2' \triangleleft q_2'' : u]^\bullet \parallel C') \end{array}\right) \end{array} \tag{82}$$

thus we close the confluence diamond by (81) and (82). ■

Lemma 18 (Weak Confluence) *For all $\varphi \in$ sHML, $q \in$ Val**

$$i_{mtr}[\mathsf{Mon}(\varphi) \triangleleft q]^* \Longrightarrow A \ \textit{ implies } \ \mathrm{cnf}(A)$$

Proof By strong induction on n, the number of transitions in $i_{\mathrm{mtr}}[\mathsf{Mon}(\varphi) \triangleleft q]^*\ (\xrightarrow{\tau})^n\ A$.

$n = 0$ We know $A = i_{\mathrm{mtr}}[\mathsf{Mon}(\varphi) \triangleleft q]^*$. It is confluent because it can perform either of two actions, namely a τ-action for the function application (see App in Fig. 2), or else an external input at i_{mtr}, (see RcvU in Fig. 2). The matching moves can be constructed by RcvU on the one hand, and by Lemma 8 on the other, analogously to the base case of Lemma 17.

$n = k + 1$ By performing an analysis similar to that of Lemma 11, but for $i_{\mathrm{mtr}}[\mathsf{Mon}(\varphi) \triangleleft q]^*$ instead, we can determine that this actor can only weakly transition to either of the forms below whereby, for cases (*ii*) to (*v*), we obtain B as a result of $i[[\![\varphi]\!]^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft r]^\bullet \Longrightarrow B$ for some r:

(*i*) $A = i_{\mathrm{mtr}}[M = \mathsf{spw}\,([\![\varphi]\!]^{\mathbf{m}}(\mathsf{nil})), \mathsf{mLoop}(M) \triangleleft q]^*$
(*ii*) $A \equiv (\nu\, i)(i_{\mathrm{mtr}}[\mathsf{mLoop}(i) \triangleleft q]^* \parallel B)$
(*iii*) $A \equiv (\nu\, i)(i_{\mathrm{mtr}}[\mathsf{rcv}\, z \rightarrow i!z\, \mathsf{end}, \mathsf{mLoop}(i) \triangleleft q]^* \parallel B)$
(*iv*) $A \equiv (\nu\, i)(i_{\mathrm{mtr}}[i!v, \mathsf{mLoop}(i) \triangleleft q]^* \parallel B)$
(*v*) $A \equiv (\nu\, i)(i_{\mathrm{mtr}}[v, \mathsf{mLoop}(i) \triangleleft q]^* \parallel B)$

We here focus on the 4th case of monitor structure; the other cases are analogous. From $i[[\![\varphi]\!]^{\mathbf{m}}(l_{\text{env}}) \triangleleft r]^{\bullet} \Longrightarrow B$ and Lemma 11 we know that

$$B \xrightarrow{\gamma} \quad \text{implies } \gamma = \mathsf{fail!} \text{ or } \gamma = \tau$$
$$B \equiv (\nu\,\mathbf{h})(i[e \triangleleft r]^{\bullet} \parallel C) \quad \text{where } \mathrm{fId}(B) = i$$

This means that $(\nu\, i)(i_{\text{mtr}}[i!v, \mathsf{mLoop}(i) \triangleleft q]^{*} \parallel B)$ can only exhibit the following actions:

$$(\nu\, i)(i_{\text{mtr}}[i!v, \mathsf{mLoop}(i) \triangleleft q]^{*} \parallel B) \xrightarrow{i_{\text{mtr}}?u} (\nu\, i)(i_{\text{mtr}}[i!v, \mathsf{mLoop}(i) \triangleleft q : u]^{*} \parallel B) \tag{83}$$

$$(\nu\, i)(i_{\text{mtr}}[i!v, \mathsf{mLoop}(i) \triangleleft q]^{*} \parallel B) \xrightarrow{\tau} (\nu\, i)(i_{\text{mtr}}[v, \mathsf{mLoop}(i) \triangleleft q]^{*} \parallel (\nu\,\mathbf{h})(i[e \triangleleft r : v]^{\bullet} \parallel C)) \tag{84}$$

$$(\nu\, i)(i_{\text{mtr}}[i!v, \mathsf{mLoop}(i) \triangleleft q]^{*} \parallel B) \xrightarrow{\tau} (\nu\, i)(i_{\text{mtr}}[i!v, \mathsf{mLoop}(i) \triangleleft q]^{*} \parallel B') \tag{85}$$

Most pairs of action can be commuted easily by PAR and SCP as they concern distinct elements of the actor system. The only non-trivial case is the pair of actions (84) and (85), which can be commuted using Lemma 8, in analogous fashion to the base case. ■