TECHNICAL REPORT

Report No. CS2013-01
Date: January 2013

Synthesising Correct Concurrent Runtime Monitors in Erlang

Adrian Francalanza
Aldrin Seychell

Department of Computer Science
University of Malta
Msida MSD 06
MALTA

Tel: +356-2340 2519
Fax: +356-2132 0539
`http://www.cs.um.edu.mt`

Synthesising Correct Concurrent Runtime Monitors in Erlang

Adrian Francalanza
Department of Computer Science
University of Malta
Msida, Malta
`adrian.francalanza@um.edu.mt`

Aldrin Seychell
Department of Computer Science
University of Malta
Msida, Malta
`asey0001@um.edu.mt`

Abstract: *In this paper, we study the correctness of automated synthesis for concurrent monitors. We adapt* sHML*, a subset of the Hennessy-Milner logic with recursion, to specify safety properties of Erlang programs. We formalise an automated translation from* sHML *formulas to Erlang monitors that analyse systems at runtime and raise a flag whenever a violation of the formula is detected, ans use this translation to build a prototype runtime verification tool. Monitor correctness for concurrent setting is then formalised, together with a technique that allows us to prove monitor correctness in stages, and use this technique to prove the correctness of our automated monitor synthesis and tool.*

Synthesising Correct Concurrent Runtime Monitors in Erlang[*]

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

Aldrin Seychell
Department of Computer Science
University of Malta
Msida, Malta
asey0001@um.edu.mt

Abstract: *In this paper, we study the correctness of automated synthesis for concurrent monitors. We adapt* sHML*, a subset of the Hennessy-Milner logic with recursion, to specify safety properties of Erlang programs. We formalise an automated translation from* sHML *formulas to Erlang monitors that analyse systems at runtime and raise a flag whenever a violation of the formula is detected, ans use this translation to build a prototype runtime verification tool. Monitor correctness for concurrent setting is then formalised, together with a technique that allows us to prove monitor correctness in stages, and use this technique to prove the correctness of our automated monitor synthesis and tool.*

1 Introduction

Runtime Verification[BFF+10] (RV), is a lightweight verification technique for determining whether the *current system run* observes a correctness property. Much of the technique's scalability comes from the fact that checking is carried out *at runtime*, allowing it to use information from the current run, such as the execution paths chosen, to optimise checking. Two crucial aspects of any RV setup are *runtime overheads*, which need to be kept to a minimum so as not to degrade system performance, and *monitor correctness*, which guarantees that the runtime checking carried out by monitors indeed corresponds to the property being checked for.

Typically, ensuring monitor correctness is non-trivial because correctness properties are specified using one formalism, *e.g.*, a high-level logic, whereas the respective monitors are described using another formalism, *e.g.*, a programming language, making it

[*]The research work disclosed in this publication is partially funded by STEPS grant contract number 7/195/2012

harder to ascertain the semantic correspondence between the two descriptions. Automatic monitor synthesis can mitigate this problem by standardising the translation from the property logic to the monitor formalism. Moreover, the regular structure of the synthesised monitors gives greater scope for a formal treatment of monitor correctness. In this work we investigate the monitor-correctness analysis of synthesised monitors, characterised as:

A system violates a property φ *iff* the monitor for φ flags a violation. (1)

Recent technological developments have made monitor correctness an even more pressing concern. In order to keep monitoring overheads low, engineers are increasingly using *concurrent monitors*[CFMP12, MJG$^+$11, KVK$^+$04, SVAR04] so as to exploit the underlying parallel architectures pervasive to today's mainstream computers.

Despite the potential for lower overheads, concurrent monitors are harder to analyse than their sequential counterparts. For instance, multiple monitor interleavings exponentially increase the computational space that need to be considered. Concurrent monitors are also susceptible to elusive errors originating from race conditions, which may result in non-deterministic monitor behaviour, deadlocks or livelocks. Whereas deadlocks may prevent monitors from flagging violations in (1) above, livelocks may lead to divergent processes that hog scarce system resources, severely affecting runtime overheads. Moreover, the possibility of non-determinism brings to the fore the implicit (stronger) requirement in correctness criteria (1): monitors *must flag* a violation whenever it occurs. Stated otherwise, in order to better utilise the underlying parallel architectures, concurrent monitors are typically required to have multiple potential interleavings which, in turn, means that for particular system runs violating a property, the respective monitors may have interleavings that correctly flag the violations and other that do not. This substantially complicates analysis for monitor correctness because all possible monitor interleaving need to be considered.

To address these issues, we propose a formal technique that alleviates the task of ascertaining the correctness of synthesised concurrent monitors by performing *three* separate (weaker) monitor-correctness checks. Since these checks are independent to one another, they can be carried out in parallel by distinct analysing entities. Alternatively, inexpensive checks 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 however, the three checks together imply the stronger correctness criteria outlined in (1).

The first monitor-correctness check of the technique is called *Violation Detectability*. It relates monitors to their respective system-property and ensures that violations are detectable (*i.e.*, there exists *at least one* monitor execution path that flags the violation) and, conversely, that the only detectable behaviour is that of violations. The may-analysis required by this check is weaker that the must-requirements of (1), but also less expensive; it may therefore also forestall other checks since the absence of a single flagging computation interleaving obviates the need to consider all potential executions. The second monitor-correctness check is called *Detection Preservation* and guarantees (observational) determinism, ensuring that monitors behave uniformly for every system run. In particular this ensures that, for a specific system run, if a monitor flags a violation for some (internal) interleaving of sub-monitor execution, it is guaranteed to also flag this violation along any other monitor interleaving. The final monitor check is called *Monitor Separability* and, as the name implies, it guarantees that the system and monitor executions can be analysed in isolation. What this really means however is that the computation of the monitor *does not affect* the execution of the monitored system *i.e.,* a form of non-interference check.

The technical development for this technique is carried out for a specific property logic and monitor language. In particular, we focus on actor-style[HBS73] monitors written in Erlang[CT09, Arm07], an established, industry strength, concurrent language for constructing fault-tolerant scalable systems. As an expository logic we consider an adaptation of SafeHML[AI99], a syntactic subset of the more expressive logic, μ-calculus, describing safety properties - which are guaranteed to be monitorable[MP90, CMP92]); the choice of our logic was, in part, motivated by the fact that the full μ-calculus was previously adapted to describe Erlang program behaviour in [Fre01], albeit for model-checking purposes.

The rest of the paper is structured as follows. Section 2 introduces the syntax and semantics of the Erlang subset considered for our study. Section 3 presents the logic and its semantics *wrt.* Erlang programs, whereas Section 4 defines the monitor synthesis of the logic. Section 5 defines monitor correctness, details our technique for proving monitor correctness and applies the technique to prove the correctness of the synthesis outlined in Section 4. Section 6 collects the technical proofs relating to the three subconditions of the technique. Finally Section 7 discusses related work and Section 8 concludes.

2 The Erlang Language

Our study focusses on a (Turing-complete) subset of the language, following [SFBE10, Fre01, Car01]; in particular, we leave out distribution, process linking and fault-trapping mechanisms. We give a formal presentation for this language subset, laying the foundations for the work in subsequent sections.

Actors, Expressions, Values and Patterns

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

Evaluation Contexts

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

Figure 1: Erlang Syntax

2.1 Syntax

We define a calculus for modelling the computation of Erlang programs. We assume denumerable sets of process identifiers $i, j, h \in \textsc{Pid}$, atoms $a, b \in \textsc{Atom}$, and variables $x, y, z \in \textsc{Var}$. The full syntax is defined in Figure 1.

An executing Erlang program is made up of a *system of actors*, $\textsc{Act}$, composed in *parallel*, $A \parallel B$, where some identifiers, *e.g.*, i, are scoped, $(i)A$. Individual actors, $i[e \triangleleft q]^m$, 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) subject to a *monitoring-modality*, $m, n \in \{\circ, \bullet, *\}$, where $\circ$, $\bullet$ and $*$ denote *monitored*, *unmonitored* and *tracing resp.*; we abuse notation and let $v : q$ denote the mailbox with v at the head of the queue and q as the tail, but also let $q : v$ denote the mailbox q appended by v at the end.

Top-level actor *expressions* typically consist of a sequence of variable binding $x_i = e_i$, terminated by an expression, e_{n+1}, *i.e.*, $x_1 = e_1, \ldots, x_n = e_n, e_{n+1}$. Expressions are expected to evaluate to values, v, and 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, $\mathsf{rcv}\, p_1 \to e_1; \ldots; p_n \to e_n\, \mathsf{end}$, case-branchings, $\mathsf{case}\, e\, \mathsf{of}\, p_1 \to e_1; \ldots; p_n \to e_n\, \mathsf{end}$, function applications, $e_1(e_2)$ and actor-spawnings, $\mathsf{spw}\, e$, amongst others. Pattern-matching consists of a list of expressions guarded by patterns, $p_i \to e_i$. Values may consist of variables, x, process ids, i, recursive functions,[1] $\mu y.\lambda x.e$, tuples $\{v_1, \ldots, v_n\}$ and lists, l, amongst others.

Expressions also specify evaluation contexts, denoted as C, also defined in Figure 1. For instance, evaluation contexts specify that an expression is only evaluated when at the top level variable binding, $x = C, e$; the other cases are also fairly standard.[2] We denote the application of a context C to an expression e as $C[e]$.

Shorthand: We write $\lambda x.e$ and d, e for $\mu y.\lambda x.e$ and $y = d, e$ *resp.* when $y \notin \mathrm{fv}(e)$. Similarly, for guarded expressions $p \to e$, we replace x in p with _ whenever $x \notin \mathrm{fv}(e)$. We write $\mu y.\lambda(x_1, \ldots x_n).e$ for $\mu y.\lambda x_1.\ldots.\lambda x_n.e$. When an actor is monitorable, we elide its modality and write $i[e \triangleleft q]$ for $i[e \triangleleft q]^{\circ}$; similarly we elide empty mailboxes, writing $i[e]$ for $i[e \triangleleft \epsilon]$. (When the surrounding context makes it absolutely clear, we sometimes also use the same abbreviation when the contents of the mailbox is not important to our discussion. *e.g.*, $i[e]$ for $i[e \triangleleft q]$ for some q.)

2.2 Semantics

We give a Labelled Transition System (LTS) semantics for systems of actors where the set of actions $\textsc{Act}_\tau$, includes a distinguished *internal* label, τ, and is defined as follows:

$$\begin{array}{rclllllll} \gamma \in \textsc{Act}_\tau & ::= & (\vec{j})i!v & \text{(bound output)} & \mid i?v & \text{(input)} & \mid \tau & \text{(internal)} \\ \alpha,\beta \in \textsc{BAct} & ::= & i!v & \text{(output)} & \mid i?v & \text{(input)} & & \end{array}$$

We write $A \xrightarrow{\gamma} B$ in lieu of $\langle A, \gamma, B\rangle \in \longrightarrow$ for the least ternary relation satisfying the rules in Figures 2, 3 and 4; we also identify a subset of *basic* actions, $\alpha,\beta \in \textsc{BAct}$ (excluding τ and bound outputs). As usual, we write *weak* transitions $A \Longrightarrow B$ and $A \overset{\gamma}{\Longrightarrow} B$, for $A \xrightarrow{\tau}{}^{*} B$ and $A \xrightarrow{\tau}{}^{*} \cdot \xrightarrow{\gamma} \cdot \xrightarrow{\tau}{}^{*} B$ *resp.* We let $s, t \in (\textsc{Act})^*$ range over lists of basic actions; the sequence of weak (basic) actions $A \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} B$, where $s = \alpha_1, \ldots, \alpha_n$ is often denoted as $A \overset{s}{\Longrightarrow} B$ (or as $A \overset{s}{\Longrightarrow}$ when B is unimportant).

The semantics of Figures 2, 3 and 4 assumes *well-formed* actor systems, whereby every actor identifier is *unique*. It is also termed to be a *tracing semantics*, whereby a distinguished *tracer* actor, identified by the modality $*$, receives messages recording the computation events of monitored actors. The tracing described by the semantics of Figure 2

[1]The preceding μy denotes the binder for function self-reference.

[2]Expressions are not allowed to evaluate under a spawn context, $\mathsf{spw}\,[-]$, which differs from the standard Erlang semantics. This however allows us to describe the spawning of a function application in lightweight fashion. Spawn in Erlang takes the module and function name of the function to be spawned, together with a list of arguments the spawned function is applied to.

$$\textsc{SndM}\ \frac{}{j[C[i!v] \triangleleft q]^{\circ} \parallel h[e \triangleleft r]^{*} \xrightarrow{i!v} j[C[v] \triangleleft q]^{\circ} \parallel h[e \triangleleft r{:}\mathrm{tr}(i!v)]^{*}}$$

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

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

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

$$\textsc{Rd1}\ \frac{\mathrm{mtch}(g, v) = e}{i[C[g] \triangleleft (v : q)]^{m} \xrightarrow{\tau} i[C[e] \triangleleft q]^{m}}$$

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

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

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

Figure 2: Erlang Semantics for Actor Systems (1)

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

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

$$\textsc{Ext}\;\frac{}{i[C[x = \mathsf{exit}, e] \triangleleft q]^m \xrightarrow{\tau} i[C[\mathsf{exit}] \triangleleft q]^m}$$

$$\textsc{Try}\;\frac{}{i[C[\mathsf{try}\, v\, \mathsf{catch}\, e] \triangleleft q]^m \xrightarrow{\tau} i[C[v] \triangleleft q]^m}$$

$$\textsc{Ctc}\;\frac{}{i[C[\mathsf{try}\,\mathsf{exit}\,\mathsf{catch}\, e] \triangleleft q]^m \xrightarrow{\tau} i[C[e] \triangleleft q]^m}$$

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

$$\textsc{Slf}\;\frac{}{i[C[\mathsf{self}] \triangleleft q]^m \xrightarrow{\tau} i[C[i] \triangleleft q]^m}$$

Figure 3: Erlang Semantics for Actor Systems (2)

$$\textsc{Scp}\;\frac{A \xrightarrow{\gamma} B}{(j)A \xrightarrow{\gamma} (j)B}\; j \notin (\mathrm{obj}(\gamma) \cup \mathrm{subj}(\gamma))$$

$$\textsc{Opn}\;\frac{A \xrightarrow{(\vec{h})i!v} B}{(j)A \xrightarrow{(j,\vec{h})i!v} B}\; i \neq j,\, j \in \mathrm{subj}((\vec{h})i!v)$$

$$\textsc{Par}\;\frac{A \xrightarrow{\gamma} A'}{A \parallel B \xrightarrow{\gamma} A' \parallel B}\; \mathrm{obj}(\gamma) \cap \mathrm{fId}(B) = \emptyset \qquad \textsc{Str}\;\frac{A \equiv A' \quad A' \xrightarrow{\gamma} B' \quad B' \equiv B}{A \xrightarrow{\gamma} B}$$

Figure 4: Erlang Semantics for Actor Systems (3)

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

$$\textsc{sCom}\frac{i \notin \mathrm{fId}(A)}{A \parallel (i)B \equiv (i)(B \parallel A)} \qquad \textsc{sSwp}\frac{}{(i)(j)A \equiv (j)(i)A}$$

$$\textsc{sCtxP}\frac{A \equiv B}{A \parallel C \equiv B \parallel C} \qquad \textsc{sCtxS}\frac{A \equiv B}{(i)A \equiv (i)B}$$

Figure 5: Structural Equivalence for Actors

closely follows the mechanism offered by the Erlang Virtual Machine (EVM) [CT09], whereby we choose to only record asynchronous message sends and mailbox message receives; see rules SndM and RcvM *resp.*; in these rules, the tracer actor receives a message in its mailbox reporting the action, defined through the function[3] defined below:

$$\mathrm{tr}(\alpha) \stackrel{\mathbf{def}}{=} \begin{cases} \{\mathsf{sd}, i, v\} & \text{if } \alpha = i!v \\ \{\mathsf{rv}, i, v\} & \text{if } \alpha = i?v \end{cases}$$

By contrast, unmonitored actor actions are not traced; see rules SndU and RcvU. Internal communication is not traced either, even when it involves monitored actions, thereby limiting tracing to external interactions (of monitored actors); see rules Com and Par. In Par, the side-condition in Par enforces the *single-receiver* property of actor systems; for instance, it prevents a transition with an action $j!v$ when actor j is part of the actor system B.

Our semantics assumes substitutions, $\sigma \in \textsc{Sub} :: \textsc{Var} \rightharpoonup \textsc{Val}$, which are partial maps from variables x_i to values v_i and denoted as $\{^{v_1,\cdots,v_n}/_{x_1,\ldots,x_n}\}$. Rules Rd1 and Rd2 (in conjunction with Snd, Rcv and Com) describe how actor communication is not atomic, as opposed to more traditional message-passing semantics [Mil89], but happens in two steps: an actor first receives a message in its mailbox and then reads it at a later stage. The mailbox reading command includes pattern-matching functionality, allowing the actor to selectively choose which messages to read first from its mailbox whenever a pattern from the pattern list is matched; when no pattern is matched, mailbox reading *blocks*. This differs from pattern matching in case branching, described by the rules Cs1 and Cs2: 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, unless it is

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

caught using Ctc.[4] Rules Rd1, Rd2, Cs1 and Cs2 make use of the auxiliary function mtch : PLst $\times$ Val $\rightarrow$ Exp$_\perp$ defined as follows:

Definition 1 (Pattern Matching) *We define* mtch *and* vmtch *as follows:*

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

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

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

Spawning, using rule Spw, launches a new actor whose scoped identifier is known only to the spawning actor, and whose monitoring modality is inherited from the spawning actor when this is either $\circ$ (monitorable) or $\bullet$ (un-monitorable); if the spawning actor is the tracer process (the $*$ modality), the monitoring modality of the new actor is set to $\bullet$ (un-monitorable). Due to the asynchronous nature of actor communication, scoped actors can send messages to the environment but *cannot* receive messages from it. Identifier scope is extended, possibly opened up to external observers, through explicit communication as in standard message-passing systems; see Opn. The functions in Def. 14 identifies values that include identifiers and thus, extending the identifiers scope. Structural equivalence, $A \equiv B$, is employed to simplify the presentation of the LTS rules; see rule Str together with Figure 5. The remaining rules in Figure 2 are fairly straightforward.

[4]In the case of exceptions, let variable bindings $x = e, d$ behave differently from standard let encodings in terms of call-by-value functions, *i.e.*, $\lambda x.d(e)$, which is why we model them separately.

Definition 2 *We define* obj, subj[5] *and* fId *as follows:*

$$\mathrm{obj}(\gamma) \overset{def}{=} \begin{cases} \emptyset & \textit{if } \gamma = \tau \\ \{i\} & \textit{if } \gamma = (\vec{h})i!v \textit{ or } i?v \end{cases}$$

$$\mathrm{subj}(\gamma) \overset{def}{=} \begin{cases} \emptyset & \textit{if } \gamma = \tau \\ (\mathrm{id_v}(v) \setminus \{\vec{h}\}) & \textit{if } \gamma = (\vec{h})i!v \\ \mathrm{id_v}(v) & \textit{if } \gamma = i?v \end{cases}$$

$$\mathrm{fId}(A) \overset{def}{=} \begin{cases} \{i\} & \textit{if } A = i[e \triangleleft q]^m \\ \mathrm{fId}(B) \cup \mathrm{fId}(C) & \textit{if } A = B \parallel C \\ \mathrm{fId}(B) \setminus \{i\} & \textit{if } A = (i)(B) \end{cases}$$

Discussion: Our tracing semantics sits at a slightly higher level of abstraction than that offered by the EVM[CT09]. For instance, trace entries typically contain more information. Moreover, in contrast to our semantics, the EVM records internal communication between monitored actors, as an output trace entry immediately followed by the corresponding input trace entry; here we describe a *sanitised* trace whereby matching trace entries are filtered out. Our tracing semantics is also more restrictive since the monitored actors are *fixed*: we model a simple (Erlang) monitor instrumentation setup whereby actors are registered to be monitored upfront before computation commences or actors that are spawned by other monitored actors. The instrumentation approach is discussed in further detail in Section 4.1.

Example 1 (Non-Deterministic behaviour) *Actors may exhibit non-deterministic behaviour as a result of internal choices, but also as a result of external choices [Mil89, HM85]. Consider the actor system:*

$$A \triangleq (j, h, i_{mon})(i[\mathsf{rcv}\ x \to \mathsf{obs}!x\,\mathsf{end} \triangleleft \epsilon]^\circ \parallel j[i!v]^\circ \parallel h[i!u]^\circ \parallel i_{mon}[e \triangleleft q]^*)$$

According to the semantics defined in Figures 2, 3 and 4, we could have the behaviour where actor j sends value v to actor i, (2)*, i reads it from its mailbox,* (3) *and subsequently outputs it to some external actor* obs*,* (4) *(while recording the external communication at the monitor's mailbox as the entry* $\{\mathsf{sd}, \mathsf{obs}, v\}$*).*

$$A \xrightarrow{\tau} (j, h, i_{mon})(i[\mathsf{rcv}\ x \to \mathsf{obs}!x\,\mathsf{end} \triangleleft v]^\circ \parallel j[v] \parallel h[i!u] \parallel i_{mon}[e \triangleleft q]^*) \tag{2}$$

$$\xrightarrow{\tau} (j, h, i_{mon})(i[\mathsf{obs}!v \triangleleft \epsilon]^\circ \parallel j[v] \parallel h[i!u] \parallel i_{mon}[e \triangleleft q]^*) \tag{3}$$

$$\xrightarrow{\mathsf{obs}!v} (j, h, i_{mon})(i[v \triangleleft \epsilon]^\circ \parallel j[v] \parallel h[i!u] \parallel i_{mon}[e \triangleleft q : \{\mathsf{sd}, \mathsf{obs}, v\}]^*) \tag{4}$$

However, if actor h sends the value u to actor i before actor j, this τ-action would

[5]The functions $\mathrm{id_v}$, $\mathrm{id_e}$ and $\mathrm{id_g}$ are defined in the appendix section.

amount to an internal choice and we observe the different external behaviour $A \overset{\mathsf{obs}!u}{\Longrightarrow} B$ *(for some B); correspondingly the monitor at B would hold the entry* $\{\mathsf{sd}, \mathsf{obs}, u\}$.

The actor system A could also receive different external inputs resulting an external choice. For instance we can derive $A \xrightarrow{i?v_1} B_1$ *or* $A \xrightarrow{i?v_2} B_2$ *where from* B_1 *we can only observe the output* $B_1 \overset{\mathsf{obs}!v_1}{\Longrightarrow} C_1$ *and, dually,* B_2 *can produce the weak external output* $B_2 \overset{\mathsf{obs}!v_2}{\Longrightarrow} C_2$ *(for some* C_1 *and* C_2*). Correspondingly, the monitor mailbox at* C_1 *would be appended by the list of entries* $\{\mathsf{rv}, i, v_1\} : \{\mathsf{sd}, \mathsf{obs}, v_1\}$ *(and dually for* C_2*).*

3 A Monitorable Logic for Open Systems

We consider an adaptation of SafeHML (SHML) [AI99], a sub-logic of the Hennessy-Milner Logic (HML) with recursion[6] for specifying correctness properties of the actor systems defined in Section 2. SHML syntactically limits specifications to *safety* properties which can be monitored at runtime[MP90, CMP92, BLS11].

3.1 Logic

Our logic assumes a denumerable set of formula variables, $X, Y \in \textsc{LVar}$, 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 directly from HML[HM85], 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 standard SHML of [AI99] by limiting necessity formulas to basic actions $\alpha, \beta \in \textsc{BAct}$. This is more of a design choice so as keep our technical development manageable. The handling of bounded output actions is well understood [MPW93] and do not pose problems to monitoring, apart from making action pattern matching cumbersome. Silent τ labels can also be monitored using minor adaptations to the monitors we defined later in Section 4; however, they would increase substantially the size of the traces recorded, unnecessarily cluttering the tracing semantics of Section 2.

[6]HML with recursion has been shown to be as expressive as the μ-calculus[Koz83].

3.2 Semantics

The semantics of our logic is defined over sets of actor systems, $S \in \mathcal{P}(\textsc{Actr})$. Traditional presentations assume variable environments, $\rho \in (\textsc{LVar} \to \mathcal{P}(\textsc{Actr}))$, mapping formula variables to sets of actor systems, together with the operation $\varphi\{^{\psi}/_{X}\}$, which substitutes free occurances of X in φ with ψ without introducing any variable capture, and the operation $\rho[X \mapsto S]$, returning the environment mapping X to S while acting as ρ for the remaining variables.

Definition 3 (Satisfaction) *For arbitrary variable environment ρ, the set of actors satisfying $\varphi \in$ sHML, denoted as $[\![\varphi]\!]\rho$, is defined by induction on the structure of φ as:*

$$\begin{aligned}
[\![\mathsf{ff}]\!]\rho &\overset{def}{=} \emptyset \\
[\![\varphi \wedge \psi]\!]\rho &\overset{def}{=} [\![\varphi]\!]\rho \cap [\![\psi]\!]\rho \\
[\![[\alpha]\varphi]\!]\rho &\overset{def}{=} \left\{A \mid B \in [\![\varphi]\!]\rho \textit{ whenever } A \overset{\alpha}{\Longrightarrow} B\right\} \\
[\![X]\!]\rho &\overset{def}{=} \rho(X) \\
[\![\mathsf{max}(X,\varphi)]\!]\rho &\overset{def}{=} \bigcup\{S \mid S \subseteq [\![\varphi]\!]\rho[X \mapsto S]\}
\end{aligned}$$

Whenever φ is a closed formula, its meaning is independent of the environment ρ and is written as $[\![\varphi]\!]$. When an actor system A satisfies a closed formula φ, we write $A \models_{s} \varphi$ in lieu of $A \in [\![\varphi]\!]$. When restricted to closed formulas, the satisfaction relation $\models_{s}$ can alternatively be specified as Definition 4 (adapted from [AI99].)

Definition 4 (Satisfiability) *A relation $\mathcal{R} \in \textsc{Actr} \times$ sHML is a satisfaction relation iff:*

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

Satisfiability, $\models_s$, is the largest satisfaction relation; we write $A \models_s \varphi$ in lieu of $(A,\varphi) \in sat$.[7]

Example 2 (Satisfiability) *SHML formulas specify safety properties* i.e., *prohibited behaviours that should not be exhibited by satisfying actors. Consider the formula*

$$\varphi_{ex} \triangleq \mathsf{max}(X,\ [\alpha][\alpha][\beta]\mathsf{ff} \wedge [\alpha]X\,) \tag{5}$$

[7]It follows from standard fixed-point theory that the implications of satisfaction relation are bi-implications for Satisfiability.

stating that a satisfying actor system cannot 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 condition needs to hold after every α action (through the subformula $[\alpha]X$); effectively the formula states that any sequence of external actions α that is greater than two *cannot be followed by an external action β.*

An actor system A_1 exhibiting (just) the external behaviour $A_1 \overset{\alpha\beta}{\Longrightarrow} A_1'$ for arbitrary A_1' satifies φ_{ex}, just as an actor system A_2 with (infinite) behaviour $A_2 \overset{\alpha}{\Longrightarrow} A_2$. An actor system A_3 with an external action trace $A_3 \overset{\alpha\alpha\beta}{\Longrightarrow} A_3'$ cannot *satify φ_{ex}; if, however, (through some internal choice) it exhibits an alternate external behaviour such as $A_3 \overset{\beta}{\Longrightarrow} A_3''$ we could not observe any violation since the external trace β is allowed by φ_{ex}.*

Because of the potenially non-deterministic nature of actors which may violate a property along one execution trace but satify it along another, we define a *violation relation*, Def. 5, characterising actors that violate a property along a specific violating trace.

Definition 5 (Violation) *The violation relation, denoted as $\models_v$, is the least relation of the form* ($\textsc{Actr} \times \textsc{Act}^* \times \text{sHML}$) *satisfying the following rules:*[8]

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

Example 3 (Violation) *Recall the safety formula φ_{ex} defined in* (5). *Actor A_3, from Ex. 2, together with the witness violating trace $\alpha\alpha\beta$ violate φ_{ex},* i.e., $(A_3, \alpha\alpha\beta) \models_v \varphi_{ex}$. *However, A_3 together with trace β do not violate φ_{ex},* i.e., $(A_3,\beta) \not\models_v \varphi_{ex}$. *Def 5 relates a violating trace with an actor* only when *that trace leads the actor to a violation. For instance, if A_3 cannot perform the trace $\alpha\alpha\alpha\beta$ then we have$(A_3, \alpha\alpha\alpha\beta) \not\models_v \varphi_{ex}$ according to Def 5, even though the trace is prohibited by φ_{ex}. Moreover, a violating trace may lead an actor system to a violation before the end of the trace is reached; for instance, we can show that $(A_3, \alpha\alpha\beta\alpha) \models_v \varphi_{ex}$ according to Def 5.*

Although it is more convenient to work with Def 5 when reasoning about runtime monitors, we still have to show that it corresponds to the dual of Def. 4.

Theorem 1 (Correspondence) $\exists s.A, s \models_v \varphi \quad \textit{iff} \quad A \not\models \varphi$

[8]We write $A, s \models_v \varphi$ in lieu of $(A, s, \varphi) \in \models_v$. It also follows from standard fixed-point theory that the constrainst of the violation relation are biimplications.

Proof For the *if* case we prove the contrapositive, namely that

$$\forall s.A, s \not\models_{\mathrm{v}} \varphi \quad \text{implies} \quad A \models_{\mathrm{s}} \varphi$$

by showing that the relation

$$\mathcal{R} = \{(\gamma, \varphi) \mid \forall s.A, s \not\models_{\mathrm{v}} \varphi\}$$

is a satisfaction relation. The proof proceeds by induction on the structure of φ:

ff**:** Contradiction, because from Def. 5 we know we can never have $A, s \not\models_{\mathrm{v}} \mathsf{ff}$ for any s.

$\varphi \wedge \psi$**:** From the definition of $\mathcal{R}$ we know that $\forall s.A, s \not\models_{\mathrm{v}} \varphi \wedge \psi$ and, by Def. 5, this implies that $\forall s.(A, s \not\models_{\mathrm{v}} \varphi$ and $A, s \not\models_{\mathrm{v}} \psi)$. Distributing the universal quantification yields

$$\forall s.A, s \not\models_{\mathrm{v}} \varphi \tag{6}$$
$$\forall s.A, s \not\models_{\mathrm{v}} \psi \tag{7}$$

and by the definition of $\mathcal{R}$, (6) and (7) we obtain $(A, \varphi) \in \mathcal{R}$ and $(A, \psi) \in \mathcal{R}$, as required for satisfiability relations by Def. 4.

$[\alpha]\varphi$**:** From the definition of $\mathcal{R}$ we know that $\forall s.A, s \not\models_{\mathrm{v}} [\alpha]\varphi$. In particular, for all $s = \alpha t$, we know that $\forall t.A, \alpha t \not\models_{\mathrm{v}} [\alpha]\varphi$. From Def. 5 it must be the case that whenever $A \overset{\alpha}{\Longrightarrow} B$ we have that $\forall t.B, t \not\models_{\mathrm{v}} \varphi$, which in turn implies that $(B, \varphi) \in \mathcal{R}$ (from the definition of $\mathcal{R}$); this is the implication required by Def. 4.

$\mathsf{max}(X, \varphi)$**:** From the definition of $\mathcal{R}$ we know that $\forall s.A, s \not\models_{\mathrm{v}} \mathsf{max}(X, \varphi)$. For each s, from $A, s \not\models_{\mathrm{v}} \mathsf{max}(X, \varphi)$ and Def. 5 we obtain $A, s \not\models_{\mathrm{v}} \varphi\{^{\mathsf{max}(X,\varphi)}/_X\}$ and thus, from the definition of $\mathcal{R}$, we conclude that $(A, \varphi\{^{\mathsf{max}(X,\varphi)}/_X\}) \in \mathcal{R}$ as required by Def. 4.

For the *only-if* case we prove

$$\exists s.A, s \models_{\mathrm{v}} \varphi \quad \text{implies} \quad A \not\models \varphi$$

by rule induction on $A, s \models_{\mathrm{v}} \varphi$. Note that $A \not\models \varphi$ means that there does not exist *any* satisfiability relation including the pair (A, φ).

$A, s \models_{\mathrm{v}} \mathsf{ff}$**:** Immediate by Def. 4.

$A, s \models_{\mathrm{v}} \varphi \wedge \psi$ **because** $A, s \models_{\mathrm{v}} \varphi$**:** By $A, s \models_{\mathrm{v}} \varphi$ and I.H. we obtain $A \not\models \varphi$. As a result, we conclude that $A \not\models \varphi \wedge \psi$.

$A, s \models_{\mathrm{v}} \varphi \wedge \psi$ **because** $A, s \models_{\mathrm{v}} \psi$**:** Similar.

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

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

Def. 5 allows us to show that SHML is a safety language as defined in [MP90, CMP92]; this relies on the standard notion of trace prefixes *i.e.*, $s \leq t$ iff $\exists.s'$ such that $t = ss'$.

Theorem 2 (Safety Relation) *$A, s \models_v \varphi$ and $s \leq t$ implies $A, t \models_v \varphi$*

Proof By Rule Induction on $A, s \models_v \varphi$:

$A, s \models_v \mathsf{ff}$: Immediate.

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

$A, s \models_v \varphi \wedge \psi$ because $A, s \models_v \psi$: Similar.

$A, s \models_v [\alpha]\varphi$ because $s = \alpha s', A \overset{\alpha}{\Longrightarrow} B$ and $B, s' \models_v \varphi$: From $s = \alpha s'$ and $s \leq t$ we know

$$t = \alpha t' \tag{8}$$
$$s' \leq t' \tag{9}$$

By (9), $B, s' \models_v \varphi$ and I.H. we obtain $B, t' \models_v \varphi$ and by $A \overset{\alpha}{\Longrightarrow} B$ and (8) we derive $A, t \models_v [\alpha]\varphi$.

$A, s \models_v \mathsf{max}(X, \varphi)$ because $A, s \models_v \varphi\{^{\mathsf{max}(X,\varphi)}/_X\}$: By $A, s \models_v \varphi\{^{\mathsf{max}(X,\varphi)}/_X\}$, $s \leq t$ and I.H. we obtain $A, t \models_v \varphi\{^{\mathsf{max}(X,\varphi)}/_X\}$ which implies $A, t \models_v \mathsf{max}(X, \varphi)$ by the same rule. □

4 Runtime Verifying SHML properties

We define a translation from SHML formulas of Section 3 to programs from Section 2 that act as *monitors*, analysing the behaviour of a system and flagging an alert whenever the property denoted by the respective formula is violated by the current execution of the system. Apart from the language translated into, our monitors differ from tests, as defined in [AI99], in a number of ways. Firstly, they monitor systems *asynchronously* and act on traces. By contrast, tests interact with the system directly (potentially inducing certain system behaviour). More importantly, however, we impose *stronger detection requirements* on monitors as opposed to those defined for test in [AI99]: whereas tests are required to have *one* possible execution that detects property violations, we require monitors to flag violations *whenever they occur*.[9]

In terms of our logic, the stronger monitoring requirements are particularly pertinent to conjunction formulas, $\varphi_1 \wedge \varphi_2$, whereby a monitor needs to ensure that neither φ_1 nor φ_2 are violated. Concurrent monitoring for the subformula φ_1 and φ_2 provides a natural translation in terms of the language presented in Section 2, bringing with it the advantages discussed the Introduction.

Conjunction formulas arise frequently in cases where systems are subject to numerous requirements from distinct parties. Although the various requirements $\varphi_1, \ldots, \varphi_n$ can sometimes be consolidated into a formula without top-level conjunction, it is often convenient to just monitor for an aggregate formula of the form $\varphi_1 \wedge \ldots \wedge \varphi_n$, which would translate to n concurrent monitors each analysing the system trace independently.

Example 4 (Conjunction Formulas) *Consider the formula the two independent formulas*

$$\varphi_{no\_dup\_ans} \triangleq [\alpha_{call}](\mathsf{max}(X, [\alpha_{ans}][\alpha_{ans}]\mathsf{ff} \wedge [\alpha_{ans}][\alpha_{call}]X)) \tag{10}$$

$$\varphi_{react\_ans} \triangleq \mathsf{max}(Y, [\alpha_{ans}]\mathsf{ff} \wedge [\alpha_{call}][\alpha_{ans}]Y) \tag{11}$$

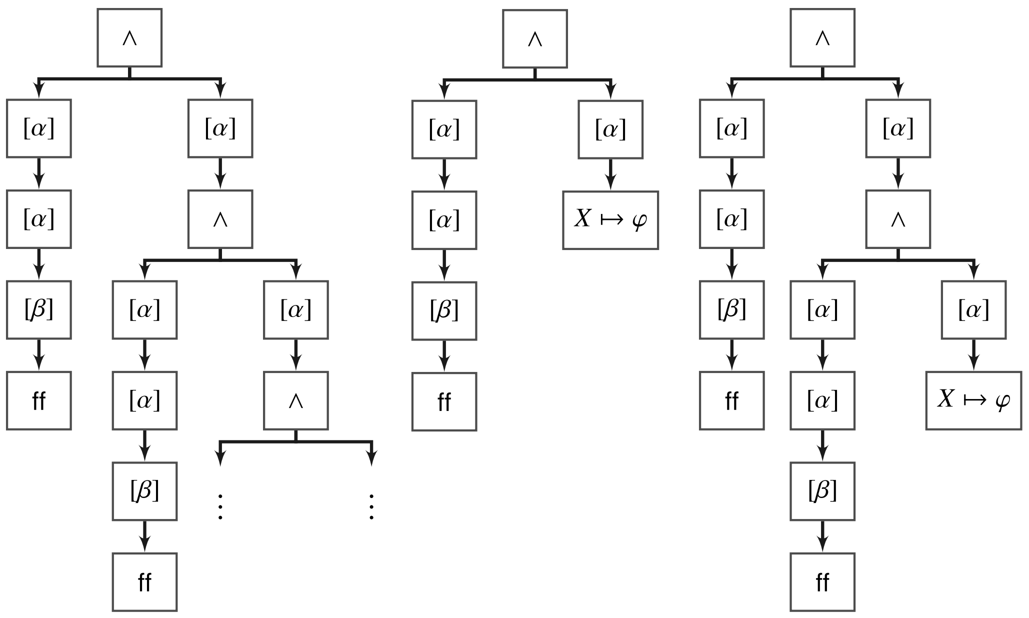
Formula $\varphi_{no\_dup\_ans}$ requires that call actions α_{call} are at most serviced by a single *answer action α_{ans}, whereas φ_{react_ans} requires that answer actions are only produced* in response to *call actions. Even though it is possible to rephrase the conjunction of the two formulas as a single formula without a top-level conjunction, it is more straightforward to monitor for $\varphi_{no\_dup\_ans} \wedge \varphi_{react\_ans}$ using two parallel monitors, one for each subformula.*[10]

Moreover, multiple conjunctions arise indirectly when they are used under fixpoint operators in formulas.

Example 5 (Conjuctions and Fixpoints) *Recall φ_{ex}, defined in* (5) *in Ex. 2. Semantically, the formula represents the infinite tree depicted in Figure 6(a), comprising of*

[9]There are other discrepancies such as the fact that monitors are required to keep overheads to a minimum, whereas tests typically are not.

[10]In cases where distinct parties are responsible for each subformula, keeping the subformulas separate may also facilitate maintainability and increases separation of concerns.



(a) Denotation of φ_{ex} defined in (5)

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

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

Figure 6: Monitor Combinator generation

an infinite number of conjunctions. Although in practice, we cannot generate an infinite number of concurrent monitors, φ_{ex} will translate into a number of concurrent monitors.

Monitor synthesis ($[\![-]\!]^{\mathbf{m}} : \text{sHML} \rightarrow \textsc{Exp}$) described in Def. 6, returns functions that are parametrised by a map (encoded as a list of tuples) from formula variables to other synthesised monitors of the same form; the map encodes the respective variable bindings in a formula and is used for formula recursive unrolling. We only limit monitoring to closed, guarded[11] sHML formulas: monitor instrumentation, performed through the function M defined below, spawns the synthesised function applied to the empty map, nil, and then acts as a message forwarder to the spawned process, mLoop, for the trace it receives through the tracing semantics of Section 2.2.

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

Definition 6 (Monitors)

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

[11]In guarded SHML formulas, formula variables appear only as a subformula of a necessity formula.

Auxiliary Function definitions and meta-operators:

$$\mathit{fork} \overset{def}{=} \mu y_{rec}.\lambda(x_{pid1}, x_{pid2}).\mathsf{rcv}\ z \to (x_{pid1}!z,\ x_{pid2}!z)\ \mathsf{end},\ y_{rec}(x_{pid1}, x_{pid2})$$

$$\mathit{lookUp} \overset{def}{=} \begin{cases} \mu y_{rec}.\lambda(x_{var}, x_{map}).\mathsf{case}\ x_{map}\ \mathsf{of} \\ \qquad (\{x_{var}, z_{mon}\} : \_) \to z_{mon} \\ \qquad \_ : z_{tl} \to y_{rec}(x_{var}, z_{tl}) \\ \qquad \mathit{nil} \to \mathit{exit} \\ \quad \mathsf{end} \end{cases}$$

In Def. 6, the monitor translation for ff immediately reports a violation by sending the message fail to a predefined actor sup handling the violation. The translation of $\varphi_1 \wedge \varphi_2$ consists in spawning the respective monitors for φ_1 and φ_2 and then forwarding the 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 (see meta-function tr($-$)), but terminates if the trace message does not correspond to α.

The translations of $\mathsf{max}(X, \varphi)$ and X are best understood together: the monitor for $\mathsf{max}(X, \varphi)$ behaves like that for φ, under the extended map where X is mapped to the monitor for φ, effectively modelling the formula unrolling $\varphi\{^{\mathsf{max}(X,\varphi)}/_X\}$ from Def. 4; the monitor for X retrieves the monitor translation for the formula it is bound to in the map through using function lookUp, and behaves like this monitor. The assumption that synthesised formulas is closed guarantees that map entries are always found by lookUp, whereas the assumption regarding guarded formulas guarantees that monitor executions do not produce infinite bound-variable expansions.

Example 6 (Monitor Translation) *The generated monitor for φ_{ex}, defined in* (5), *is*

```
 Monitor = m_max('X',
                m_and(m_nec_send(alphaTo, alphaMsg
                            m_nec_send(alphaTo, alphaMsg,
                                   m_nec_send(betaTo, betaMsg, m_fls()))),
                       m_nec_send(alphaTo, betaMsg, m_var('X')))),
M(Monitor).
```

where the referenced functions are defined as

```
m_fls() → fun (_Mappings) → sup ! fail end.

m_nec_recv(Receiver, Message, Phi) →
  fun (Mappings) →
     receive
       {action, {recv, Receiver, Message}} → Phi(Mappings);
```

```
        {action, _} → ok
      end
   end.

m_nec_send(To, Message, Phi) →
   fun (Mappings) →
      receive
        {action, {send, _SentFrom, To, Message}} → Phi(Mappings);
        {action, _} → ok
      end
   end.

m_and(Phi1, Phi2) →
   fun (Mappings) →
      A = spawn(fun() → Phi1(Mappings)),
      B = spawn(fun() → Phi2(Mappings)),
      fork(A, B)
   end.

fork(A, B) → receive Msg →
                        A ! Msg,
                        B ! Msg
              end,
              fork(A, B).

m_max(Var, Phi) → fun (Mappings) →
                       Phi([{Var, Phi} | Mappings])
                     end.

m_var(Var) → fun(Mappings) →
                  Monitor = lookUp(Var, Mappings),
                  Monitor(Mappings)
              end.

lookUp(Var, Mappings) →
      case Mappings of
           [{Var, Monitor} | _Other] → Monitor;
           [ _ | Tail] → lookUp(Var, Tail);
           [] → exit
      end.
```

The code in this example first creates the structure of functions that together represent the property that is being verified. The functions called `m_form` *are the direct Erlang translation from the Def. 6. The process* M *is responsible to start up the monitor for the property generated through the call to these monitor functions and forwarding the trace messages to these monitors. When spawning the monitor for the property, the property is passed an empty environment. The function for the maximum fixpoint extends this environment with a reference to the definition of the property itself and con-*

tinues to execute the remaining property. When a free instance of the variable X is encountered, the monitor definition is retrieved from the environment using the function `lookUp(Var, Mappings)` *and continues executing as the result of this function. Thus, the actor executing the property for X evolves to an $\wedge$ node while spawning two new actors, one checking for the subformula* $[\alpha][\alpha][\beta]\mathsf{ff}$, *and another one checking for the formula* $[\alpha]X$. *This is depicted in Figure 6(c). Whenever the monitor for the variable X is encountered following a sequence of received traces, the same unfolding is repeated by the rightmost actor in Figure 6(c), thereby increasing the monitor overhead incrementally, as required by the runtime behaviour of the system being monitored.*

4.1 Tool Implementation

The monitor generation procedure given in Def. 6 acts as a basis for a prototype implementation of a tool automating monitor synthesis of sHML formula into actual Erlang monitors. The tool can be downloaded from [Sey13]. Several design decisions were taken while implementing the tool so as to integrate and instrument the system with the generated monitors.

As discussed earlier, we make use of the tracing mechanism that is available in the EVM to gather the trace of a particular process. However, when monitoring a system, we need to collect traces from a collection of processes that are communicating with each other and also with external processes. In Erlang, the tracing functionality is enabled using the `trace/2` built-in function where the first argument is the process identifier to trace and the second parameter is a list of flags to specify the type of tracing required. One of the flags that is used in our tool is called `set_on_spawn` that enables tracing on newly spawned processes. This flag's behaviour matches the language semantics in the rule Spw since if the process spawning a new child process is traceable, the new child process is also traceable.

Another challenge that arose when instrumenting the system was that of the approach to use so as to *initialise* the monitoring tool. The ideal way to initialise the monitor is to enable the monitoring while the system is already executing. In order to be able to do so, a list of all the PIDs of the processes in the system being monitored is required. In order to avoid the race-condition where new processes are spawned while enabling tracing, it would be necessary to suspend the execution of the system through some instrumentation mechanism. Instead, our tool provides the monitor with the function that needs to be called when starting up the system. The monitor spawns a special *initiator* process that blocks its execution immediately using the *receive* construct. The monitor then enables the tracing for this newly spawned process with the setting of the `set_on_spawn` flag, after which it notifies the process that it can resume its execution and start up the system. The result of this setup is that all processes spawned by the system are monitored as well.[12]

After instrumenting the system to receive trace messages for its actions, the actual monitoring needs to take place. The user specifies the correctness property using the sHML logic introduced in Section 3. Hence, a parser is also provided in order to translate the property in sHML to the actual synthesis presented in this study. This synthesis is used after the instrumentation is in place in order to monitor the system for violating traces.

5 Correctness

We now discuss how, in the model of the runtime system, monitors are instrumented to execute *wrt.* monitored systems, Section 5.1, how correct monitor behaviour is specified in such a setting, Section 5.2.

5.1 System Instrumentation

We limit monitoring to *monitorable* systems, Def. 7, whereby all actors of a system are monitorable. This guarantees that all the basic actions produced by the system are recorded as trace entries at the monitor's mailbox; note that due to the asynchronous nature of communication, even scoped actors can generate visible actions by sending messages to other actors in the environment. An important property for our monitoring setup is that monitorable systems are closed *wrt.* weak sequences of basic actions; this is essential for our instrumentation to remain valid over the course of a monitored execution.

Definition 7 (Monitorable Systems) *An actor system A is said to be* monitorable *iff*

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

Lemma 1 *A is monitorable and $A \parallel i[e \triangleleft q]^* \overset{s}{\Rightarrow} B \parallel i[e' \triangleleft q']^*$ implies B is monitorable*

Proof By induction on the length of $A \overset{s}{\Longrightarrow} B$ and then by rule induction for the inductive case. Note that in rule Spw, the spawned process inherits the monitoring modality of the process spawning it. Since in a monitorable system, all actors are monitored, new spawned actors will be set with a monitorable modality as well. $\square$

[12]The blocking is necessary so as to prevent the system from spawning new processes that are not monitored and resulting in the same race condition as before.

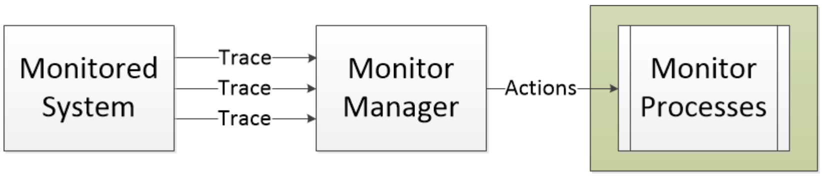


Figure 7: A high level architecture of the monitor

Our correctness results concern (unmonitored) *basic* systems, Def. 8, that are instrumented to execute in parallel with a synthesised monitor from Def. 6 as depicted in Figure 7. Our instrumentation is defined in terms of the operation $\lceil - \rceil$, Def. 9, converting basic systems into monitorable ones; see Lemma 2. Importantly, instrumentation does not affect the external visible behaviour of a basic system; see Lemma 3.

Definition 8 (Basic Systems) *A system is said to be* basic *if it does not contain a monitor actor*[13] *and each actor is unmonitored. Formally, basic systems are characterised by the predicate*

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

Definition 9 (Instrumentation) *Instrumentation, denoted as* $\lceil - \rceil :: \textsc{Actr} \rightarrow \textsc{Actr}$ *is defined inductively as:*

$$\begin{aligned} \lceil i[e \triangleleft q]^m \rceil &\overset{def}{=} i[e \triangleleft q]^\circ \\ \lceil B \parallel C \rceil &\overset{def}{=} \lceil B \rceil \parallel \lceil C \rceil \\ \lceil (i)B \rceil &\overset{def}{=} (i)\lceil B \rceil \end{aligned}$$

Lemma 2 *If A is a basic system then* $\lceil A \rceil$ *is monitorable.*

Proof By induction on the structure of A. □

Lemma 3 *For all basic actors A where* $i_{mon} \notin \mathrm{fn}(A)$*:*

- $A \xrightarrow{\alpha} B$ *iff* $(i_{mon})(\lceil A \rceil \parallel i_{mon}[e \triangleleft q]^m) \xrightarrow{\alpha} (i_{mon})(\lceil B \rceil \parallel i_{mon}[e \triangleleft q : \mathrm{tr}(\alpha)]^m)$
- $A \xrightarrow{\tau} B$ *iff* $(i_{mon})(\lceil A \rceil \parallel i_{mon}[e \triangleleft q]^m) \xrightarrow{\tau} (i_{mon})(\lceil B \rceil \parallel i_{mon}[e \triangleleft q]^m)$

Proof By rule induction. □

[13] A monitor actor is identified by the tracing modality $*$.

5.2 Monitor Correctness

Specifications relating to monitor correctness are complicated by two factors: system *non-determinism* and system *divergence*. We deal with the non-determinism of the monitored system by requiring proper monitor behaviour only when the system performs a violating execution; this can be expressed through the violation relation formalised in Def. 5.

System divergence however complicates statements relating to any deterministic detection behaviour expected from the synthesised monitor should the monitored system perform a violating execution: although we would like to require that the monitor *must* always detect and flag a violation whenever it occurs, a divergent system executing in parallel with it can postpone indefinitely this behaviour. Even though we have no guarantees on the behaviour of the system being monitored, the EVM provides fairness guarantees[CT09] for actor executions. In such cases, it therefore suffices to require a weaker property from our synthesised monitors, reminiscent of the condition in fair/should-testing[RV07].

Definition 10 (Should-α) $A \Downarrow_{\alpha} \overset{\mathit{def}}{=} (A \Longrightarrow B \textit{ implies } B \overset{\alpha}{\Longrightarrow})$

Definition 11 (Correctness) *$e \in$ Exp is a correct monitor for $\varphi \in$ sHML iff for any basic actors $A \in$ Actr, $i \notin \mathrm{fId}(A)$, and execution traces $s \in \mathrm{Act}^* \setminus \{\mathsf{sup!fail}\}$:*

$$(i)(\lceil A \rceil \parallel i[e]^*) \overset{s}{\Rightarrow} B \quad \textit{implies} \quad (A, s \models_v \varphi \quad \textit{iff} \quad B \Downarrow_{\mathsf{sup!fail}})$$

6 Proving Correctness

Def. 11, defined in the previous section, allows us to state the main result of the paper, Theorem 3.

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

Proving Theorem 3 directly can be an arduous task because it requires reasoning about all the possible execution paths of the monitored system in parallel with the instrumented monitor. Instead we propose a technique that teases apart three sub-properties about our synthesised monitors from the correctness property of Theorem 3. As argued in the Introduction, these sub-properties can be checked by distinct analysing entities or used as vetting checks so as to abort early our correctness analysis. More importantly, together, these weaker properties imply the stronger property of Theorem 3.

The first sub-property is *Violation Detection*, Lemma 4, guaranteeing that for every violating trace s of formula φ there exists an execution by the respective synthesised monitor that detects the violation. This property is easier to verify than Theorem 3 because it requires us to consider the execution of the monitor in isolation and, more importantly, requires us to verify the *existence of a single execution path* that detects the violation; concurrent monitors typically have multiple execution paths.

Lemma 4 (Violation Detection) *For basic* $A \in \textsc{Actr}$ *and* $i_{mon} \notin \mathrm{fId}(A)$*,* $A \overset{s}{\Longrightarrow}$ *implies:*

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

Unlike Lemma 4, the next property, called Detection Preservation (Lemma 5), is not concerned with relating detections to actual violations; 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 reduction *wrt.* their mailbox contents (as it turns out to be the case for the monitors given in Def. 6) this condition suffices to guarantee that the monitor will deterministically detect violations. More generally, however, in a setting that guarantees fair actor executions, Lemma 5 ensures that detection will always eventually occur, even when monitors execute in parallel with other, potentially divergent, systems.

Lemma 5 (Detection Preservation) *For all* $\varphi \in \mathrm{sHML}$*,* $q \in \textsc{Val}^*$

$$i_{mon}[\mathsf{M}(\varphi) \triangleleft q]^* \xRightarrow{\mathsf{sup!fail}} \textit{ and } i_{mon}[\mathsf{M}(\varphi) \triangleleft q]^* \Longrightarrow B \;\textit{ implies }\; B \xRightarrow{\mathsf{sup!fail}}$$

The third sub-property we require is Separability, Lemma 6, 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 6 (Monitor Separability) *For all basic actors* $A \in \textsc{Actr}, i_{mon} \notin \mathrm{fId}(A), \varphi \in \mathrm{sHML}, s \in \textsc{Act}^* \setminus \{\mathsf{sup!fail}\}$*,*

$$(i_{mon})(\lceil A \rceil \parallel i_{mon}[\mathsf{M}(\varphi)]^*) \overset{s}{\Rightarrow} B \textit{ implies } \exists B', B'' \textit{s.t.} \begin{cases} B \equiv (i_{mon})(B' \parallel B'') \\ A \overset{s}{\Rightarrow} A' \textit{ s.t. } B' = \lceil A' \rceil \\ i_{mon}[\mathsf{M}(\varphi) \triangleleft \mathrm{tr}(s)]^* \Rightarrow B'' \end{cases}$$

These three properties suffice to show monitor correctness.

Recall Theorem 3 (Monitor Correctness). *For all* $\varphi \in$ sHML, *basic actors* $A \in$ ACTR, $i_{mon} \notin \mathrm{fn}(A)$, $s \in$ ACT$^* \setminus \{\mathsf{sup!fail}\}$, *whenever* $(i_{mon})(\lceil A \rceil \parallel i_{mon}[M(\varphi)]^*) \overset{s}{\Rightarrow} B$ *then:*

$$A, s \models_v \varphi \quad \textit{iff} \quad B \Downarrow_{\mathsf{sup!fail}}$$

Proof For the *only-if* case, we assume

$$(i_{mon})(A \parallel i_{mon}[\mathsf{M}(\varphi)]^*) \overset{s}{\Rightarrow} B \tag{12}$$

$$A, s \models_v \varphi \tag{13}$$

From Def. 10 we can further assume

$$B \Rightarrow B' \tag{14}$$

and then be required to prove that $B' \xRightarrow{\mathsf{sup!fail}}$. From (12) (14) and Lemma 6 we know

$$\exists B'', B''' s.t.\ B' = (i_{mon})(B'' \parallel B''') \tag{15}$$

$$A \overset{s}{\Longrightarrow} A' \text{ for some } A' \text{ where } \lceil A' \rceil = B'' \tag{16}$$

$$i_{mon}[\mathsf{M}(\varphi) \triangleleft \mathrm{tr}(s)]^* \Longrightarrow B''' \tag{17}$$

From (16), (13) and Lemma 4 we obtain

$$i_{mon}[\mathsf{M}(\varphi) \triangleleft \mathrm{tr}(s)]^* \xRightarrow{\mathsf{sup!fail}} \tag{18}$$

and from (17), (18) and Lemma 5 we obtain $B''' \xRightarrow{\mathsf{sup!fail}}$ and hence, by (15), and by rules PAR and SCP, we obtain $B' \xRightarrow{\mathsf{sup!fail}}$, as required.

For the *if* case we know:

$$(i_{mon})(\lceil A \rceil \parallel i_{mon}[\mathsf{M}(\varphi)]^*) \overset{s}{\Longrightarrow} B \tag{19}$$

$$B \Downarrow_{\mathsf{sup!fail}} \tag{20}$$

and have to prove that $A, s \models_v \varphi$. From (20) we know $B \xRightarrow{\mathsf{sup!fail}}$ which, together with (19) implies

$$\exists B' \ s.t.\ (i_{mon})(\lceil A \rceil \parallel i_{mon}[\mathsf{M}(\varphi)]^*) \overset{s}{\Longrightarrow} B' \xrightarrow{\mathsf{sup!fail}} \tag{21}$$

From Lemma 6 and (21) we obtain

$$\exists B'', B''' s.t.\ B' = (i_{mon})(B'' \parallel B''') \tag{22}$$

$$A \overset{s}{\Longrightarrow} A' \text{ for some } A' \text{ where } \lceil A' \rceil = B'' \tag{23}$$

$$i_{mon}[\mathsf{M}(\varphi) \triangleleft \text{tr}(s)]^{*} \Longrightarrow B''' \tag{24}$$

From (21), (22) and the freshness of $\mathsf{sup!fail}$ to A we deduce that $B'' \xrightarrow{\mathsf{sup!fail}}$, and subsequently, by (24), we obtain

$$i_{mon}[\mathsf{M}(\varphi) \triangleleft \text{tr}(s)]^{*} \overset{\mathsf{sup!fail}}{\Longrightarrow} \tag{25}$$

Finally, by (23), (25) and Lemma 4 we obtain $A, s \models_{\text{v}} \varphi$, as required. □

6.1 Proofs of the Sub-Properties

In this section we detail the proofs of the individual sub-properties identified in Sec. 6; on first reading, the reader may safely skip this section.

6.1.1 Violation Detection

The first sub-property we consider is Lemma 4, Violation Detection. One of the main Lemmas used in the proof, namely Lemma 10, relies on an encoding of formula substitutions, $\theta :: \text{LV}\textsc{ar} \rightharpoonup \text{sHML}$, 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 Def. 6. Formula substitutions are denoted as lists of individual substitutions, $\{\varphi_1/X_1\} \ldots \{\varphi_n/X_n\}$ where every X_i is distinct, and empty substitutions are denoted as ϵ.

Definition 12 (Formula Substitution Encoding)

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

We can show that our monitor lookup function of Def. 6 models variable substitution, Lemma 7. 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, which justifies the abuse of notation in subsequent proofs that assume a single possible representation of a formula substitution.

Lemma 7 *$\theta(X) = \varphi$ implies $i[\mathsf{lookUp}('X', \mathrm{enc}(\theta)) \triangleleft q]^m \Longrightarrow i[[\![\varphi]\!]^{\mathbf{m}} \triangleleft q]^m$*

Proof By induction on the number of mappings $\{\varphi_1/X_1\} \ldots \{\varphi_n/X_n\}$ in θ. $\square$

Lemma 8 *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. $\square$

In one direction, Lemma 4 relies on Lemma 10 in order to establish the correspondence between violations and the possibility of detections; this lemma, in turn, uses Lemma 9 which relates possible detections by monitors synthesised from subformulas to possible detections by monitors synthesised from conjunctions using these subformulas.

Lemma 9 *For an arbitrary θ, $(i)(i_{mon}[\mathsf{mLoop}(j_1) \triangleleft \mathrm{tr}(s)]^* \parallel i[[\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta))]^\bullet) \overset{\mathsf{sup!fail}}{\Longrightarrow}$ implies $(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^* \parallel i[[\![\varphi_1 \wedge \varphi_2]\!]^{\mathbf{m}}(\mathrm{enc}(\theta))]^\bullet) \overset{\mathsf{sup!fail}}{\Longrightarrow}$ for any $\varphi_2 \in$ sHML.*

Proof By Def. 6 we know that

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

We then prove by induction the structure of s that

$$\begin{aligned}&(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^\bullet \parallel i[[\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^\bullet) \overset{\mathsf{sup!fail}}{\Longrightarrow} \quad \text{implies} \\ &(i)\left(\begin{array}{l} i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^* \parallel \\ (j,h)(i[\mathsf{fork}(j,h)]^\bullet \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^\bullet \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^\bullet) \end{array}\right) \overset{\mathsf{sup!fail}}{\Longrightarrow}\end{aligned}$$

$s = \epsilon$**:** From Def. 6 we know that $i_{mon}[\mathsf{mLoop}(i) \triangleleft \epsilon]^\bullet$ in the system $(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^\bullet \parallel i[[\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^\bullet)$ is stuck (after one τ-transition). Thus it must have been the case that $i[[\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^\bullet \overset{\mathsf{sup!fail}}{\Longrightarrow}$. The result thus follows by repeated applications of rules Par and Scp.

$s = \alpha t$**:** We have two subcases to consider. Either $i[[\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^\bullet \overset{\mathsf{sup!fail}}{\Longrightarrow}$ immediately, in which case the result follows analogously to the previous case. Alternatively, we have

$$\begin{aligned}&(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(\alpha) : \mathrm{tr}(t)]^* \parallel i[[\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^\bullet) \Longrightarrow \\ &\quad (i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(t)]^* \parallel i[[\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q : \mathrm{tr}(\alpha)]^\bullet) \overset{\mathsf{sup!fail}}{\Longrightarrow}\end{aligned} \tag{26}$$

By (26) and I.H. we obtain

$$(i)\left(\begin{array}{l} i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(t)]^{*} \parallel \\ (j,h)\left(\begin{array}{l} i[\mathsf{fork}(j,h)]^{\bullet} \parallel \\ j[[\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q:\mathrm{tr}(\alpha)]^{\bullet} \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q:\mathrm{tr}(\alpha)]^{\bullet} \end{array}\right) \end{array}\right) \overset{\mathsf{sup!fail}}{\Longrightarrow}$$

and the result follows from the fact that

$$(i)\left(\begin{array}{l} i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel \\ (j,h)(i[\mathsf{fork}(j,h)]^{\bullet} \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^{\bullet} \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^{\bullet}) \end{array}\right) \Longrightarrow$$
$$(i)\left(\begin{array}{l} i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(t)]^{*} \parallel \\ (j,h)\left(\begin{array}{l} i[\mathsf{fork}(j,h)]^{\bullet} \parallel \\ j[[\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q:\mathrm{tr}(\alpha)]^{\bullet} \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q:\mathrm{tr}(\alpha)]^{\bullet} \end{array}\right) \end{array}\right)$$

□

Lemma 10 *If* $A, s \models_v \varphi\theta$ *and* $l_{env} = \mathrm{enc}(\theta)$ *then*

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

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

$A, s \models_{\mathrm{v}} \mathsf{ff}\theta$**:** Using Def. 6 for the definition of $[\![\mathsf{ff}]\!]^{\mathbf{m}}$ and the rule App (and Par and Scp), we have

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

The result follows trivially, since the process i can transition with a sup!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

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

The result thus follows from Lemma 9, which allows us to conclude that

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

$A, s \models_{\mathrm{v}} (\varphi_1 \wedge \varphi_2)\theta$ **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 Def. 6

for the property $[\alpha]\varphi$ we derive (27), by executing mLoop— see Def. 6 — we obtain (28), and then by rule RD1 we derive (29) below.

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(\alpha t)]^{*} \parallel i[[\![\varphi]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet}) \xrightarrow{\tau} \quad (27)$$

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(\alpha t)]^{*} \parallel i[\mathsf{rcv}\,(\mathrm{tr}(\alpha) \rightarrow [\![\varphi]\!]^{\mathbf{m}}(l_{\mathrm{env}})\,;\,\_ \rightarrow \mathsf{ok})\,\mathsf{end}]^{\bullet}) \Longrightarrow \quad (28)$$

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(t)]^{*} \parallel i[\mathsf{rcv}\,(\mathrm{tr}(\alpha) \rightarrow [\![\varphi]\!]^{\mathbf{m}}(l_{\mathrm{env}})\,;\,\_ \rightarrow \mathsf{ok})\,\mathsf{end} \triangleleft \mathrm{tr}(\alpha)]^{\bullet}) \xrightarrow{\tau} \quad (29)$$

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(t)]^{*} \parallel i[[\![\varphi]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet})$$

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

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

and, thus, the result follows by (27), (28) and (29).

$A, s \models_{\mathrm{v}} (\mathsf{max}(X, \varphi))\theta$ **because** $A, s \models_{\mathrm{v}} \varphi\{\mathsf{max}(X,\varphi)/X\}\theta$**:** By Def. 6 and APP for process i, we derive

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\mathsf{max}(X,\varphi)]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet}) \Longrightarrow$$
$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\varphi]\!]^{\mathbf{m}}(\{'X', [\![\varphi]\!]^{\mathbf{m}}\} : l_{\mathrm{env}})]^{\bullet}) \quad (30)$$

Assuming the appropriate α-conversion for X in $\mathsf{max}(X, \varphi)$, we note that from $l_{\mathrm{env}} = \mathrm{enc}(\theta)$ and Def. 12 we obtain

$$\mathrm{enc}(\{\mathsf{max}(X,\varphi)/X\}\theta) = \{'X', [\![\varphi]\!]^{\mathbf{m}}\} : l_{\mathrm{env}} \quad (31)$$

By $A, s \models_{\mathrm{v}} \varphi\{\mathsf{max}(X,\varphi)/X\}\rho$, (31) and I.H. we obtain

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\varphi]\!]^{\mathbf{m}}(\{'X', [\![\varphi]\!]^{\mathbf{m}}\} : l_{\mathrm{env}})]^{\bullet}) \overset{\mathsf{sup!fail}}{\Longrightarrow} \quad (32)$$

The result follows from (30) and (32). □

In the other direction, Lemma 4 relies on Lemma 15, which establishes a correspondence between violation detections and actual violations, as formalised in Def. 5.

Lemma 15 relies on a technical result, Lemma 14 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 conjuction formula $\varphi_1 \wedge \varphi_2$. Lemma 14 employs Corollary 1, which in turn relies on the technical Lemmata 11 and 12, which prove the result for the specific cases were the violation is generated by the synthesised monitor of φ_1 and φ_2 *resp.*.

Lemma 11 *For some $l \leq n$:*

$$(j,h)\Big(i\left[\mathsf{fork}(j,h) \triangleleft q^1_{frk}q^2_{frk}\right]^\bullet \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{env}) \triangleleft q]^\bullet \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{env}) \triangleleft r]^\bullet\Big)(\xrightarrow{\tau})^n$$

$$(j,h)\Big(i\left[e^{j,h}_{fork} \triangleleft q^2_{frk}\right]^\bullet \parallel A \parallel B\Big) \xrightarrow{\mathsf{sup!fail}} \quad \textit{because} \quad A \xrightarrow{\mathsf{sup!fail}}$$

$$\textit{for some } A \equiv (\vec{j'})(j[e \triangleleft q']^\bullet \parallel A') \textit{ where } j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{env}) \triangleleft qq^1_{frk}]^\bullet \Longrightarrow A$$

$$\textit{and some } B \equiv (\vec{h'})(h[d \triangleleft r']^\bullet \parallel B') \textit{ where } h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{env}) \triangleleft rq^1_{frk}]^\bullet \Longrightarrow B$$

$$\textit{implies} \quad (j)(i_{mon}[\mathsf{mLoop}(j) \triangleleft q^1_{frk}q^2_{frk}]^* \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{env}) \triangleleft q]^\bullet)(\xrightarrow{\tau})^l \xrightarrow{\mathsf{sup!fail}}$$

and

$$(j,h)\Big(i\left[v,\ h!v,\ \mathsf{fork}(j,h) \triangleleft q^1_{frk}q^2_{frk}\right]^\bullet \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{env}) \triangleleft q]^\bullet \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{env}) \triangleleft r]^\bullet\Big)(\xrightarrow{\tau})^n$$

$$(j,h)\Big(i\left[e^{j,h}_{fork} \triangleleft q^2_{frk}\right]^\bullet \parallel A \parallel B\Big) \xrightarrow{\mathsf{sup!fail}} \quad \textit{because} \quad A \xrightarrow{\mathsf{sup!fail}}$$

$$\textit{for some } A \equiv (\vec{j'})(j[e \triangleleft q']^\bullet \parallel A') \textit{ where } j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{env}) \triangleleft qq^1_{frk}]^\bullet \Longrightarrow A$$

$$\textit{and some } B \equiv (\vec{h'})(h[d \triangleleft r']^\bullet \parallel B') \textit{ where } h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{env}) \triangleleft rq^1_{frk}]^\bullet \Longrightarrow B$$

$$\textit{implies} \quad (j)(i_{mon}[\mathsf{mLoop}(j) \triangleleft q^1_{frk}q^2_{frk}]^* \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{env}) \triangleleft q]^\bullet)(\xrightarrow{\tau})^l \xrightarrow{\mathsf{sup!fail}}$$

Proof We prove both statements simultaneously, by induction on the structure of the mailbox at actor i_{mon}, $(q^1_{\text{frk}}q^2_{\text{frk}})$.

$(q^1_{\text{frk}}q^2_{\text{frk}}) = \epsilon$: We prove the first case; the second case is analogous. From the structure of fork, Def. 6, we know that $i\ [\mathsf{fork}(j,h) \triangleleft \epsilon]^\bullet$ will get stuck after the function application, which means that it must be the case that $j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\text{env}}) \triangleleft q]^\bullet \Longrightarrow A$, from which the conclusion holds trivially by PAR and SCP.

$(q^1_{\text{frk}}q^2_{\text{frk}}) = v : q''_{\text{frk}}$: We prove the first case again, and leave the second, analogous case to the reader. We have two subcases to consider:

- Either $j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\text{env}}) \triangleleft q]^\bullet \Longrightarrow A$, independently of the actor at i_{mon}. The proof follows as in the base case.
- Or it is *not* the case that $j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\text{env}}) \triangleleft q]^\bullet \Longrightarrow A$. This means that the

reduction sequence can be decomposed as

$$(j,h)\begin{pmatrix} i\left[\mathsf{fork}(j,h) \triangleleft (q^1_{\mathrm{frk}} q^2_{\mathrm{frk}})\right]^\bullet \\ \parallel j[\![\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft q]^\bullet \parallel h[\![\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft r]^\bullet \end{pmatrix} (\xrightarrow{\tau})^k$$
$$(j,h)\Big(i\,[v,\ h!v,\ \mathsf{fork}(j,h) \triangleleft q''_{\mathrm{frk}}]^\bullet \parallel (\vec{j'})(j[e \triangleleft q' : v]^\bullet \parallel A''') \parallel B''\Big)(\xrightarrow{\tau})^{n-k} \tag{33}$$
$$(j,h)\Big(i\left[e^{j,h}_{\mathrm{fork}} \triangleleft q'_{\mathrm{frk}}\right]^\bullet \parallel A \parallel B\Big) \xrightarrow{\mathsf{sup!fail}}$$

for some $k = k_1 + k_2 + 3$, where

$$j[\![\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft q]^\bullet\ (\xrightarrow{\tau})^{k_1}\ (\vec{j'})(j[e \triangleleft q']^\bullet \parallel A''') \tag{34}$$
$$B'' \equiv (\vec{h'})(h[d \triangleleft r']^\bullet \parallel B''') \text{ where } h[\![\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft r]^\bullet\ (\xrightarrow{\tau})^{k_2}\ B''. \tag{35}$$

By (33), (34) and (35) we can reconstruct the reduction sequence

$$(j,h)\begin{pmatrix} i\left[v,\ h!v,\ \mathsf{fork}(j,h) \triangleleft q''_{\mathrm{frk}}\right]^\bullet \\ \parallel j[\![\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft q : v]^\bullet \parallel h[\![\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft r]^\bullet \end{pmatrix} (\xrightarrow{\tau})^{n-3}$$
$$(j,h)\Big(i\left[e^{j,h}_{\mathrm{fork}} \triangleleft q'_{\mathrm{frk}}\right]^\bullet \parallel A \parallel B\Big) \xrightarrow{\mathsf{sup!fail}}$$

By and I.H. we deduce that for $l \leq n-3$

$$(j)(i_{mon}[\mathsf{mLoop}(j) \triangleleft q''_{\mathrm{frk}}]^* \parallel j[\![\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft q : v]^\bullet)(\xrightarrow{\tau})^l \xrightarrow{\mathsf{sup!fail}}$$

Thus, by preceeding this transition sequence by three additional τ-transitions we obtain

$$(j)(i_{mon}[\mathsf{mLoop}(j) \triangleleft v : q''_{\mathrm{frk}}]^* \parallel j[\![\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft q]^\bullet)(\xrightarrow{\tau})^l \xrightarrow{\mathsf{sup!fail}}$$

□

Lemma 12 *For some $l \leq n$:*

$$
\begin{array}{c}
(j,h)\Big(i\Big[\mathsf{fork}(j,h) \triangleleft q^1_{frk}q^2_{frk}\Big]^\bullet \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{env}) \triangleleft q]^\bullet \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{env}) \triangleleft r]^\bullet\Big)(\xrightarrow{\tau})^n \\
(j,h)\Big(i\Big[e^{j,h}_{fork} \triangleleft q^2_{frk}\Big]^\bullet \parallel A \parallel B\Big) \xrightarrow{\mathsf{sup!fail}} \quad \textit{because} \quad B \xrightarrow{\mathsf{sup!fail}} \\
\textit{for some } A \equiv (\vec{j'})(j[e \triangleleft q']^\bullet \parallel A') \textit{ where } j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{env}) \triangleleft qq^1_{frk}]^\bullet \Longrightarrow A \\
\textit{and some } B \equiv (\vec{h'})(h[d \triangleleft r']^\bullet \parallel B') \textit{ where } h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{env}) \triangleleft rq^1_{frk}]^\bullet \Longrightarrow B \\
\textit{implies} \quad (h)(i_{mon}[\mathsf{mLoop}(h) \triangleleft q^1_{frk}q^2_{frk}]^* \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{env}) \triangleleft r]^\bullet)(\xrightarrow{\tau})^l \xrightarrow{\mathsf{sup!fail}}
\end{array}
$$

and

$$
\begin{array}{c}
(j,h)\Big(i\Big[v,\ \mathsf{fork}(j,h) \triangleleft q^1_{frk}q^2_{frk}\Big]^\bullet \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{env}) \triangleleft q]^\bullet \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{env}) \triangleleft r]^\bullet\Big)(\xrightarrow{\tau})^n \\
(j,h)\Big(i\Big[e^{j,h}_{fork} \triangleleft q^2_{frk}\Big]^\bullet \parallel A \parallel B\Big) \xrightarrow{\mathsf{sup!fail}} \quad \textit{because} \quad B \xrightarrow{\mathsf{sup!fail}} \\
\textit{for some } A \equiv (\vec{j'})(j[e \triangleleft q']^\bullet \parallel A') \textit{ where } j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{env}) \triangleleft qq^1_{frk}]^\bullet \Longrightarrow A \\
\textit{and some } B \equiv (\vec{h'})(h[d \triangleleft r']^\bullet \parallel B') \textit{ where } h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{env}) \triangleleft rq^1_{frk}]^\bullet \Longrightarrow B \\
\textit{implies} \quad (h)(i_{mon}[\mathsf{mLoop}(h) \triangleleft q^1_{frk}q^2_{frk}]^* \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{env}) \triangleleft r]^\bullet)(\xrightarrow{\tau})^l \xrightarrow{\mathsf{sup!fail}}
\end{array}
$$

Proof Analogous to the proof of Lemma 11. □

Corollary 1 *For some $l \leq n$:*

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

Proof Follows from Lemma 11 and Lemma 12 and the fact that $i[\mathsf{fork}(j,h) \triangleleft q_{\mathrm{frk}}]^\bullet$ cannot generate the action $\mathsf{sup!fail}$. □

Lemma 14 uses another technical result, Lemma 13, 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 13 (Mailbox Increase) $(\vec{h})(i[e \triangleleft q]^m \parallel A) \xrightarrow{\tau} (\vec{j})(i[e' \triangleleft q']^m \parallel B)$ *where* $i \notin \vec{h}$ *and* $q' \leq q$ *implies* $(\vec{h})(i[e \triangleleft q : v]^m \parallel A) \xrightarrow{\tau} (\vec{j})(i[e' \triangleleft q' : v]^m \parallel B)$

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

Equipped with Corollary 1 and Lemma 13, we are in a position to prove Lemma 14.

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

$$(i)\left(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^* \parallel (j,h)\left(\begin{array}{l} i\ [\mathsf{fork}(j,h) \triangleleft \mathrm{tr}(t)]^\bullet \\ \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{env})]^\bullet \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{env})]^\bullet \end{array}\right)\right)(\xrightarrow{\tau})^k \xrightarrow{\mathsf{sup!fail}}$$

$$\begin{aligned} \textit{implies}\quad & (i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(ts)]^* \parallel i[[\![\varphi_1]\!]^{\mathbf{m}}(l_{env})]^\bullet)(\xrightarrow{\tau})^l \xrightarrow{\mathsf{sup!fail}} \\ \textit{or}\quad & (i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(ts)]^* \parallel i[[\![\varphi_2]\!]^{\mathbf{m}}(l_{env})]^\bullet)(\xrightarrow{\tau})^l \xrightarrow{\mathsf{sup!fail}} \end{aligned}$$

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 $i_{mon}[\mathsf{mLoop}(i)]^*$ is stuck. Thus we conclude that it must be the case that

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

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

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

- If

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

for some $k \leq n$ then, by Cor. 1 we obtain

$$\begin{aligned} & (j)(i_{mon}[\mathsf{mLoop}(j) \triangleleft \mathrm{tr}(t)]^* \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet)(\xrightarrow{\tau})^l \xrightarrow{\mathsf{sup!fail}} \\ \text{or}\quad & (h)(i_{mon}[\mathsf{mLoop}(h) \triangleleft \mathrm{tr}(t)]^* \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet)(\xrightarrow{\tau})^l \xrightarrow{\mathsf{sup!fail}} \end{aligned}$$

for some $l \leq k$. By Lemma 13 we thus obtain

$$\begin{aligned} & (j)(i_{mon}[\mathsf{mLoop}(j) \triangleleft \mathrm{tr}(ts)]^* \parallel j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet)(\xrightarrow{\tau})^l \xrightarrow{\mathsf{sup!fail}} \\ \text{or}\quad & (h)(i_{mon}[\mathsf{mLoop}(h) \triangleleft \mathrm{tr}(ts)]^* \parallel h[[\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet)(\xrightarrow{\tau})^l \xrightarrow{\mathsf{sup!fail}} \end{aligned}$$

as required.

- Otherwise, it must be the case that

$$(i)\left(\begin{array}{l} i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^* \\ \parallel (j,h)\left(\begin{array}{l} i\,[\mathsf{fork}(j,h) \triangleleft \mathrm{tr}(t)]^\bullet \\ \parallel j[\![\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet \parallel h[\![\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet \end{array}\right)\end{array}\right)(\xrightarrow{\tau})^k \tag{36}$$

$$(i)\left(\begin{array}{l} i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s')]^* \\ \parallel (j,h)\,(i\,[e_{\mathrm{fork}} \triangleleft q : \mathrm{tr}(\alpha)]^\bullet \parallel A) \end{array}\right)(\xrightarrow{\tau})^{n-k} \xrightarrow{\mathsf{sup!fail}} \tag{37}$$

For some $k = 3 + k_1$ where

$$\begin{array}{r} (j,h)\left(\begin{array}{l} i\,[\mathsf{fork}(j,h) \triangleleft \mathrm{tr}(t)]^\bullet \\ \parallel j[\![\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet \parallel h[\![\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet \end{array}\right)(\xrightarrow{\tau})^{k_1} \\ (j,h)\,(i\,[e_{\mathrm{fork}} \triangleleft q]^\bullet \parallel A) \end{array} \tag{38}$$

By (38) and Lemma 13 we obtain

$$\begin{array}{r} (j,h)\left(\begin{array}{l} i\,[\mathsf{fork}(j,h) \triangleleft \mathrm{tr}(t) : \mathrm{tr}(\alpha)]^\bullet \\ \parallel j[\![\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet \parallel h[\![\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet \end{array}\right)(\xrightarrow{\tau})^{k_1} \\ (j,h)\,(i\,[e_{\mathrm{fork}} \triangleleft q : \mathrm{tr}(\alpha)]^\bullet \parallel A) \end{array}$$

and by (37) we can construct the sequence of transitions:

$$(i)\left(\begin{array}{l} i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s')]^* \\ \parallel (j,h)\left(\begin{array}{l} i\,[\mathsf{fork}(j,h) \triangleleft \mathrm{tr}(t) : \alpha]^\bullet \\ \parallel j[\![\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet \parallel h[\![\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet \end{array}\right)\end{array}\right)(\xrightarrow{\tau})^{n-3} \xrightarrow{\mathsf{sup!fail}}$$

Thus, by I.H. we obtain, for some $l \leq n - 3$

$$\begin{array}{rl} & (i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(t\alpha s')]^* \parallel i[\![\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet)(\xrightarrow{\tau})^l \xrightarrow{\mathsf{sup!fail}} \\ \text{or} & (i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(t\alpha s')]^* \parallel i[\![\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet)(\xrightarrow{\tau})^l \xrightarrow{\mathsf{sup!fail}} \end{array}$$

The result follows since $s = \alpha s'$. □

Lemma 15 *If* $A \overset{s}{\Longrightarrow}$, $l_{env} = \mathrm{enc}(\theta)$ *and* $(i)(i_{mon}[\mathit{mLoop}(i) \triangleleft \mathrm{tr}(s)]^* \parallel i[\![\![\varphi]\!]^{\mathbf{m}}(l_{env})]^\bullet) \overset{\mathit{sup!fail}}{\Longrightarrow}$ *then* $A, s \models_v \varphi\theta$*, whenever* $\mathrm{fv}(\varphi) \subseteq \mathrm{dom}(\theta)$.

Proof By strong induction on $(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^* \parallel i[\![\![\varphi]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet)(\xrightarrow{\tau})^n \xrightarrow{\mathsf{sup!fail}}$.

$n = 0$: By inspection of the definition for mLoop, and by case analysis of $[\![\varphi]\!]^{\mathbf{m}}(l_{\mathrm{env}})$

from Def. 6, it can never be the case that

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

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 Def. 5.

$\varphi = [\alpha]\psi$**:** By Def. 6, we know that

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^* \parallel i[[\![[\alpha]\psi]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet)(\xrightarrow{\tau})^{k_1} \tag{39}$$

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_2)]^* \parallel i[[\![[\alpha]\psi]\!]^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft \mathrm{tr}(s_1)]^\bullet) \xrightarrow{\tau} \tag{40}$$

$$(i)\left(\begin{array}{l} i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_2)]^* \parallel \\ i\left[\mathsf{rcv}\left(\begin{array}{l}\mathrm{tr}(\alpha) \rightarrow [\![\psi]\!]^{\mathbf{m}}(l_{\mathrm{env}})\,; \\ \_ \rightarrow \mathsf{ok}\end{array}\right)\mathsf{end} \triangleleft \mathrm{tr}(s_1)\right]^\bullet\end{array}\right)(\xrightarrow{\tau})^{k_2} \xrightarrow{\mathsf{sup!fail}} \tag{41}$$

$$\text{where } k + 1 = k_1 + k_2 + 1 \text{ and } s = s_1 s_2 \tag{42}$$

From the analysis of the code in (41), the only way for the action sup!fail to be triggered is by choosing the guarded branch $\mathrm{tr}(\alpha) \rightarrow [\![\varphi]\!]^{\mathbf{m}}(l_{\mathrm{env}})$ in actor i. This means that (41) can be decomposed into the following reduction sequences.

$$(i)\left(\begin{array}{l} i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_2)]^* \parallel \\ i\left[\mathsf{rcv}\left(\begin{array}{l}\mathrm{tr}(\alpha) \rightarrow [\![\psi]\!]^{\mathbf{m}}(l_{\mathrm{env}})\,; \\ \_ \rightarrow \mathsf{ok}\end{array}\right)\mathsf{end} \triangleleft \mathrm{tr}(s_1)\right]^\bullet\end{array}\right)(\xrightarrow{\tau})^{k_3} \tag{43}$$

$$(i)\left(\begin{array}{l} i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_4)]^* \parallel \\ i\left[\mathsf{rcv}\left(\begin{array}{l}\mathrm{tr}(\alpha) \rightarrow [\![\psi]\!]^{\mathbf{m}}(l_{\mathrm{env}})\,; \\ \_ \rightarrow \mathsf{ok}\end{array}\right)\mathsf{end} \triangleleft \mathrm{tr}(s_1 s_3)\right]^\bullet\end{array}\right) \xrightarrow{\tau} \tag{44}$$

$$(i)\left(\begin{array}{l} i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_4)]^* \parallel \\ i\,[[\![\psi]\!]^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft \mathrm{tr}(s_5)]^\bullet\end{array}\right)(\xrightarrow{\tau})^{k_4} \xrightarrow{\mathsf{sup!fail}} \tag{45}$$

$$\text{where } k_2 = k_3 + k_4 + 1 \text{ and } s_1 s_3 = \alpha s_5 \text{ and } s_2 = s_3 s_4 \tag{46}$$

By (42) and (46) we derive

$$s = \alpha t \text{ where } t = s_5 s_4 \tag{47}$$

From the definition of mLoop we can derive

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(t)]^{*} \parallel i[[\![\psi]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet})(\xrightarrow{\tau})^{k_5}$$
$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_4)]^{*} \parallel i\;[[\![\psi]\!]^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft \mathrm{tr}(s_5)]^{\bullet}) \quad (48)$$

where $k_5 \leq k_1 + k_3$. From (47) we can split $A \overset{s}{\Longrightarrow}$ as $A \overset{\alpha}{\Longrightarrow} A' \overset{t}{\Longrightarrow}$ and from (48), (45), the fact that $k_5 + k_4 < k + 1 = n$ from (42) and (46), and I.H. we obtain

$$A', t \models_{\mathrm{v}} \psi\theta \qquad (49)$$

From (49), $A \overset{\alpha}{\Longrightarrow} A'$ and Def. 5 we thus conclude $A, s \models_{\mathrm{v}} ([\alpha]\psi)\theta$.

$\varphi = \varphi_1 \wedge \varphi_2$ From Def. 6, we can decompose the transition sequence as follows

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\varphi_1 \wedge \varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet})(\xrightarrow{\tau})^{k_1} \qquad (50)$$

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_2)]^{*} \parallel i[[\![\varphi_1 \wedge \varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft \mathrm{tr}(s_1)]^{\bullet}) \xrightarrow{\tau} \qquad (51)$$

$$(i)\left(\begin{array}{l} i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_2)]^{*} \\ \parallel i \left[\begin{array}{ll} y_1 = \mathsf{spw}\,([\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})), & \\ y_2 = \mathsf{spw}\,([\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})), & \triangleleft \mathrm{tr}(s_1) \\ \mathsf{fork}(y_1, y_2) & \end{array}\right]^{\bullet} \end{array}\right)(\xrightarrow{\tau})^{k_2} \qquad (52)$$

$$(i)\left(\begin{array}{l} i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_4)]^{*} \\ \parallel i \left[\begin{array}{ll} y_1 = \mathsf{spw}\,([\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})), & \\ y_2 = \mathsf{spw}\,([\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})), & \triangleleft \mathrm{tr}(s_1 s_3) \\ \mathsf{fork}(y_1, y_2) & \end{array}\right]^{\bullet} \end{array}\right)(\xrightarrow{\tau})^{2} \qquad (53)$$

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

(54)

where $k + 1 = k_1 + 1 + k_2 + 2 + k_3$, $s = s_1 s_2$ and $s_2 = s_3 s_4$ (55)

From (54) we can deduce that there are two possible transition sequences how action $\mathsf{sup!fail}$ was reached:

1. If $\mathsf{sup!fail}$ was reached because

$$j[[\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet}(\xrightarrow{\tau})^{k_4} \xrightarrow{\mathsf{sup!fail}}$$

on its own, for some $k_4 \leq k_3$ then, by Par and Scp we deduce

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^* \parallel j[\![\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet)(\xrightarrow{\tau})^{k_4} \xrightarrow{\mathsf{sup!fail}}$$

From (55) we know that $k_4 < k+1 = n$, and by the premise $A \overset{s}{\Longrightarrow}$ and I.H. we obtain $A, s \models_{\mathrm{v}} \varphi_1\theta$. By Def. 5 we then obtain $A, s \models_{\mathrm{v}} (\varphi_1 \wedge \varphi_2)\theta$

2. Alternatively, (54) can be decomposed further as

$$(i)\left(\begin{array}{l} i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_4)]^* \\ \parallel (j)\left(\begin{array}{l} i\begin{bmatrix} y_2 = \mathsf{spw}\,([\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})), \\ \mathsf{fork}(j, y_2) \end{bmatrix} \triangleleft \mathrm{tr}(s_1 s_3)\end{array}\right]^\bullet \\ \quad \parallel j[\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet \end{array}\right)\right)(\xrightarrow{\tau})^{k_4} \tag{56}$$

$$(i)\left(\begin{array}{l} i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_6)]^* \\ \parallel (j)\left(\begin{array}{l} i\begin{bmatrix} y_2 = \mathsf{spw}\,([\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})), \\ \mathsf{fork}(j, y_2) \end{bmatrix} \triangleleft \mathrm{tr}(s_1 s_3 s_5)\end{array}\right]^\bullet \\ \quad \parallel j[\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet \end{array}\right)\right)(\xrightarrow{\tau})^{2} \tag{57}$$

$$(i)\left(\begin{array}{l} i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_6)]^* \\ \parallel (j,h)\left(\begin{array}{l} i\,[\mathsf{fork}(j,h) \triangleleft \mathrm{tr}(s_1 s_3 s_5)]^\bullet \\ \parallel j[\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet \\ \parallel h[\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet \end{array}\right)\end{array}\right)(\xrightarrow{\tau})^{k_5} \xrightarrow{\mathsf{sup!fail}} \tag{58}$$

$$\text{where} k_3 = k_4 + 2 + k_5 \text{ and } s_4 = s_5 s_6 \tag{59}$$

From (58) and Lemma 14 we know that either

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_1 s_3 s_5 s_6)]^* \parallel i[\![\varphi_1]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet)(\xrightarrow{\tau})^{k_6} \xrightarrow{\mathsf{sup!fail}}$$

$$\text{or } (i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_1 s_3 s_5 s_6)]^* \parallel i[\![\varphi_2]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet)(\xrightarrow{\tau})^{k_6} \xrightarrow{\mathsf{sup!fail}}$$

where $k_6 \leq k_5$

From (55) and (59) 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 $A, s \models_{\mathrm{v}} \varphi_1\theta$ or $A, s \models_{\mathrm{v}} \varphi_2\theta$, and in either case, by Def. 5 we deduce $A, s \models_{\mathrm{v}} (\varphi_1 \wedge \varphi_2)\theta$.

$\varphi = X$ By Def. 6, we can deconstruct $(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^* \parallel i[\![X]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^\bullet)(\xrightarrow{\tau}$

$)^{k+1} \xrightarrow{\mathsf{sup!fail}}$ as

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![X]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet}) \Longrightarrow\xrightarrow{\tau} \tag{60}$$

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_2)]^{*} \parallel i[y = \mathsf{lookUp}('X', l_{\mathrm{env}}), y(l_{\mathrm{env}}) \triangleleft \mathrm{tr}(s_1)]^{\bullet}) \Longrightarrow\xrightarrow{\tau} \tag{61}$$

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_4)]^{*} \parallel i[y = v, y(l_{\mathrm{env}}) \triangleleft \mathrm{tr}(s_1 s_3)]^{\bullet}) \Longrightarrow\xrightarrow{\tau} \tag{62}$$

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_6)]^{*} \parallel i[v(l_{\mathrm{env}}) \triangleleft \mathrm{tr}(s_1 s_3 s_5)]^{\bullet}) \Longrightarrow\xrightarrow{\mathsf{sup!fail}} \tag{63}$$

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

Since $X \in \mathrm{dom}(\theta)$, we know that

$$\theta(X) = \psi \tag{64}$$

for some ψ. By the assumption $l_{\mathrm{env}} = \mathrm{enc}(\theta)$ and Lemma 7 we obtain that $v = [\![\psi]\!]^{\mathbf{m}}$. Hence, by (60), (61), (62) and (63) we can reconstruct

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\psi]\!]^{\mathbf{m}}(l_{\mathrm{env}})]^{\bullet})(\xrightarrow{\tau})^{k_1}$$
$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s_6)]^{*} \parallel i[[\![\psi]\!]^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft \mathrm{tr}(s_1 s_3 s_5)]^{\bullet})(\xrightarrow{\tau})^{k_2} \xrightarrow{\mathsf{sup!fail}} \tag{65}$$

where $k_1 + k_2 < k + 1 = n$. By (65) and I.H. we obtain $A, s \models_{\mathrm{v}} \psi$, which is the result required, since by (64) we know that $X\theta = \psi$.

$\varphi = \mathsf{max}(X, \psi)$ By Def. 6, we can deconstruct

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

as follows:

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

from which we can reconstruct the transition sequence

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\psi]\!]^{\mathbf{m}}(\{'X', \psi\} : l_{\mathrm{env}})]^{\bullet})(\xrightarrow{\tau})^{k_1+k_2} \xrightarrow{\mathsf{sup!fail}} \tag{66}$$

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 (66), $A \overset{s}{\Longrightarrow}$ and I.H. to obtain

$A, s \models_{\mathrm{v}} \psi\{^{\mathsf{max}(X,\psi)}/_X\}\theta$. By Def. 5 we then conclude $A, s \models_{\mathrm{v}} \mathsf{max}(X,\psi)\theta$. □

We are now in a position to prove Lemma 4; we recall that the lemma was stated *wrt.* closed sHML formulas.

Recall Lemma 4 (Violation Detection). *Whenever* $A \overset{s}{\Longrightarrow}$ *then :*

$$A, s \models_{v} \varphi \quad \textit{iff} \quad i_{mon}[\mathsf{M}(\varphi) \triangleleft \mathrm{tr}(s)]^{*} \xRightarrow{\mathsf{sup!fail}}$$

Proof For the *only-if* case, we assume $A \overset{s}{\Longrightarrow}$ and $A, s \models_{\mathrm{v}} \varphi$ and are required to prove $i_{mon}[\mathsf{M}(\varphi) \triangleleft \mathrm{tr}(s)]^{*} \xRightarrow{\mathsf{sup!fail}}$. We recall from Sec. 4 that M was defined as

$$\lambda x_{\mathrm{frm}}.z_{\mathrm{pid}} = \mathsf{spw}\,([\![x_{\mathrm{frm}}]\!]^{\mathbf{m}}(\mathsf{nil})), \mathsf{mLoop}(z_{\mathrm{pid}}). \tag{67}$$

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

$$i_{mon}[\mathsf{M}(\varphi) \triangleleft \mathrm{tr}(s)]^{*} \Longrightarrow (i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\varphi]\!]^{\mathbf{m}}(\mathsf{nil})]^{\bullet}) \tag{68}$$

Assumption $A, s \models_{\mathrm{v}} \varphi$ can be rewritten as $A, s \models_{\mathrm{v}} \varphi\theta$ for $\theta = \epsilon$, and thus, by Def. 12 we know $\mathsf{nil} = \mathrm{enc}(\theta)$. By Lemma 10 we obtain

$$(i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\varphi]\!]^{\mathbf{m}}(\mathsf{nil})]^{\bullet}) \xRightarrow{\mathsf{sup!fail}} \tag{69}$$

and the result thus follows from (68) and (69).

For the *if* case, we assume $A \overset{s}{\Longrightarrow}$ and $i_{mon}[\mathsf{M}(\varphi) \triangleleft \mathrm{tr}(s)]^{*} \xRightarrow{\mathsf{sup!fail}}$ 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 Def. 12, $\mathsf{nil} = \mathrm{enc}(\theta)$. By (67) we can decompose the transition sequence $i_{mon}[\mathsf{M}(\varphi) \triangleleft \mathrm{tr}(s)]^{*} \xRightarrow{\mathsf{sup!fail}}$ as

$$\begin{aligned} & i_{mon}[\mathsf{M}(\varphi) \triangleleft \mathrm{tr}(s)]^{*}(\xrightarrow{\tau})^{3} \\ & \qquad (i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft \mathrm{tr}(s)]^{*} \parallel i[[\![\varphi]\!]^{\mathbf{m}}(\mathsf{nil})]^{\bullet}) \xRightarrow{\mathsf{sup!fail}} \end{aligned} \tag{70}$$

The result, *i.e.*, $A, s \models_{\mathrm{v}} \varphi$, follows from (70) and Lemma 15. □

6.1.2 Detection Preservation

In order to prove Lemma 5, we are able to require a stronger guarantee, *i.e.*, confluence under weak transitions (Lemma 18) for the concurrent monitors described in Def. 6. Lemma 18 relies heavily on Lemma 17.

Definition 13 (Confluence modulo Inputs with Identical Recepients))

$$\mathrm{cnf}(A) \overset{\mathit{def}}{=} A \xrightarrow{\gamma} A', A \xrightarrow{\pi} A'' \textit{ implies } \begin{cases} \gamma = i?v_1, \pi = i?v_2 \textit{ or;} \\ \gamma = \pi, A' = A'' \textit{ or;} \\ A' \xrightarrow{\pi} A''', A'' \xrightarrow{\gamma} A''' \textit{ for some } A''' \end{cases}$$

Before we embark on showing that our synthesised monitors (Def. 6) remain confluent after a sequence of silent transitions, Lemma 17 and Lemma 18, we find it convenient to prove a technical result, Lemma 16, 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 reason about the possible interactions that concurrent monitors may engage in when proving Lemma 17.

Lemma 16 (Monitor Reductions and Structure) *For all* $\varphi \in \mathrm{sHML}, q \in (\mathrm{VAL})^*$ *and* $\theta :: \mathrm{LVAR} \rightharpoonup \mathrm{sHML}$ *if* $i[[\![\varphi]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^{\bullet}(\xrightarrow{\tau})^n A$ *then*

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

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

   $\varphi = [\alpha]\psi$: $A \equiv i[\mathsf{rcv}\ (\mathrm{tr}(\alpha) \rightarrow [\![\psi]\!]^{\mathbf{m}}(\mathrm{enc}(\theta))\,;\, \_ \rightarrow \mathsf{ok})\ \mathsf{end} \triangleleft q]^{\bullet}$ *or*
   ($A \equiv B$ *where* $i[[\![\psi]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft r]^{\bullet}(\xrightarrow{\tau})^k B$ *for some* $k < n$ *and* $q = \mathrm{tr}(\alpha) : r$)
   or
   $A \equiv i[\mathsf{ok} \triangleleft r]^{\bullet}$ *where* $q = u : r$

   $\varphi = \varphi_1 \wedge \varphi_2$: $A \equiv i\left[\begin{array}{l} y_1 = \mathsf{spw}\,([\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta))), \\ \quad y_2 = \mathsf{spw}\,([\![\varphi_2]\!]^{\mathbf{m}}(\mathrm{enc}(\theta))), \mathsf{fork}(y_1, y_2) \end{array} \triangleleft q\right]^{\bullet}$
   or
   $A \equiv (j_1)\Big(\, i[e \triangleleft q]^{\bullet} \parallel (\widetilde{h_1})(j_1[e_1 \triangleleft q_1]^{\bullet} \parallel B)\,\Big)$ *where*

   - e *is* $y_1 = j_1, y_2 = \mathsf{spw}\,([\![\varphi_2]\!]^{\mathbf{m}}(\mathrm{enc}(\theta))), \mathsf{fork}(y_1, y_2)$ *or* $y_2 = \mathsf{spw}\ ([\![\varphi_2]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)))\,, \mathsf{fork}(j_1, y_2)$

- $j_1[[\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta))]^\bullet \ (\xrightarrow{\tau})^k \ (\widetilde{h_1})(j_1[e_1 \triangleleft q_1]^\bullet \parallel B)$ *for some* $k < n$

or

$$A \equiv (j_1, j_2)\begin{pmatrix} i[y_2 = j_2, \mathit{fork}(j_1, y_2) \triangleleft q]^\bullet \\ \parallel \ (\widetilde{h_1})(j_1[e_1 \triangleleft q_1]^\bullet \parallel B) \ \parallel \ (\widetilde{h_2})(j_2[e_2 \triangleleft q_2]^\bullet \parallel C) \end{pmatrix}$$

where

- $j_1[[\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta))]^\bullet \ (\xrightarrow{\tau})^k \ (\widetilde{h_1})(j_1[e_1 \triangleleft q_1]^\bullet \parallel B)$ *for some* $k < n$
- $j_2[[\![\varphi_2]\!]^{\mathbf{m}}(\mathrm{enc}(\theta))]^\bullet \ (\xrightarrow{\tau})^l \ (\widetilde{h_2})(j_2[e_2 \triangleleft q_2]^\bullet \parallel C)$ *for some* $l < n$

or

$A \equiv (j_1, j_2)\big(i[e \triangleleft r]^\bullet \ \parallel \ (\widetilde{h_1})(j_1[e_1 \triangleleft q_1']^\bullet \parallel B) \ \parallel \ (\widetilde{h_2})(j_2[e_2 \triangleleft q_2']^\bullet \parallel C) \big)$ *where*

- e *is either* $\mathit{fork}(j_1, j_2)$ *or* $(\mathsf{rcv}\ z \to j_1!z, j_2!z\, \mathsf{end}, \mathit{fork}(j_1, j_2))$ *or* $j_1!u, i_2!u, \mathit{fork}(j_1, j_2)$ *or* $j_2!u, \mathit{fork}(j_1, j_2)$
- $j_1[[\![\varphi_1]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q_1]^\bullet \ (\xrightarrow{\tau})^k \ (\widetilde{h_1})(j_1[e_1 \triangleleft q_1']^\bullet \parallel B)$ *for* $k < n$, $q_1 < q$
- $j_2[[\![\varphi_2]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q_2]^\bullet \ (\xrightarrow{\tau})^l \ (\widetilde{h_2})(j_2[e_2 \triangleleft q_2']^\bullet \parallel C)$ *for* $l < n$, $q_2 < q$

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

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

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* $A \equiv i[\mathit{exit} \triangleleft q]^\bullet$

$\varphi = \max(X, \psi)$**:** $A \equiv B$ *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 involved a long and tedious list of case analysis exhausting all possibilities.

$n = 0$ Trivially, $A \equiv i[[\![\varphi]\!]^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^\bullet$ and it cannot perform any external actions.

$n = k + 1$ We have $i[[\![\varphi]\!]^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft q]^\bullet (\xrightarrow{\tau})^k A' \xrightarrow{\tau} A$. The proof proceeds by case analysis on φ. We only consider the case when $\varphi = \varphi_1 \wedge \varphi_2$, which is the most involving. The proof for the other cases are analogous but simpler.

$\varphi = \varphi_1 \wedge \varphi_2$**:** By I.H., we know that the actor A' has one of the four stated forms above; we consider the second subcase, where

$$A' \equiv (j_1)\left(i \begin{bmatrix} y_1 = j_1, \\ y_2 = \mathsf{spw}\,(\llbracket\varphi_2\rrbracket^{\mathbf{m}}(\mathrm{enc}(\theta))), & \triangleleft\ q \\ \mathsf{fork}(y_1, y_2) \end{bmatrix}^{\bullet} \parallel (\widetilde{h_1})\,(j_1[e_1 \triangleleft q_1]^{\bullet} \parallel B) \right)$$

where

$$j_1[\llbracket\varphi_1\rrbracket^{\mathbf{m}}(\mathrm{enc}(\theta))]^{\bullet}\ (\xrightarrow{\tau})^{l}\ (\widetilde{h_1})\,(j_1[e_1 \triangleleft q_1]^{\bullet} \parallel B)\ \text{ for some } l < k. \qquad (71)$$

Since $l < k + 1$, by (71) and I.H. we also know that

$$(\widetilde{h_1})\,(j_1[e_1 \triangleleft q_1]^{\bullet} \parallel B) \xrightarrow{\alpha} B \text{ implies } \alpha = \mathsf{sup!fail} \qquad (72)$$

Thus, from the structure of the expression at actor i and (72) we know that $A' \xrightarrow{\tau} A$ can be generated as a result of two subcases:

- Par, Scp and

$$i[y_1 = j_1, y_2 = \mathsf{spw}\,(\llbracket\varphi_2\rrbracket^{\mathbf{m}}(\mathrm{enc}(\theta))), \mathsf{fork}(y_1, y_2) \triangleleft q]^{\bullet}$$
$$\xrightarrow{\tau} i[y_2 = \mathsf{spw}\,(\llbracket\varphi_2\rrbracket^{\mathbf{m}}(\mathrm{enc}(\theta))), \mathsf{fork}(j_1, y_2) \triangleleft q]^{\bullet}$$

  which means that the structure of A' is included in the cases stated, and by the structure of $i[y_2 = \mathsf{spw}\,(\llbracket\varphi_2\rrbracket^{\mathbf{m}}(\mathrm{enc}(\theta))), \mathsf{fork}(j_1, y_2) \triangleleft q]^{\bullet}$ and by (72) $A \xrightarrow{\alpha}$ implies $\alpha = \mathsf{sup!fail}$.

- Par, Scp and $(\widetilde{h_1})\,(j_1[e_1 \triangleleft q_1]^{\bullet} \parallel B) \xrightarrow{\tau} (\widetilde{h_1'})\left(j_1[e_1' \triangleleft q_1']^{\bullet} \parallel B'\right)$. By (71) we obtain

$$j_1[\llbracket\varphi_1\rrbracket^{\mathbf{m}}(\mathrm{enc}(\theta))]^{\bullet}\ (\xrightarrow{\tau})^{l+1}\ (\widetilde{h_1'})\,(j_1[e_1' \triangleleft q_1']^{\bullet} \parallel B')\ \text{ where } l + 1 < k + 1 \qquad (73)$$

  which means that A is of the required structure. Clearly, the expression at actor i cannot perform external actions, and since $l + 1 < k + 1$ we can apply the I.H. on (73) and deduce that $(\widetilde{h_1'})\left(j_1[e_1' \triangleleft q_1']^{\bullet} \parallel B'\right) \xrightarrow{\alpha}$ implies $\alpha = \mathsf{sup!fail}$. This allows us to conclude that $A \xrightarrow{\alpha}$ implies $\alpha = \mathsf{sup!fail}$. □

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

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

$n = 0$: 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.,*

$$i[\llbracket\varphi\rrbracket^{\mathbf{m}}(\mathrm{enc}(\theta)) \triangleleft q]^{\bullet} \xrightarrow{\tau} i[e \triangleleft q]^{\bullet} \text{ for some } e. \tag{74}$$

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 (74) and Lemma 13 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 16 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 16, A can have any of 4 general structures, one of which is

$$A \equiv (j_1, j_2)\left(i[j_2!u, \mathsf{fork}(j_1, j_2) \triangleleft q]^{\bullet} \begin{array}{l} \parallel (\widetilde{h_1})(j_1[e_1 \triangleleft q_1']^{\bullet} \parallel B) \\ \parallel (\widetilde{h_2})(j_2[e_2 \triangleleft q_2']^{\bullet} \parallel C) \end{array} \right) \tag{75}$$

where

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

$$j_2[\llbracket\varphi_2\rrbracket^{\mathbf{m}}(l_{\mathrm{env}}) \triangleleft q_2]^{\bullet} (\xrightarrow{\tau})^{l} (\widetilde{h_2})(j_2[e_2 \triangleleft q_2']^{\bullet} \parallel C) \text{ for } l < n, q_2 < q \tag{77}$$

By Lemma 16, (76) and (77) we also infer that the only external action that can be performed by the processes $(\widetilde{h_1})(j_1[e_1 \triangleleft q_1']^{\bullet} \parallel B)$ and $(\widetilde{h_2})(j_2[e_2 \triangleleft q_2']^{\bullet} \parallel C)$ is $\mathsf{sup!fail}$. Moreover by (76) and (77) we can also show that

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

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

$$A \xrightarrow{\tau} (j_1, j_2)\left(i[u, \mathsf{fork}(j_1, j_2) \blacktriangleleft q]^\bullet \begin{array}{l} \parallel (\widetilde{h_1})(j_1[e_1 \blacktriangleleft q_1']^\bullet \parallel B) \\ \parallel (\widetilde{h_2})(j_2[e_2 \blacktriangleleft q_2' : u]^\bullet \parallel C) \end{array} \right) \tag{78}$$

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

because

$$(\widetilde{h_1})(j_1[e_1 \blacktriangleleft q_1']^\bullet \parallel B) \xrightarrow{\tau} (\widetilde{h_1'})(j_1[e_1' \blacktriangleleft q_1'']^\bullet \parallel B') \tag{79}$$

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

because

$$(\widetilde{h_2})(j_2[e_2 \blacktriangleleft q_2']^\bullet \parallel C) \xrightarrow{\tau} (\widetilde{h_2'})(j_2[e_2' \blacktriangleleft q_2'']^\bullet \parallel C') \tag{80}$$

$$A \xrightarrow{i?v} (j_1, j_2)\left(\begin{array}{l} i[j_2!u, \mathsf{fork}(j_1, j_2) \blacktriangleleft q : v]^\bullet \\ \parallel (\widetilde{h_1})(j_1[e_1 \blacktriangleleft q_1']^\bullet \parallel B) \\ \parallel (\widetilde{h_2})(j_2[e_2 \blacktriangleleft q_2']^\bullet \parallel C) \end{array} \right) \tag{81}$$

We consider actions (78) and (80) and leave the other combinations for the interested reader. From (80) and Lemma 13 we derive

$$(\widetilde{h_2})(j_2[e_2 \blacktriangleleft q_2' : u]^\bullet \parallel C) \xrightarrow{\tau} (\widetilde{h_2'})(j_2[e_2' \blacktriangleleft q_2'' : u]^\bullet \parallel C')$$

and by Par and Scp we obtain

$$\begin{array}{l} (j_1, j_2)\left(i[u, \mathsf{fork}(j_1, j_2) \blacktriangleleft q]^\bullet \begin{array}{l} \parallel (\widetilde{h_1})(j_1[e_1 \blacktriangleleft q_1']^\bullet \parallel B) \\ \parallel (\widetilde{h_2})(j_2[e_2 \blacktriangleleft q_2' : u]^\bullet \parallel C) \end{array} \right) \xrightarrow{\tau} \\ \qquad (j_1, j_2)\left(i[u, \mathsf{fork}(j_1, j_2) \blacktriangleleft q]^\bullet \begin{array}{l} \parallel (\widetilde{h_1})(j_1[e_1 \blacktriangleleft q_1']^\bullet \parallel B) \\ \parallel (\widetilde{h_2'})(j_2[e_2' \blacktriangleleft q_2'' : u]^\bullet \parallel C') \end{array} \right) \end{array} \tag{82}$$

Using Com, Str, Par and Scp we can derive

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

$$(j_1, j_2)\left(i[u, \mathsf{fork}(j_1, j_2) \triangleleft q]^\bullet \begin{array}{l} \parallel (\widetilde{h}_1)(j_1[e_1 \triangleleft q_1']^\bullet \parallel B) \\ \parallel (\widetilde{h}_2')(j_2[e_2' \triangleleft q_2'' : u]^\bullet \parallel C') \end{array} \right) \quad (83)$$

thus we close the confluence diamond by (82) and (83). □

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

$$i_{mon}[\mathsf{M}(\varphi) \triangleleft q]^* \Longrightarrow A \;\; \textit{implies} \;\; \text{cnf}(A)$$

Proof By strong induction on n, the number of reduction steps $i_{mon}[\mathsf{M}(\varphi) \triangleleft q]^* (\xrightarrow{\tau})^n A$.

$n = 0$ We know $A = i_{mon}[\mathsf{M}(\varphi) \triangleleft q]^*$. It is confluent because it can perform either of two actions, namely a τ-action for the function application (see App in Figure 3), or else an external input at i_{mon}, (see RcvU in Fig 2). The matching moves can be constructed by RcvU on the one hand, and by Lemma 13 on the other, analogously to the base case of Lemma 17.

$n = k + 1$ By performing a similar analysis to that of Lemma 16, but for $i_{mon}[\mathsf{M}(\varphi) \triangleleft q]^*$, we can determine that this actor can only weakly transition to either of the following forms:

1. $A = i_{mon}[M = \mathsf{spw}\,(\llbracket\varphi\rrbracket^{\mathbf{m}}(\mathsf{nil})), \mathsf{mLoop}(M) \triangleleft q]^*$
2. $A \equiv (i)(i_{mon}[\mathsf{mLoop}(i) \triangleleft q]^* \parallel B)$ where $i[\llbracket\varphi\rrbracket^{\mathbf{m}}(l_{\text{env}}) \triangleleft r]^\bullet \Rightarrow B$ for some r.
3. $A \equiv (i)(i_{mon}[\mathsf{rcv}\, z \rightarrow i!z\, \mathsf{end}, \mathsf{mLoop}(i) \triangleleft q]^* \parallel B)$ where $i[\llbracket\varphi\rrbracket^{\mathbf{m}}(l_{\text{env}}) \triangleleft r]^\bullet \Rightarrow B$ for some r.
4. $A \equiv (i)(i_{mon}[i!v, \mathsf{mLoop}(i) \triangleleft q]^* \parallel B)$ where $i[\llbracket\varphi\rrbracket^{\mathbf{m}}(l_{\text{env}}) \triangleleft r]^\bullet \Rightarrow B$ for some r.
5. $A \equiv (i)(i_{mon}[v, \mathsf{mLoop}(i) \triangleleft q]^* \parallel B)$ where $i[\llbracket\varphi\rrbracket^{\mathbf{m}}(l_{\text{env}}) \triangleleft r]^\bullet \Rightarrow B$ for some r.

We here focus on the 4th case of monitor structure; the other cases are analogous. From $i[\llbracket\varphi\rrbracket^{\mathbf{m}}(l_{\text{env}}) \triangleleft r]^\bullet \Rightarrow B$ and Lemma 16 we know that

$$B \xrightarrow{\gamma} \quad \text{implies } \gamma = \mathsf{sup!fail} \text{ or } \gamma = \tau$$

$$B \equiv (\vec{h})(i[e \triangleleft r]^\bullet \parallel C) \quad \text{where } \text{fId}(B) = i$$

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

$$(i)(i_{mon}[i!v, \mathsf{mLoop}(i) \triangleleft q]^{*} \parallel B) \xrightarrow{i_{mon}?u} (i)(i_{mon}[i!v, \mathsf{mLoop}(i) \triangleleft q : u]^{*} \parallel B) \tag{84}$$

$$(i)(i_{mon}[i!v, \mathsf{mLoop}(i) \triangleleft q]^{*} \parallel B) \xrightarrow{\tau} (i)(i_{mon}[v, \mathsf{mLoop}(i) \triangleleft q]^{*} \parallel (\vec{h})(i[e \triangleleft r : v]^{\bullet} \parallel C)) \tag{85}$$

$$(i)(i_{mon}[i!v, \mathsf{mLoop}(i) \triangleleft q]^{*} \parallel B) \xrightarrow{\tau} (i)(i_{mon}[i!v, \mathsf{mLoop}(i) \triangleleft q]^{*} \parallel C) \tag{86}$$

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 (85) and (86), which can be commuted using Lemma 13, in analogous fashion to the proof for the base case. □

Lemma 18 allows us to prove Lemma 19, and subsequently Lemma 20; the latter Lemma implies Detection Preservation, Lemma 5, used by Theorem 3.

Lemma 19 *For all* $\varphi \in$ sHML, $q \in \text{Val}^{*}$

$$i_{mon}[\mathsf{M}(\varphi) \triangleleft q]^{*} \Longrightarrow A, A \xRightarrow{\mathsf{sup!fail}} \textit{ and } A \xrightarrow{\tau} B \textit{ implies } B \xRightarrow{\mathsf{sup!fail}}$$

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

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

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

$$B \xrightarrow{\tau} A'' \tag{87}$$

$$A' \xrightarrow{\tau} A'' \quad \text{for some } A'' \tag{88}$$

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

$$i_{mon}[\mathsf{M}(\varphi) \triangleleft q]^{*} \Longrightarrow A',$$

and subsequently, by (88), $A'(\xrightarrow{\tau})^k\cdot \xrightarrow{\text{sup!fail}}$ and I.H. we obtain $A'' \xRightarrow{\text{sup!fail}}$; the required result then follows from (87). □

Lemma 20 (Detection Confluence) *For all* $\varphi \in$ sHML, $q \in$ Val*

$$i_{mon}[M(\varphi) \triangleleft q]^* \Longrightarrow A, A \xRightarrow{sup!fail} \textit{ and } A \Longrightarrow B \textit{ implies } B \xRightarrow{sup!fail}$$

Proof By induction on $A(\xrightarrow{\tau})^n B$:

$n = 0$**:** The result is immediate since $B = A$.

$n = k + 1$**:** For some B', We have $A \xrightarrow{\tau} B'$ and $B'(\xrightarrow{\tau})^k B$. From $A \xrightarrow{\tau} B'$, $A \xRightarrow{\text{sup!fail}}$, $i_{mon}[\mathsf{M}(\varphi) \triangleleft q]^* \Longrightarrow A$ and Lemma 19 we obtain

$$B' \xRightarrow{\text{sup!fail}} . \tag{89}$$

By $i_{mon}[\mathsf{M}(\varphi) \triangleleft q]^* \Longrightarrow A$ and $A \xrightarrow{\tau} B'$ we obtain $i_{mon}[\mathsf{M}(\varphi) \triangleleft q]^* \Longrightarrow B'$ and by $B'(\xrightarrow{\tau})^k B$, (89) and I.H. we obtain $B \xRightarrow{\text{sup!fail}}$. □

Recall Lemma 5 (Detection Preservation). *For all* $\varphi \in$ sHML, $q \in$ Val*

$$i_{mon}[M(\varphi) \triangleleft q]^* \xRightarrow{sup!fail} \textit{ and } i_{mon}[M(\varphi) \triangleleft q]^* \Longrightarrow B \textit{ implies } B \xRightarrow{sup!fail}$$

Proof Immediate, by Lemma 20 for the special case where $i_{mon}[\mathsf{M}(\varphi) \triangleleft q]^* \Longrightarrow i_{mon}[\mathsf{M}(\varphi) \triangleleft q]^*$. □

6.1.3 Monitor Separability

With the body of supporting lemmata proved thus far, the proof for Lemma 6 turns out to be relatively straightforward. In particular, we make use of Lemma 3, relating the behaviour of a monitored system to the same system when unmonitored, Lemma 13 delineating behaviour preservation after extending mailbox contents at specific actors, and Lemma 16, so as to reason about the structure and generic behaviour of synthesised monitors.

Recall Lemma 6 (Monitor Separability). *For all basic actors $\varphi \in$ sHML, $A \in$ Actr where i_{mon} is fresh to A, and $s \in$ Act$^* \setminus \{\mathsf{sup}!\mathsf{fail}\}$,*

$$(i_{mon})(\lceil A \rceil \parallel i_{mon}[\mathsf{M}(\varphi)]^*) \overset{s}{\Rightarrow} B \textit{ implies } \exists B', B'' \textit{s.t.} \begin{cases} B \equiv (i_{mon})(B' \parallel B'') \\ A \overset{s}{\Longrightarrow} A' \textit{ s.t. } B' = \lceil A' \rceil \\ i_{mon}[\mathsf{M}(\varphi) \triangleleft \mathrm{tr}(s)]^* \Longrightarrow B'' \end{cases}$$

Proof By induction on n in $(i_{mon})(\lceil A \rceil \parallel i_{mon}[\mathsf{M}(\varphi)]^*)(\xrightarrow{\gamma_k})^n B$, the length of the sequence of actions:

$n = 0$**:** We know that $s = \epsilon$ and $A = (i_{mon})(\lceil A \rceil \parallel i_{mon}[\mathsf{M}(\varphi)]^*)$. Thus the conditions hold trivially.

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

$$C \equiv (i_{mon})(C' \parallel C'') \tag{90}$$

$$A \overset{t}{\Longrightarrow} A'' \textit{ s.t. } C' = \lceil A'' \rceil \tag{91}$$

$$i_{mon}[\mathsf{M}(\varphi) \triangleleft \mathrm{tr}(t)]^* \Longrightarrow C'' \tag{92}$$

$$\gamma = \tau \text{ implies } t = s \quad \text{and} \quad \gamma = \alpha \text{ implies } t\alpha = s \tag{93}$$

and by (92) and Lemma 16 we know that

$$C'' \equiv (\vec{h})(i_{mon}[e \triangleleft q]^* \parallel C''') \tag{94}$$

$$\mathrm{fId}(C'') = \{i_{mon}\} \tag{95}$$

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

$\gamma = \alpha$**:** By (93) we know that $s = t\alpha$. By (95) and (94), it must be the case that $C \equiv (i_{mon})(C' \parallel C'') \xrightarrow{\alpha} B$ happens because

$$\text{for some } B' \; C' \xrightarrow{\alpha} B' \tag{96}$$

$$B \equiv (i_{mon})(B' \parallel (\vec{h})(i_{mon}[e \triangleleft q : \mathrm{tr}(\alpha)]^* \parallel C''')) \tag{97}$$

By (96), (91) and Lemma 3 we know that $\exists A'$ such that $\lceil A' \rceil = B'$ and that $A'' \xrightarrow{\alpha} A'$. Thus by (91) and $s = t\alpha$ we obtain

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

By (92), (94) and repeated applications of Lemma 13 we also know that

$$i_{mon}[\mathsf{M}(\varphi) \triangleleft \mathrm{tr}(t) : \mathrm{tr}(\alpha)]^{*} = i_{mon}[\mathsf{M}(\varphi) \triangleleft \mathrm{tr}(s)]^{*} \Longrightarrow$$
$$(\vec{h})(i_{mon}[e \triangleleft q : \mathrm{tr}(\alpha)]^{*} \parallel C''') = B''$$

The result then follows from (97).

$\gamma = \tau$: 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. □

7 Related Work

Runtime monitoring of web services and component-based systems has attracted a considerable amount of study, *e.g.*, [FJN+11, GCN+07, CPQFC10] focussing on verifying systems with respect to external interactions with clients of the services or components. The type of correctness properties that this body of work describes is similar to the properties that can be specified using sHML formulas involving external interactions. These systems are generally based on the design by contract principle and RV is applied to verify whether such systems are satisfying the contract they were designed to satisfy. To the best of our knowledge, none of the works in this area tackles aspects of *correct* monitor synthesis but only focus their evaluation on performance benchmarks.

Colombo *et al.* [CFG11, CFG12] present an RV tool, eLarva, that also makes use of the tracing mechanism in Erlang to gather the execution trace of the system. In eLarva, a specification of an automata is synthesised into an Erlang program. Several other works and tools such as MOP [CR03, CR07], Larva [CPS09] and polyLarva [CFMP12] implement the monitors using an aspect-oriented programming approach by synthesising monitors as aspects that are weaved at runtime with the system under test. The concurrency aspect in these tools is hidden within the AOP library being used, in most cases AspectJ. Thus, the correctness of the tool also depends on how concurrency is well-handled in such AOP tools. In these works, the authors do not tackle the issue of correctly synthesising the specification *i.e.*, the correctness of the translation between the specification of the property to the concrete monitor.

The works of Geilen [Gei01] and Sen [SRA04] focus on monitor correctness for LTL formulas defined over infinite strings and give their synthesis in terms of automata and pseudo-code *resp.*. Leuker *et al.* [BLS11] also present a translation from LTL to non-deterministic Büchi automata that recognizes good and bad prefixes and as a result acts as monitors for a specified LTL formula. The monitor specifications however sit at a higher level of abstraction than ours, *e.g.*, Büchi automata, and avoids issues relating to concurrent monitoring.

Fredlund [Fre01] adapted the μ-calculus, which is a variant of HML with recursion, to specify correctness properties in Erlang albeit for model-checking purposes. Fredlund also gives a formal semantics for Erlang but does not provide a monitoring semantics for the tracing functionality available in the EVM.

Finally, there is work that gives translation of HML with recursion formulas to tests in a process calculus such as CCS [Mil82]. Aceto *et al.* [AI99] gives such a translation, restricting it to a subset of the HML with recursion for safety properties, similar to the logic discussed in Section 3. Cerone and Hennessy [CH10] built on this and differentiate between *may* and *must* testing. None of the synthesised CCS tests are concurrent however; we have also already discussed how tests differ from monitors in Section 4.

8 Conclusions

We have studied the problem of ensuring monitor correctness in concurrent settings. In particular, we have constructed a formally-specified tool that automatically synthesises monitors from sHML formulas so as to asynchronously detect property violation by Erlang programs at runtime. We have then showed how the synthesised monitors can be proven correct with respect to a novel definition for monitor correctness in concurrent settings.

The contributions of the paper are:

1. We present a *monitoring semantics* for a core subset of Erlang, Figures 2, 3, 4 and 5, modelling the tracing mechanism offered by the Erlang VM, and used by asynchronous monitoring tools such as the one presented in this paper and by others, such as Exago [DWA$^+$10] and eLarva [CFG11, CFG12].

2. We adapt sHML to be able to specify safety Erlang properties, Def. 4. We give an alternative, *violation* characterisation for sHML, Def. 5, that is more amenable to reasoning about violating executions of concurrent (Erlang) programs. We prove that this violation relation corresponds to the dual of the satisfaction definition of the same logic, Theorem 1, and also prove that sHML is indeed a safety language, Theorem 2, in the sense of [MP90, CMP92].

3. We give a formal translation form sHML formulas to Erlang expressions from which we construct a tool, available at [Sey13], that allows us to *automate concurrent-monitor synthesis*.

4. We formalise a novel definition of *monitor correctness* in concurrent settings, Def. 11, dealing with issues such as non-determinism and divergence.

5. We propose a *proof technique* that teases apart different aspects of the monitor correctness definition, Lemma 4, Lemma 5 and Lemma 6, allowing us to prove correctness in stages. We subsequently apply this technique to *prove that our synthesised monitors are correct*, Theorem 3.

Future Work There are many avenues emanating from this work that are worth investigating. The Erlang monitoring semantics of Section 2 can be used as a basis to formally prove existing Erlang monitoring tools such as [CFG11, CFG12]. There is also substantial work to be done on logics for runtime verification; a simple example would be extending sHML to handle limited, monitorable forms of liveness properties (often termed co-safety properties [MP90, CMP92]). It is also worth exploring mechanisms for synchronous monitoring, as opposed to asynchronous variant studied in this paper.

The overheads of the monitors produced by our synthesis need to be assessed better. We also conjecture that more efficient monitor synthesis, perhaps using Erlang features such as process linking and supervision, can be attained; our framework can probably be extended in straightforward fashion to reason about the correctness of these more efficient monitors.

Monitor distribution can also be used to lower monitoring overheads even further [CFMP12] and Erlang provides the necessary abstractions to facilitate distribution. Distributed monitoring can also be used to increase the expressivity of our tool so as to handle correctness properties for distributed programs. The latter extension, however, poses a departure from our setting because the unique trace described by our framework would be replaced by separate independent traces at each location, and the lack of a total ordering of events may prohibit the detection of certain violations [FGP11, FGP12].

There is also substantial work to be done on how to handle the violations detected, which changes the focus from runtime verification to runtime enforcement [FFM12]. In the case of Erlang, corrective actions may range from the complete halting of the system, to more surgical halting of the violating actors (through exit-killing and trapping mechanism[Arm07, CT09]), to the runtime replacement of the violating sub-system with “limp-home” code that provides limited functionality known (a-priori) to be safe; the latter violation handling mechanisms allow for a more pragmatic, graceful degradation of the system being analysed.

References

[AI99] Luca Aceto and Anna Ingólfsdóttir. Testing Hennessy-Milner Logic with Recursion. In *Proceedings of the Second International Conference on Foundations of Software Science and Computation Structure, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, FoSSaCS '99, pages 41–55, London, UK, 1999. Springer-Verlag.

[Arm07] Joe Armstrong. *Programming Erlang - Software for a Concurrent World.* The Pragmatic Bookshelf, 2007.

[BFF$^+$10] Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors. *First International Conference, RV 2010*, volume 6418 of *Lecture Notes in Computer Science*. Springer, 2010.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20:14:1–14:64, September 2011.

[Car01] Richard Carlsson. An introduction to core erlang. In *PLI01 (Erlang Workshop)*, 2001.

[CFG11] Christian Colombo, Adrian Francalanza, and Rudolph Gatt. Elarva: A monitoring tool for erlang. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, volume 7186 of *Lecture Notes in Computer Science*, pages 370–374. Springer Berlin Heidelberg, 2011.

[CFG12] Christian Colombo, Adrian Francalanza, and Ian Grima. Simplifying contract-violating traces. In *Proceedings Sixth Workshop on Formal Languages and Analysis of Contract-Oriented Software*, volume 94 of *EPTCS*, pages 11–20, 2012.

[CFMP12] Christian Colombo, Adrian Francalanza, Ruth Mizzi, and Gordon J. Pace. polylarva: Runtime verification with configurable resource-aware monitoring boundaries. In *SEFM*, pages 218–232, 2012.

[CH10] Andrea Cerone and Matthew Hennessy. Process behaviour: Formulae vs. tests (extended abstract). In *EXPRESS'10*, pages 31–45, 2010.

[CMP92] Edward Chang, Zohar Manna, and Amir Pnueli. Characterization of temporal property classes. In *In Proc. Automata, Languages and Programming, volume 623 of LNCS*, pages 474–486. Springer-Verlag, 1992.

[CPQFC10] Tien-Dung Cao, Trung-Tien Phan-Quang, P. Felix, and R. Castanet. Automated runtime verification for web services. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 76 –82, july 2010.

[CPS09] C. Colombo, G.J. Pace, and G. Schneider. Larva — safer monitoring of real-time java programs (tool paper). In *Software Engineering and Formal Methods, 2009 Seventh IEEE International Conference on*, pages 33 –37, nov. 2009.

[CR03] Feng Chen and Grigore Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Electronic Notes in Theoretical Computer Science*, pages 106–125. Elsevier Science, 2003.

[CR07] Feng Chen and Grigore Rosu. Mop: An efficient and generic runtime verification framework. Technical Report UIUCDCS-R-2007-2836, University of Illinois at Urbana-Champaign, March 2007.

[CT09] Francesco Cesarini and Simon Thompson. *ERLANG Programming*. O'Reilly, 2009.

[DWA+10] John Derrick, Neil Walkinshaw, Thomas Arts, Clara Benac Earle, Francesco Cesarini, Lars-Ake Fredlund, Victor Gulias, John Hughes, and Simon Thompson. Property-based testing - the protest project. In FrankS. Boer, MarcelloM. Bonsangue, Stefan Hallerstede, and Michael Leuschel, editors, *Formal Methods for Components and Objects*, volume 6286 of *Lecture Notes in Computer Science*, pages 250–271. Springer Berlin Heidelberg, 2010.

[FFM12] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.

[FGP11] Adrian Francalanza, Andrew Gauci, and Gordon J. Pace. Distributed system contract monitoring. In *FLACOS*, volume 68 of *EPTCS*, pages 23–37, 2011.

[FGP12] Adrian Francalanza, Andrew Gauci, and Gordon Pace. Distributed system contract monitoring. Accessible at `http://staff.um.edu.mt/afra1/papers/mon-framewk.pdf` (submitted for publication), 2012.

[FJN+11] Yliès Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga, and Saddek Bensalem. Runtime verification of component-based systems. In

Software Engineering and Formal Methods (SEFM), volume 7041 of *Lecture Notes in Computer Science*, pages 204–220. Springer, 2011.

[Fre01] Lars-Åke Fredlund. *A Framework for Reasoning about Erlang Code*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2001.

[GCN+07] Yuan Gan, Marsha Chechik, Shiva Nejati, Jon Bennett, Bill O'Farrell, and Julie Waterhouse. Runtime monitoring of web service conversations. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, CASCON '07, pages 42–57, Riverton, NJ, USA, 2007. IBM Corp.

[Gei01] Marc Geilen. On the construction of monitors for temporal logic properties. *ENTCS*, 55(2):181–199, 2001.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245. Morgan Kaufmann, 1973.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, January 1985.

[Koz83] Dexter Kozen. Results on the propositional μ-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[KVK+04] MoonZoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-mac: A run-time assurance approach for java programs. *Formal Methods in System Design*, 24:129–155, 2004.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Mil89] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[MJG+11] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, 2011. To appear.

[MP90] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties (invited paper, 1989). In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, PODC '90, pages 377–410, New York, NY, USA, 1990. ACM.

[MPW93] Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114:149–171, 1993.

[RV07] Arend Rensink and Walter Vogler. Fair testing. *Inf. Comput.*, 205(2):125–198, February 2007.

[Sey13] Aldrin Seychell. DetectEr, 2013. Accessible at `http://www.cs.um.edu.mt/svrg/Tools/detectEr/`.

[SFBE10] Hans Svensson, Lars-Åke Fredlund, and Clara Benac Earle. A unified semantics for future erlang. In *Proceedings of the 9th ACM SIGPLAN workshop on Erlang*, Erlang '10, pages 23–32, New York, NY, USA, 2010. ACM.

[SRA04] Koushik Sen, Grigore Rosu, and Gul Agha. Generating optimal linear temporal logic monitors by coinduction. In *ASIAN*, volume 2896 of *LNCS*, pages 260–275. Springer-Verlag, 2004.

[SVAR04] Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Roşu. Efficient decentralized monitoring of safety in distributed systems. *International Conference on Software Engineering*, pages 418–427, 2004.

9 Appendix

Definition 14 *We define* $\mathrm{id_v}$, $\mathrm{id_e}$ *and* $\mathrm{id_g}$ *as follows:*

$$\mathrm{id_v}(v) \stackrel{\mathit{def}}{=} \begin{cases} \{i\} & \mathit{if}\ v = i \\ \mathrm{id_v}(u) & \mathit{if}\ v = \mu y.\lambda x.u \\ \mathrm{id_e}(e) & \mathit{if}\ v = \mu y.\lambda x.e \\ \bigcup_{i=0}^{n} \mathrm{id_v}(u_i) & \mathit{if}\ v = \{u_0, \dots u_n\} \\ \mathrm{id_v}(u) \cup \mathrm{id_v}(l) & \mathit{if}\ v = u : l \\ \emptyset & \mathit{if}\ v = a\ \mathit{or}\ x\ \mathit{or}\ \mathsf{nil}\ \mathit{or}\ \mathsf{exit} \end{cases}$$

$$\mathrm{id_e}(e) \stackrel{\mathit{def}}{=} \begin{cases} \mathrm{id_e}(d) & \mathit{if}\ e = \mathsf{spw}\, d \\ \mathrm{id_e}(e') \cup \mathrm{id_e}(d) & \mathit{if}\ e = e'!d\ \ \mathit{or}\ \ \mathsf{try}\, e'\, \mathsf{catch}\, d\ \ \mathit{or}\ \ x = e', d\ \ \mathit{or}\ \ e'(d) \\ \mathrm{id_g}(g) & \mathit{if}\ e = \mathsf{rcv}\, g\, \mathsf{end} \\ \mathrm{id_e}(d) \cup \mathrm{id_g}(g) & \mathit{if}\ e = \mathsf{case}\, d\, \mathsf{of}\, g\, \mathsf{end} \\ \emptyset & \mathit{if}\ e = \mathsf{self}\ \mathit{or}\ x\ \mathit{or}\ \mathsf{nil}\ \mathit{or}\ \mathsf{exit} \end{cases}$$

$$\mathrm{id_g}(g) \stackrel{\mathit{def}}{=} \begin{cases} \mathrm{id_e}(e) \cup \mathrm{id_g}(f) & \mathit{if}\ g = p \to e\, ; f \\ \emptyset & \mathit{if}\ g = \epsilon \end{cases}$$