Runtime Adaptation for Actor Systems

Ian Cassar and Adrian Francalanza

CS, ICT, University of Malta, Malta. {icas0005,afra1}@um.edu.mt

Abstract. We study the problem of extending RV techniques in the context of (asynchronous) actor systems, so as to be able to carry out a degree of system *adaptation* at runtime. We propose extensions to specification logics that provide handles for *programming* both monitor synchronisations (with individual actors), as well as the administration of the *resp.* adaptations once the triggering behaviour is observed. Since this added functionality allows the specifier to introduce *erroneous* adaptation procedures, we also develop *static analysis* techniques based on substructural type systems to assist the construction of correct adaptation scripts.

1 Introduction

Runtime Adaptation (RA) [17, 16] is a technique prevalent to long-running, highly available software systems, whereby system characteristics (*e.g.,* its structure, locality *etc.*) are altered *dynamically* in response to runtime events (*e.g.,* detected hardware faults or software bugs, changes in system loads), while causing *limited disruption* to the execution of the system. Numerous examples can be found in service-oriented architectures [21, 15] (*e.g.,* cloud-services, web-services, *etc.*) for self-configuring, self-optimising and self-healing purposes; the inherent component-based, decoupled organisation of such systems facilitates the implementation of adaptive actions *affecting a subset* of the system while allowing other parts to continue executing normally.

Actor systems [2, 14, 9] consist of *independently*-executing components called *actors*. Every actor is *uniquely-identifiable*, has its own *local memory*, and can either *spawn* other actors or interacting with them through *asynchronous messaging*.[1] Actors are often used to build service-oriented systems with limited downtime [3, 14]. Coding practices such as *fail-fast* design-patterns [9, 14] already advocate for a degree of RA for building robust, fault-tolerant systems: through mechanisms such as *process linking* and *supervision trees*, crashed actors are detected by *supervisor* actors, which respond through adaptations such as *restarting* the actors, *replacing* them with limp-home surrogate actors, or *killing* further actors that may potentially be affected by the crash.

In this paper, we study ways how RA for actor systems can be *extended* to respond to runtime events that go beyond actor crashes. For instance, we would like to observe *sequences of events* that allow us to take *preemptive* action before a crash happens; alternatively, we would also like to observe *positive* (liveness) events that allow us to adapt the system to execute more *efficiently* (*e.g.,* by switching off unused parts). More generally, we intend to develop a framework for extending actor-system functionality through RA, so as to improve aspects such as resilience and resource management.

[1] Messages are received in a message buffer called a *mailbox*, read only by the owning actor.

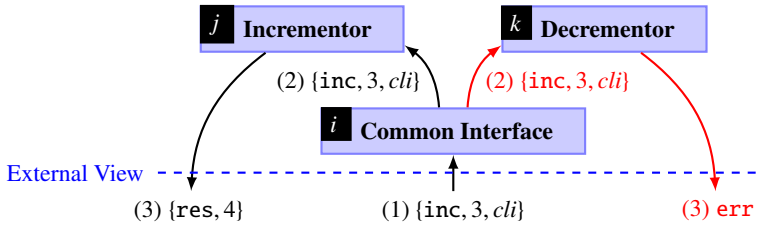

Fig. 1. A server actor implementation offering integer increment and decrement services

We propose to do this by *extending existing* Runtime Verification (RV) tools such as [25, 11, 10, 12]. The appeal of such an approach is that RV tools *already* provide mechanisms for *specifying* the behaviour to be observed, together with *instrumentation mechanisms* for observing such behaviour. As a proof-of-concept, our study focusses on one of these actor-based RV tools — detectEr[2] [12, 7] — and investigates ways how violation detections can be *replaced* by adaptation actions that respond to behaviours detected, while reusing as many elements as possible from the hosting technology.

Example 1. Fig. 1 depicts a server consisting of a front-end *Common Interface* actor with identifier i receiving client requests, a back-end *Incrementor* actor with identifier j, handling integer increment requests, and a back-end *Decrementor* actor k, handing decrement requests. A client sends service requests to actor i of the form $\{tag, arg, ret\}$ where *tag* selects the type of service, *arg* carries the service arguments and *ret* specifies the return address for the result (typically the client actor ID). The interface actor forwards the request to one of its back-end servers (depending on the *tag*) whereas the back-end servers process the requests, sending results (or error messages) to *ret*. The tool detectEr allows us to specify safety properties such as (1), explained below:

$$\varphi \stackrel{\text{def}}{=} \mathsf{max}\ Y.\ [i?\{\mathtt{inc}, x, y\}]\ (\ ([j \triangleright y!\{\mathtt{res}, x+1\}]\, Y)\ \&\ ([\_ \triangleright y!\mathtt{err}]\,\mathsf{ff})\ ) \qquad (1)$$

It is a (recursive, max $Y. \ldots$) property requiring that, from an *external* viewpoint, *every* increment request received by i, action $i?\{\mathtt{inc}, x, y\}$, is followed by an answer from j to the address y carrying $x+1$, action $j \triangleright y!\{\mathtt{res}, x+1\}$ (recursing through variable Y). However, increment requests followed by an error message sent from *any* actor back to y, action $\_ \triangleright y!\mathtt{err}$, represent a violation, ff. detectEr can synthesise a monitor (a system of actors) corresponding to (1) and instrument it with a system execution [12].

$$\mathsf{max}\ Y.\ [i?\{\mathtt{inc}, x, y\}]\Big((\,[j \triangleright y!\{\mathtt{res}, x+1\}]\ Y)\ \&\ ([z \triangleright y!\mathtt{err}]\ \mathsf{restr}(i)\ \mathsf{prg}(z)\ Y)\Big) \qquad (2)$$

We aim to *extend* properties such as (1) with *adaptation actions* to be taken by the monitor once a violation is detected, as shown in property (2) above. The specifier presumes that the error (which may arise after a number of correct interactions) is caused by the interface actor i (as shown in Fig. 1, where an `inc` request is erroneously forwarded to the decrementor actor k) — she may, for instance, have prior knowledge that actor i is a newly-installed, untested component. The monitor thus restarts actor i,

[2] An RV tool for long-running reactive (actor) systems written in Erlang [3].

adaptation restr(i), and empties the mailbox of the backend server—which may contain more erroneously forwarded messages—through adaptation prg(z) (the actor to be purged is determined at runtime, where z is bound to identifier k from the previous action $[z \triangleright y!\texttt{err}]$). Importantly, note that in the above execution (where k is the actor sending the error message), actor j is *not affected* by any adaptation action taken. ■

To implement adaptation sequences such as those in Ex. 1, the *resp.* monitors require adequate synchronisation control over the asynchronous system being monitored. For instance, to mitigate effects of erroneous components as soon as detections are made, the monitor may want to synchronise with the execution of actor i from Fig. 1 each time a client request is received, temporarily suspending its execution until it determines that the request is serviced correctly (at which point it is released again). Moreover, adaptations such as actor restarts and mailbox purging require the *resp.* actors to be *temporarily suspended* for the adaptation implementation to execute correctly.

Adequate synchronisation procedures are generally hard to infer and automate from specification scripts such as (2) of Ex. 1 (*e.g.,* an early suspension of actor i's execution — before it communicates with one of its backend actors — stalls the entire system) and is exacerbated by findings in prior work [7], concluding that actor synchronisations carry substantial overheads and should thus be kept to a minimum. In our work, we thus extend the specification language to include *explicit* de/synchronisation commands, thereby transferring synchronisation responsibility to the specifier of the adaptation property. This allows for *fine-tuned* synchronisations carrying low overheads, but also permits the specifier to *introduce synchronisation errors* herself (*e.g.,* applying synchronous adaptations to actors that are not yet synchronised).

Therefore, in this paper we also develop a type system that analyses specification scripts with adaptations and de/synchronisation commands and identifies errors prior to deployment. We also prove that the type system is—in some sense—sound, accepting scripts are free from certain monitor runtime errors. Since static analyses typically approximate computation, the type system may reject otherwise valid specification scripts; in such cases, the specifier may use the type system as a tool assisting script development, directing her to the parts that may potentially lead to errors.

In what follows, Sec. 2 presents the logic used by detectEr and Sec. 3 and Sec. 4 extend this with synchronisation directives and adaptation mechanisms. Sec. 5 presents our type system. This is accompanied in Sec. 6 by an extended runtime semantics for monitors carrying dynamic checks corresponding to the type disciplines of Sec. 5; we prove type system soundness *wrt.* this runtime semantics. Sec. 7 concludes.

2 The Logic

Following [12, 7], detectEr (safety) properties are expressed using the logic sHML [1], a syntactic subset of the modal μ-calculus [18]. The syntax is defined in Fig. 2 and assumes two distinct denumerable sets of *term variables*, $x, y, \ldots \in \textsc{Var}$ (to quantify over values) and *formula variables* $X, Y, \ldots \in \textsc{LVar}$ (to define recursive logical formulas). It is parametrised by boolean expressions, $b, c \in \textsc{Bool}$ equipped with a *decidable* evaluation function, $b \Downarrow c$ where $c \in \{\mathsf{true}, \mathsf{false}\}$, and a set of action patterns $e \in \textsc{Pat}$ that may

Syntax

$$\varphi, \psi \in \textsc{Frm} \ ::= \ \mathsf{tt} \mid \mathsf{ff} \mid \varphi \,\&\, \psi \mid [e]\varphi \mid X \mid \mathsf{max}\, X.\, \varphi \mid \mathsf{if}\, b\, \mathsf{then}\, \varphi\, \mathsf{else}\, \psi$$

Semantics

$$\varphi_1 \,\&\, \varphi_2 \equiv \varphi_2 \,\&\, \varphi_1 \qquad \varphi_1 \,\&\, (\varphi_2 \,\&\, \varphi_3) \equiv (\varphi_1 \,\&\, \varphi_2) \,\&\, \varphi_3 \qquad \mathsf{tt} \,\&\, \varphi \equiv \varphi \qquad \mathsf{ff} \,\&\, \varphi \equiv \mathsf{ff}$$

$$\textsc{rIdem1}\ \frac{}{\mathsf{ff} \xrightarrow{\alpha} \mathsf{ff}} \qquad \textsc{rIdem2}\ \frac{}{\mathsf{tt} \xrightarrow{\alpha} \mathsf{tt}}$$

$$\textsc{rTru}\ \frac{b \Downarrow \mathsf{true}}{\mathsf{if}\, b\, \mathsf{then}\, \varphi\, \mathsf{else}\, \psi \xrightarrow{\tau} \varphi} \qquad \textsc{rFls}\ \frac{b \Downarrow \mathsf{false}}{\mathsf{if}\, b\, \mathsf{then}\, \varphi\, \mathsf{else}\, \psi \xrightarrow{\tau} \psi} \qquad \textsc{rStr}\ \frac{\varphi \equiv \varphi' \xrightarrow{\gamma} \psi' \equiv \psi}{\varphi \xrightarrow{\gamma} \psi}$$

$$\textsc{rCn1}\ \frac{\varphi \xrightarrow{\alpha} \varphi' \quad \psi \xrightarrow{\alpha} \psi'}{\varphi \,\&\, \psi \xrightarrow{\alpha} \varphi' \,\&\, \psi'} \qquad \textsc{rCn2}\ \frac{\varphi \xrightarrow{\tau} \varphi'}{\varphi \,\&\, \psi \xrightarrow{\tau} \varphi' \,\&\, \psi} \qquad \textsc{rCn3}\ \frac{\psi \xrightarrow{\tau} \psi'}{\varphi \,\&\, \psi \xrightarrow{\tau} \varphi \,\&\, \psi'}$$

$$\textsc{rMax}\ \frac{}{\mathsf{max}\, X.\, \varphi \xrightarrow{\tau} \varphi[\mathsf{max}\, X.\, \varphi / X]} \qquad \textsc{rNc1}\ \frac{\mathsf{mtch}(e, \alpha) = \sigma}{[e]\varphi \xrightarrow{\alpha} \varphi\sigma} \qquad \textsc{rNc2}\ \frac{\mathsf{mtch}(e, \alpha) = \bot}{[e]\varphi \xrightarrow{\alpha} \mathsf{tt}}$$

Fig. 2. The Logic and its Derivative Semantics

contain term variables. Formulas include truth and falsehood, tt and ff, conjunctions, $\varphi \,\&\, \psi$, modal necessities, $[e]\varphi$, maximal fixpoints (for recursive properties), $\mathsf{max}\, X.\, \varphi$, and conditionals to reason about data, $\mathsf{if}\, b\, \mathsf{then}\, \varphi\, \mathsf{else}\, \psi$. Free term variables in a subformula φ of a necessity formula, $[e]\varphi$ are *bound* by the variables used in the pattern e; similarly $\mathsf{max}\, X.\, \varphi$ is a binder for X in φ. We work up-to *alpha*-conversion of formulas.

A derivative semantics [24] for the *closed and guarded* logic formulas is given as a Labelled Transition System (LTS), defined by the transition rules in Fig. 2. It models the monitoring for violations of the *resp.* (safety) property, and assumes a set of (visible) actions $\alpha, \beta \in \textsc{Act}$ and a distinguished *silent* action, τ (we let γ range over $\textsc{Act} \cup \{\tau\}$). Visible actions represent system operations and contain values $v, u \in \textsc{Val}$, that range over either actor identifiers, $i, j, h \in \textsc{Pid}$, or generic data such as integers, $d \in \textsc{Data}$. The semantics also assumes a partial function $\mathsf{match}(e, \alpha)$ matching action patterns, e, with visible actions, α; when a match is successful, the function returns a substitution from the term variable found in the pattern to the corresponding values of the matched action, $\sigma :: \textsc{Var} \rightharpoonup \textsc{Val}$. We work up-to structural equivalence of formulas $\varphi \equiv \psi$; see rules in Fig. 2 for commutativity, associativity *etc.*

In Fig. 2, formulas tt and ff are idempotent *wrt.* external transitions and interpreted as final states (verdicts). Conditional formulas silently branch to the *resp.* subformula depending on the evaluation of the boolean expression (rTru and rFls) whereas rule rMax silently unfolds a recursive formula. Necessity formulas, $[e]\varphi$, transition only with a *visible* action, α: if the action matches the pattern, $\mathsf{mtch}(e, \alpha) = \sigma$, it transitions to the necessity subformula where the variables bound by the matched pattern are substituted with the *resp.* matched values obtained from the action, $\varphi\sigma$; otherwise, the necessity formula transitions to tt in case of a mismatch (rNc2) — see [7] for details. The rules for

conjunction formulas model the parallel execution of subformulas as described in [12]: subformulas are allowed to perform independent silent transitions (rCn2 and rCn3) but transition together for external actions, depending on their individual transitions (rCn1). Finally, rStr allows us to abstract over structurally equivalent formulas. We write $\varphi \overset{\gamma}{\Longrightarrow} \psi$ in lieu of $\varphi(\overset{\tau}{\longrightarrow})^* \overset{\gamma}{\longrightarrow} (\overset{\tau}{\longrightarrow})^*\psi$. We let $t \in \text{Act}^*$ range over *lists* of *visible* actions and write $\varphi \overset{t}{\Rightarrow} \psi$ to denote $\varphi \overset{\alpha_1}{\Longrightarrow} \ldots \overset{\alpha_n}{\Longrightarrow} \psi$ where $t = \alpha_1 \ldots \alpha_n$.

Example 2. Recall property φ from (1) of Ex. 1. Using the semantics of Fig. 2, we can express an execution leading to a violation detection for the action sequence below as:

$$\varphi \overset{i?\{\texttt{inc},5,h\}}{\Longrightarrow} (([j \triangleright h\,!\{\texttt{res}, 5+1\}]\,\varphi)\ \&\ ([\_ \triangleright h\,!\,\texttt{err}]\,\mathsf{ff}))$$

(using rules rMax and rNc1, where $\mathsf{mtch}(i?\{\texttt{inc}, x, y\}, i?\{\texttt{inc}, 5, h\}) = \{x \mapsto 5, y \mapsto h\}$)

$$\overset{j \triangleright h\,!\{\texttt{res},6\}}{\Longrightarrow} \varphi \overset{i?\{\texttt{inc},3,h'\}}{\Longrightarrow} (([j \triangleright h'\,!\{\texttt{res}, 3+1\}]\,\varphi)\,\&\,([\_ \triangleright h'\,!\,\texttt{err}]\,\mathsf{ff})) \overset{k \triangleright h'\,!\,\texttt{err}}{\Longrightarrow} \mathsf{ff} \qquad \blacksquare$$

The derivative semantics corresponds to the violation semantics of [12]: actor A with trace t violates φ, assertion $(A, t) \vDash_v \varphi$, iff φ transitions to ff along t and A can generate t.

Theorem 1 (Semantic Correspondence). $(A, t) \vDash_v \varphi \quad \textit{iff} \quad (\varphi \overset{t}{\Rightarrow} \mathsf{ff} \ \textit{and}\ A \overset{t}{\Rightarrow})$

3 Designing Runtime Adaptation Mechanisms

(Erlang) actor systems, such as those monitored for by detectEr, provide natural *units of adaptations* in terms of the individual actors themselves. We identify *two* classes of adaptation actions, namely *asynchronous* adaptations, $\mathsf{aA}(w)$, and *synchronous* adaptations, $\mathsf{sA}(w)$: they both take a list of actor references as argument — $w, r \in (\text{Pid} \cup \text{Var})^*$. In particular, whereas asynchronous adaptations may be administered on the *resp.* actors while they are executing, the synchronous ones require the adaptees' execution to be suspended for the adaptation to function correctly. Examples of asynchronous adaptations include actor (*i.e.,* process [3]) killing and actor linking/unlinking; both examples are native (and atomic) commands offered by the host language [3]. We have implemented additional adaptation actions that require a more complex sequence of operations, such as a *purge* action (it empties the messages contained in the actor's mailbox), and a *restart* action (it restarts the actor execution, emptying its mailbox and refreshing its internal state, while preserving its unique identifier); these constitute examples of synchronous adaptations that require the suspension of the *resp.* actor.

Synchronous adaptations require a mechanism for *gradually* suspending the actor executions of interest while a property is being monitored for, so that actors are in the required status when the adaptation is administered. There are many ways how one can program incremental synchronisations between the system and the monitor. We chose to piggyback on the specification scripts presented in Sec. 2 and extend necessity formulas with a synchronisation modality, $[e]^\rho\,\varphi$, where ρ ranges over either b (blocking), stating that the subject of the action (*i.e.,* an actor) is suspended if the action is matched by pattern e, or a (asynchronous), stating that the action subject is allowed to continue

executing asynchronously when the pattern e is matched. We recall that if the necessity formula $[e]\varphi$ mismatches with a trace action, its observation terminates (see ʀNc2 in Fig. 2); in our case this would also mean that the synchronous adaptation contained in the continuation, φ (for which we have been incrementally blocking actor executions) are never administered. In such cases, we provide a mechanism for *releasing* the actors blocked thus far: necessity formulas are further extended with a list of actor references, r, that denote the (blocked) actors to be released in case the necessity pattern e mismatches, $[e]^{\rho}_{r}\,\varphi$. Since adaptations in a script may be followed by further observations, we also require a similar release mechanism for adaptation actions, $\mathsf{aA}(w)_r$ and $\mathsf{sA}(w)_r$, where the actor list r is unblocked after the adaptation is administered to the actors w.

Remark 1. Although minimally intrusive, the expressivity of our mechanism for incremental synchronisation relies on what system actions can be observed (*i.e.,* the level of abstraction the system is monitored at). For instance, recall the system depicted in Fig. 1. If monitored from an external viewpoint, the communications sent from the interface actor, i, to the backend actors, j and h, are *not* visible (according to [12], they are seen as internal τ-actions). However, for observations required by properties such as (2), we would need to block actor i only *after* it sends a message to either of the backend actors—otherwise the entire system blocks. This requires observing the system at a *lower level of abstraction*, where certain τ-actions are converted into visible ones *e.g.,* the instrumentation used by detectEr allows us to observe internal actions such as function calls or internal messages sent between actors as discussed for Fig. 1. See [7] for more examples of this.

Example 3. The script extensions discussed in this section allow us to augment the adaptation script outlined in (2) from Ex. 1 as follows:

$$\varphi' \stackrel{\text{def}}{=} \mathsf{max}\ Y.\ [i?\{\mathtt{inc}, x, y\}]^{a}_{\epsilon} \\ [\mathtt{tau}(i, \_, \{\mathtt{inc}, x, y\})]^{b}_{\varepsilon} \begin{pmatrix} ([j \triangleright y\,!\,\{\mathtt{res}, x+1\}]^{a}_{\epsilon}\ Y)\ \& \\ ([z \triangleright y\,!\,\mathtt{err}]^{b}_{i}\ \mathsf{restr}(i)_{\varepsilon}\,\mathsf{prg}(z)_{i,z}\ Y) \end{pmatrix} \tag{3}$$

After *asynchronously* observing action $i\,?\{\mathtt{inc}, v, h\}$ (for some v, h pattern matched to x and y *resp.*), the monitor *synchronously* listens (modality b) for an internal communication action from i to some actor with this data, $\{\mathtt{inc}, v, h\}$, action $[\mathtt{tau}(i, \_, \{\mathtt{inc}, x, y\})]^{b}_{\varepsilon}$. If this action is observed, the subject of the action, *i.e.,* actor i, is blocked. If the subsequent action observed is an error reply, $z \triangleright h\,!\,\mathtt{err}$ (from an actor bound to z at runtime), we block actor z (again, note modality b) and start the synchronous adaptation actions, $\mathsf{restr}(i)_{\varepsilon}$ and $\mathsf{prg}(z)_{i;z}$. Note that the last adaptation action releases the two blocked actors i and z before recursing; similarly the necessity formula for the error reply releases the blocked actor i if the *resp.* action is not matched, $[z \triangleright y\,!\,\mathtt{err}]^{b}_{i}$. ■

4 A Formal Model for Runtime Adaptation

Fig. 3 describes a semantics for the extended logic with adaptations discussed in Sec. 3. Apart from the extended necessity and the asynchronous/synchronous adaptation formulas, it uses two additional constructs, $\mathsf{blk}(r)\,\varphi$ and $\mathsf{rel}(r)\,\varphi$; these are not meant to be part of the specification scripts but are used as part of the runtime syntax. Since the

Extended Logic Syntax with Adaptations and Synchronisations

$$\varphi, \psi \in \textsc{Frm} ::= \ \ldots \quad | \quad [e]^{\rho}_{w}\,\varphi \quad | \quad \mathsf{aA}(r)_w\varphi \quad | \quad \mathsf{sA}(r)_w\varphi \quad | \quad \mathsf{blk}(r)\,\varphi \quad | \quad \mathsf{rel}(r)\,\varphi$$

Monitor Transition Rules

$$\textsc{rNc1}\ \frac{\mathsf{mtch}(e,\alpha) = \sigma \qquad \mathsf{subj}(\alpha) = i}{[e]^{b}_{r}\,\varphi \xrightarrow{\ \alpha\ } \mathsf{blk}(i)\,(\varphi\sigma)} \qquad \textsc{rNc2}\ \frac{\mathsf{mtch}(e,\alpha) = \sigma}{[e]^{a}_{r}\,\varphi \xrightarrow{\ \alpha\ } \varphi\sigma}$$

$$\textsc{rNc3}\ \frac{\mathsf{mtch}(e,\alpha) = \bot}{[e]^{\rho}_{r}\,\varphi \xrightarrow{\ \alpha\ } \mathsf{rel}(r)\,\mathsf{tt}} \qquad \textsc{rAdA}\ \frac{}{\mathsf{aA}(w)_r\varphi \xrightarrow{\ \mathbf{a}(w)\ } \mathsf{rel}(r)\,\varphi} \qquad \textsc{rAdS}\ \frac{}{\mathsf{sA}(w)_r\varphi \xrightarrow{\ \mathbf{s}(w)\ } \mathsf{rel}(r)\,\varphi}$$

$$\textsc{rRel}\ \frac{}{\mathsf{rel}(r)\,\varphi \xrightarrow{\ \mathbf{r}(r)\ } \varphi} \qquad \textsc{rBlk}\ \frac{}{\mathsf{blk}(r)\,\varphi \xrightarrow{\ \mathbf{b}(r)\ } \varphi} \qquad \textsc{rCn4}\ \frac{\varphi \xrightarrow{\ \mu\ } \varphi'}{\varphi\,\&\,\psi \xrightarrow{\ \mu\ } \varphi'\,\&\,\psi}$$

System Transition Rules

$$\textsc{sNew}\ \frac{}{s \xrightarrow{\ \tau\ } s, i:\circ} \qquad \textsc{sAct}\ \frac{\mathsf{subj}(\alpha) = i \qquad \mathsf{ids}(\alpha) \subseteq \mathrm{dom}(s)}{s, i:\circ \xrightarrow{\ \alpha\ } s, i:\circ} \qquad \textsc{sAdA}\ \frac{w \subseteq \mathrm{dom}(s)}{s \xrightarrow{\ \mathbf{a}(w)\ } s}$$

$$\textsc{sBlk}\ \frac{}{s, w:\circ \xrightarrow{\ \mathbf{b}(w)\ } s, w:\bullet} \qquad \textsc{sRel}\ \frac{}{s, w:\bullet \xrightarrow{\ \mathbf{r}(w)\ } s, w:\circ} \qquad \textsc{sAdS}\ \frac{}{s, w:\bullet \xrightarrow{\ \mathbf{s}(w)\ } s, w:\bullet}$$

Instrumentation Transition Rules

$$\textsc{iAda}\ \frac{\varphi \xrightarrow{\ \mu\ } \varphi' \qquad s \xrightarrow{\ \mu\ } s'}{s \triangleright \varphi \xrightarrow{\ \tau\ } s' \triangleright \varphi'} \qquad \textsc{iTrm}\ \frac{\varphi \overset{\mu}{\not\longrightarrow} \qquad s \xrightarrow{\ \alpha\ } s' \qquad \varphi \overset{\alpha}{\not\longrightarrow} \qquad \varphi \overset{\tau}{\not\longrightarrow}}{s \triangleright \varphi \xrightarrow{\ \alpha\ } s' \triangleright \mathsf{tt}}$$

$$\textsc{iAct}\ \frac{\varphi \overset{\mu}{\not\longrightarrow} \qquad s \xrightarrow{\ \alpha\ } s' \qquad \varphi \xrightarrow{\ \alpha\ } \varphi'}{s \triangleright \varphi \xrightarrow{\ \alpha\ } s' \triangleright \varphi'} \qquad \textsc{iSys}\ \frac{s \xrightarrow{\ \tau\ } s}{s \triangleright \varphi \xrightarrow{\ \tau\ } s' \triangleright \varphi} \qquad \textsc{iMon}\ \frac{\varphi \xrightarrow{\ \tau\ } \varphi'}{s \triangleright \varphi \xrightarrow{\ \tau\ } s \triangleright \varphi'}$$

Fig. 3. A Runtime Semantics for Instrumented Properties with Adaptations

extended logic affects the system being monitored through adaptations and synchronisations, the operational semantics is given in terms of *configurations*, $s \triangleright \varphi$. In addition to closed formulas, configurations include the monitored system represented abstractly as a partial map, $s :: \textsc{Pid} \rightharpoonup \{\bullet, \circ\}$, describing whether an actor (through its unique identifier) is currently blocked (suspended), $\bullet$, or executing, $\circ$. We occasionally write $w : \bullet$ to denote the list of mappings $i_1 : \bullet, \ldots, i_n : \bullet$ where $w = i_1, \ldots, i_n$ (similarly for $w : \circ$).

To describe adaptation interactions between the monitor and the system, the LTS semantics of Fig. 3 employs four additional labels, ranged over by the variable μ. These include the asynchronous and synchronous adaptation labels, $\mathbf{a}(w)$ and $\mathbf{s}(w)$, to denote *resp.* that an asynchronous and synchronous action affecting actors w has been executed. They also include a blocking action, $\mathbf{b}(w)$, and an unblocking (release) action, $\mathbf{r}(w)$, affecting the execution of actors with identifiers in the list w.

The semantics is defined in terms of three LTSs: one for logical formulas (monitors), one for systems, and one for configurations, which is based on the other two LTSs. The LTS semantics for formulas extends the rules in Fig. 2 with the exception of those for the necessity formulas, which are replaced by rules rNc1, rNc2 and rNc3. Whereas rNc2 follows the same format as that of rNc1 from Fig. 2, the rule for *synchronous* necessity formulas, rNc1, transitions into a blocking construct, $\mathsf{blk}(i)\,\varphi$, for the subject of the action, i, in case a pattern match is successful. In case of mismatch, rNc3 transitions the necessity formula to a release construct, $\mathsf{rel}(r)\,\varphi$, with the specified release list of actors, r. Asynchronous and synchronous adaptation formulas transition with the *resp.* labels to a release construct as well (rules rAdA and rAdS), as do block and release constructs (rules rRel and rBlk). Finally, rule rCn4 allows monitor actions affecting the system, μ, to be carried out under a conjunction formula, independent of the other branch; we elide the obvious symmetric rule rCn5.

The system transition rules allow further actor spawning (sNew) but restrict actions to those whose subject is currently active, *i.e.,* unblocked $i : \circ$ (sAct). Whereas asynchronous adaptations can be applied to any actor list, irrespective of their status (sAdA), synchronous ones require the adaptees to be blocked (sAdS). Finally rules sBlk and sRel model the *resp.* actor status transitioning from active to blocked (and viceversa).

The instrumentation rules for configurations describe how system (visible) actions, α, affect the monitors and, dually, how the monitor adaptation and synchronisation actions, μ, affect the system. For instance, if the monitor instigates action μ and the system allows it, they both transition together as a silent move (iAda). Dually, if the system generates action α and the monitor can observe it, they also transition in unison (iAct); if the monitor cannot observe this action (iTrm), it terminates as formula tt. Note that both rules iAct and iTrm require the monitor not to be in a position to perform an adaptation/synchronisation action, *i.e.,* premise $\varphi \overset{\mu}{\not\longrightarrow}$; this gives *precedence* to monitor actions over system ones in our instrumentation. Rules iSys and iMon allow systems and monitors to transition independently *wrt.* τ-actions.

Example 4. Recall the adaptation formula φ' defined in (3) of Ex. 3. For the system $s = (i:\circ,\, j:\circ,\, k:\circ,\, h:\circ)$ we can model the runtime execution with adaptations:

$$s \triangleright \varphi' \overset{i?\{\mathtt{inc},1,h\}}{\Longrightarrow} \cdot \overset{\mathtt{tau}(i,k,\{\mathtt{inc},1,h\})}{\Longrightarrow} s \triangleright \mathsf{blk}(i) \begin{pmatrix} [j \triangleright h\,!\,\{\mathtt{res},2\}]^a_\epsilon\ \varphi' \ \ \& \\ [z \triangleright h\,!\,\mathtt{err}]^b_i\ \mathsf{restr}(i)_\varepsilon\, \mathsf{prg}(z)_{i,z}\, \varphi' \end{pmatrix} \tag{4}$$

$$\overset{\tau}{\longrightarrow} ((j,h,k):\circ, i:\bullet) \triangleright \left([j \triangleright h\,!\,\{\mathtt{res},2\}]^a_\epsilon\, \varphi' \,\&\, [z \triangleright h\,!\,\mathtt{err}]^b_i\, \mathsf{restr}(i)_\varepsilon \mathsf{prg}(z)_{i,z} \varphi' \right) \tag{5}$$

$$\overset{k \triangleright h\,!\,\mathtt{err}}{\longrightarrow} ((j,h,k):\circ, i:\bullet) \triangleright \mathsf{blk}(k)\ \mathsf{restr}(i)_\varepsilon\, \mathsf{prg}(k)_{i,k}\, \varphi' \tag{6}$$

$$\overset{\tau}{\longrightarrow} ((j,h):\circ, i:\bullet, k:\bullet) \triangleright \mathsf{restr}(i)_\varepsilon\, \mathsf{prg}(k)_{i,k}\, \varphi' \tag{7}$$

$$\overset{\tau}{\Longrightarrow} ((j,h):\circ, i:\bullet, k:\bullet) \triangleright \mathsf{rel}(i,k)\, \varphi' \ \overset{\tau}{\longrightarrow}\ s \triangleright \varphi' \tag{8}$$

In particular, the synchronous pattern-matches in (4) and (6) yield the runtime actor blocking constructs, that are applied (incrementally) in (5) and (7). This allows the synchronous adaptations in (8) to proceed, followed by the unblocking of the *resp.* actors. Erroneous blocking directives result in stuck synchronous adaptations (see sAdS). For instance, if we change the first blocking necessity in φ' of (3) to an asynchronous one,

$[\texttt{tau}(i, \_, \{\texttt{inc}, x, y\})]^a_\varepsilon$, it yields the execution below (φ'' is the erroneous formula):

$$s \triangleright \varphi'' \xrightarrow{i?\{\texttt{inc},1,h\}} \cdot \xrightarrow{\texttt{tau}(i,k,\{\texttt{inc},1,h\})} \cdot \xrightarrow{k \triangleright h!\texttt{err}} ((i, j, h) : \circ, k : \bullet) \triangleright \mathsf{restr}(i)_\varepsilon\, \mathsf{prg}(k)_{i,k}\, \varphi''$$

The final configuration is stuck because the synchronous adaptation on i cannot be carried out since i is *not* blocked. A similar situation is reached if a blocked actor is *released prematurely*. For instance, if we erroneously change the release list of the necessity subformula $[j \triangleright y!\{\texttt{res}, x+1\}]^a_\epsilon\, Y$ from ϵ to i, this releases i upon mismatch, interfering with adaptation actions along the other branch of the conjunction. ■

The semantics of Fig. 3 allows us to formalise configurations in an erroneous state, *i.e.*, when a monitor wants to apply synchronous adaptations that the system prohibits.

Definition 1. $\mathit{error}(s \triangleright \varphi) \stackrel{\mathit{def}}{=} \varphi \xrightarrow{\mathbf{s}(w)} \textit{ and } s \not\xrightarrow{\mathbf{s}(w)} \textit{ for some } w \in \mathrm{dom}(s)$

5 Static Type Checking

Synchronisation errors in adaptation-scripts, such as those outlined in Ex. 4 can be hard to detect by the specifier. We therefore develop a type system with the aim of assisting script construction, by filtering out the errors defined in Def. 1. It relies on the type structure defined in Fig. 4 where values are partitioned into either generic data, dat, or actor identifiers; identifiers are further divided into unrestricted, uid, and linear, lid. The type system is substructural [22], using linear types to statically track how the actor identifiers used for adaptations are blocked and released by the parallel branches (*i.e.*, conjunctions) of the *resp.* script. In fact, type checking (internally) uses a sub-category for linear identifier types, lbid, to denote a *blocked* linear identifier. Type checking works on *typed scripts*, where the syntax of Fig. 3 is extended so that the binding variables used in action patterns are annotated by the types dat, uid or lid.

Example 5. The adaptation-script (4) of Ex. 3 would be annotated as follows:

$$\varphi' \stackrel{\mathbf{def}}{=} \mathsf{max}\ Y.\ [i?\{\texttt{inc}, x : \mathsf{dat}, y : \mathsf{uid}\}]^a_\epsilon\ [\texttt{tau}(i, \_, \{\texttt{inc}, x, y\})]^b_\varepsilon \begin{pmatrix} ([j \triangleright y!\{\texttt{res}, x+1\}]^a_\epsilon\ Y)\ \&\ \\ ([z : \mathsf{lid} \triangleright y!\texttt{err}]^b_i\ \mathsf{restr}(i)_\varepsilon\, \mathsf{prg}(z)_{i,z}\, Y) \end{pmatrix} \quad (9)$$

In (9) above, pattern variables x, y and z are associated to types dat, uid and lid *resp.* ■

Our type system for (typed) adaptation-scripts is defined as the least relation satisfying the rules in Fig. 4. Type judgements take the form $\Sigma; \Gamma \vdash \varphi$ where

- Value environments, $\Gamma \in \textsc{Env} :: (\textsc{Pid} \cup \textsc{Var}) \rightharpoonup \textsc{Typ}$, map identifiers or variables to types — we let meta-variable $l \in (\textsc{Pid} \cup \textsc{Var})$ range over identifiers and variables;
- Formula environments, $\Sigma \in \textsc{LVar} \rightharpoonup \textsc{Env}$, map formula variables to value environments — they are used to analyse recursive formulas (see rules tMax and tVar).

We sometimes write $\Gamma \vdash \varphi$ in place of $\emptyset; \Gamma \vdash \varphi$. The rules in Fig. 4 assume standard *environment extensions*, (Γ, Γ'), and use *environment splitting*, $\Gamma_1 + \Gamma_2$, to distribute linearly mappings amongst two environment (see rules sU and sL in Fig. 4 — we elide

Type Structure and Type Environment Splitting

$$\mathbf{T}, \mathbf{U} \in \textsc{Typ} \;::=\; \mathsf{dat} \;\; \textit{(generic)} \;\mid\; \mathsf{uid} \;\; \textit{(unrestricted)} \;\mid\; \mathsf{lid} \;\; \textit{(linear)} \;\mid\; \mathsf{lbid} \;\; \textit{(blocked)}$$

$$\textsc{sE}\,\frac{}{\emptyset + \emptyset = \emptyset} \qquad \textsc{sU}\,\frac{\Gamma_1 + \Gamma_2 = \Gamma_3 \qquad \mathbf{T} \in \{\mathsf{dat}, \mathsf{uid}\}}{(\Gamma_1, l{:}\mathbf{T}) + (\Gamma_2, l{:}\mathbf{T}) = (\Gamma_3, l{:}\mathbf{T})} \qquad \textsc{sL}\,\frac{\Gamma_1 + \Gamma_2 = \Gamma_3 \qquad \mathbf{T} \in \{\mathsf{lid}, \mathsf{lbid}\}}{(\Gamma_1, l{:}\mathbf{T}) + \Gamma_2 = (\Gamma_3, l{:}\mathbf{T})}$$

Adaptation-Script Typing Rules

$$\textsc{tNcA}\,\frac{\Sigma; (\Gamma, \mathsf{bnd}(e)) \vdash \varphi \qquad \Sigma; \Gamma \vdash \mathsf{rel}(r)\,\mathsf{tt}}{\Sigma; \Gamma \vdash [e]^{a}_{r}\,\varphi} \qquad \textsc{tFls}\,\frac{}{\Sigma; \Gamma \vdash \mathsf{ff}} \qquad \textsc{tTru}\,\frac{}{\Sigma; \Gamma \vdash \mathsf{tt}}$$

$$\textsc{tNcB}\,\frac{\mathsf{subj}(e) = l \qquad \Sigma; (\Gamma, \mathsf{bnd}(e)) \vdash \mathsf{blk}(l)\,\varphi \qquad \Sigma; \Gamma \vdash \mathsf{rel}(r)\,\mathsf{tt}}{\Sigma; \Gamma \vdash [e]^{b}_{r}\,\varphi} \qquad \textsc{tIf}\,\frac{\Sigma; \Gamma \vdash \varphi \qquad \Sigma; \Gamma \vdash \psi}{\Sigma; \Gamma \vdash \mathsf{if}\, b\, \mathsf{then}\, \varphi\, \mathsf{else}\, \psi}$$

$$\textsc{tBlk}\,\frac{\Gamma = \Gamma', w{:}\mathsf{lid} \qquad \Sigma; (\Gamma', w{:}\mathsf{lbid}) \vdash \varphi}{\Sigma; \Gamma \vdash \mathsf{blk}(w)\,\varphi} \qquad \textsc{tRel}\,\frac{\Gamma = \Gamma', w{:}\mathsf{lbid} \qquad \Sigma; (\Gamma', w{:}\mathsf{lid}) \vdash \varphi}{\Sigma; \Gamma \vdash \mathsf{rel}(w)\,\varphi}$$

$$\textsc{tAdA}\,\frac{\Gamma = \Gamma', w{:}\mathsf{lid} \qquad \Sigma; \Gamma \vdash \mathsf{rel}(r)\,\varphi}{\Sigma; \Gamma \vdash \mathsf{aA}(w)_{r}\,\varphi} \qquad \textsc{tAdS}\,\frac{\Gamma = \Gamma', w{:}\mathsf{lbid} \qquad \Sigma; \Gamma \vdash \mathsf{rel}(r)\,\varphi}{\Sigma; \Gamma \vdash \mathsf{sA}(w)_{r}\,\varphi}$$

$$\textsc{tCn1}\,\frac{\mathsf{excl}(\varphi, \psi) = \bot \qquad \Sigma; \Gamma_1 \vdash \varphi \qquad \Sigma; \Gamma_2 \vdash \psi}{\Sigma; (\Gamma_1 + \Gamma_2) \vdash \varphi \,\&\, \psi} \qquad \textsc{tMax}\,\frac{(\Sigma, X \mapsto \Gamma); \Gamma \vdash \varphi}{\Sigma; \Gamma \vdash \mathsf{max}\, X.\, \varphi}$$

$$\textsc{tCn2}\,\frac{\mathsf{excl}(\varphi, \psi) = \langle r_\varphi, r_\psi \rangle \qquad \Sigma; \mathsf{eff}(\Gamma, r_\psi) \vdash \varphi \qquad \Sigma; \mathsf{eff}(\Gamma, r_\varphi) \vdash \psi}{\Sigma; \Gamma \vdash \varphi \,\&\, \psi} \qquad \textsc{tVar}\,\frac{\Sigma(X) \subseteq \Gamma}{\Sigma; \Gamma \vdash X}$$

Fig. 4. A Type System for Adaptation sHML scripts

the symmetric rule sR). Similar to before, we write $w{:}\mathbf{T}$ to denote the list of mappings $l_1{:}\mathbf{T}, \ldots, l_n{:}\mathbf{T}$ where $w = l_1, \ldots, l_n$. In addition to $\mathsf{subj}(e)$, the typing rules use another auxiliary function on patterns that extracts a map of type bindings, $\mathsf{bnd}(e)$. For instance, from Ex. 5, we have $\mathsf{bnd}(i?\{\mathtt{inc}, x{:}\mathsf{dat}, y{:}\mathsf{uid}\}) = x{:}\mathsf{dat}, y{:}\mathsf{uid}$.

The typing rules for asynchronous and blocking necessities are similar: tNcA extends the environment Γ with the bindings introduced by the pattern e to check that the continuation formula typechecks, $\Sigma; (\Gamma, \mathsf{bnd}(e)) \vdash \varphi$; it also checks that the resultant actor releases (in case of action mismatch) also typecheck, $\Sigma; \Gamma \vdash \mathsf{rel}(r)\,\mathsf{tt}$. Rule tNcB performs similar checks, but the continuation formula typechecking is prefixed by the blocking of the subject of the pattern, $\Sigma; (\Gamma, \mathsf{bnd}(e)) \vdash \mathsf{blk}(l)\,\varphi$. Typing for actor blocking and releasing changes the respective bindings from lid to lbid (and vice-versa) to typecheck the continuations, rules tBlk and tRel. Typechecking asynchronous adaptations requires the adaptees to be linearly typed, rule tAdA, whereas synchronous adaptations require adaptees to be linearly blocked, rule tAdS; in both cases, they consider the *resp.* released actors when typechecking the continuations, $\Sigma; \Gamma \vdash \mathsf{rel}(r)\,\varphi$.

We have two rules for typechecking conjunction formulas. Since conjunction subformulas may be executing in parallel (recall rules rCn1, rCn2, rCn3 and rCn4 from Fig. 2 and Fig. 3) rule tCn1, typechecks each subformula *wrt.* a split value environment, $\Gamma_1 + \Gamma_2$, as is standard in linear type systems. Unfortunately, this turns out to be too coarse of an analysis and rejects useful adaptation-scripts such as (9) from Ex. 5.

Example 6. The conjunction formula used in (9) from Ex. 5 has the form:

$$([j \triangleright y!\{\texttt{res}, x+1\}]^a_\epsilon\ Y)\ \&\ ([z{:}\mathsf{lid} \triangleright y!\texttt{err}]^b_i\ \mathsf{restr}(i)_\varepsilon\, \mathsf{prg}(z)_{i,z}\ Y)$$

where the subformulas are necessity formulas with *mutually exclusive* patterns *i.e.*, there is no action satisfying both patterns $j \triangleright y!\{\texttt{res}, x+1\}$ and $z{:}\mathsf{lid} \triangleright y!\texttt{err}$. In such cases, a conjunction formula operates more like an *external choice* construct rather than a parallel composition [20], where only one branch continues monitoring. ■

In order to refine our analysis, we define an approximating function $\mathsf{excl}(\varphi, \psi)$ that syntactically analyses subformulas to determine whether they are mutually exclusive or not. When this can be determined statically, it means that only one branch will continue, whereas the other will terminate, releasing the actors specified by the *resp.* necessity formulas (recall rNc3 from 3). Accordingly, $\mathsf{excl}(\varphi, \psi)$ denotes that mutual exclusion can be determined by returning a tuple consisting of two release sets, $\langle r_\varphi, r_\psi \rangle$ containing the actors released by the *resp.* subformulas when an action is mismatched. Rule tCn2 then typechecks each subformula *wrt.* the *entire* environment Γ, adjusted to take into consideration the actors release by the other (defunct) branch, *e.g.*, $\Sigma; \mathsf{eff}(\Gamma, r_\psi) \vdash \varphi$. When this cannot be determined, *i.e.*, $\mathsf{excl}(\varphi, \psi) = \bot$, rule tCn1 is used.

The rest of the typing rules are standard. *E.g.*, rule tIf approximates the analysis of the boolean condition and requires typechecking to hold for both branches.

Example 7. We can typecheck (9) *wrt.* $\Gamma = i{:}\mathsf{lid}, j{:}\mathsf{uid}$. The typesystem also rejects erroneous scripts discussed earlier. *E.g.*, for any environment, we *cannot* typecheck the erroneous script φ'' from Ex. 4 — with the necessary type annotations as in (9). Similarly, we *cannot* typecheck the script below (mentioned earlier in Ex. 4).

$$\varphi''' \stackrel{\text{def}}{=} \mathsf{max}\ Y.\ [i?\{\texttt{inc}, x{:}\mathsf{dat}, y{:}\mathsf{uid}\}]^a_\epsilon\ [\texttt{tau}(i, \_, \{\texttt{inc}, x, y\})]^b_\varepsilon \begin{pmatrix} ([j \triangleright y!\{\texttt{res}, x+1\}]^a_i\ Y)\ \& \\ ([z{:}\mathsf{lid} \triangleright y!\texttt{err}]^b_i\ \mathsf{restr}(i)_\varepsilon\, \mathsf{prg}(z)_{i,z}\ Y) \end{pmatrix}$$

φ''' differs from (9) only *wrt.* the necessity subformula $[j \triangleright y!\{\texttt{res}, x+1\}]^a_i\ Y$, which releases i when it mismatches an action. As discussed in Ex. 4, this results in a premature release of actor i, which interferes with the synchronous adaptation $\mathsf{restr}(i)_\varepsilon$ along the other branch. However, rule tCn2 detects this interference. ■

6 Dynamic Analysis of Typed Scripts

The typed adaptation-scripts of Sec. 5 need to execute *wrt.* the systems described in Sec. 4. Crucially, however, we cannot expect that a monitored system observes the type discipline assumed by the script. This, in turn, may create *type incompatibilities* that need to be detected and handled *at runtime* by the monitor.

Example 8. Recall the typed script (9) from Ex. 5. There are two classes of type incompatibilities that may arise during runtime monitoring:

- When listening for a pattern, *e.g.*, $i?\{\texttt{inc}, x : \mathsf{dat}, y : \mathsf{uid}\}$, the system may generate the action $i?\{\texttt{inc}, 5, 6\}$; matching the two would incorrectly map the identifier variable y (of type uid) to the data value 6; we call this a *type mismatch* incompatibility.
- When listening for pattern $z : \mathsf{lid} \triangleright y\,!\,\texttt{err}$, the system may generate a matching action $i \triangleright h\,!\,\texttt{err}$ mapping variable z to i. Aliasing z with i violates the linearity assumption associated with z, lid, which assumes it to be distinct from any other identifier mentioned in the script [22]; we call this an *aliasing* incompatibility.

A system that is typed *wrt.* the same type environment that (9) is typechecked with (*e.g.*, $\Gamma = i : \mathsf{lid}, j : \mathsf{uid}$ from Ex. 7) would not generate any of the incompatibilities above. ■

In the absence of system typing, our monitors need to perform *dynamic type checks* (at runtime) and *abort monitoring* as soon as a type incompatibility is detected: any violations to the type discipline assumed by the script potentially renders unsafe any adaptations specified, and should thus *not* be administered on the system. In order to perform dynamic type checks, the operational semantics of typed scripts is defined *wrt.* the type environment with which they are typechecked — together with the type annotations included for binding (value) variables in the necessity patterns, it captures the (type) assumptions the script makes on the system being monitored.

Example 9. The execution of the typed script (9) would use the type environment $\Gamma = i : \mathsf{lid}, j : \mathsf{uid}$ from Ex. 5 to determine that an action such as $i?\{\texttt{inc}, 5, i\}$ cannot be matched with pattern $i?\{\texttt{inc}, x : \mathsf{dat}, y : \mathsf{uid}\}$, as this would lead to a *type mismatch* between $y : \mathsf{uid}$ and the resulting map to $i : \mathsf{lid}$ (note the mismatching types). Conversely, matching pattern $i?\{\texttt{inc}, x : \mathsf{dat}, y : \mathsf{uid}\}$ with action $i?\{\texttt{inc}, 5, h\}$ would not only constitute a *valid match*, but also allow monitoring to *extend* the assumed knowledge of the system from Γ to $\Gamma' = (\Gamma, h : \mathsf{uid})$, where h is associated to the type of the matched pattern variable y. The extended environment Γ' would then allow the monitor to detect a type mismatch between pattern $z : \mathsf{lid} \triangleright h\,!\,\texttt{err}$ and action $h \triangleright h\,!\,\texttt{err}$. Importantly however, it also allows the monitor to also detect an aliasing violation between the same pattern and action $i \triangleright h\,!\,\texttt{err}$ — variable z cannot be mapped to i, since $i \in \mathrm{dom}(\Gamma')$. It would however allow $z : \mathsf{lid} \triangleright h\,!\,\texttt{err}$ to be matched to action $k \triangleright h\,!\,\texttt{err}$, which would (again) extend the current type environment to $(\Gamma', k : \mathsf{lid})$ using the script type association $z : \mathsf{lid}$. ■

Although safe, the mechanism discussed in Ex. 9 turns out to be rather restrictive for recursive properties (using maximal fixpoints). Note that, by alpha-conversion, the variable bindings made by a necessity formula under a fixpoint formula is *different* for every unfolding of that fixpoint formula: *e.g.*, unfolding script (9) twice yields

$$[i?\{\texttt{inc}, x : \mathsf{dat}, y : \mathsf{uid}\}]^a_\epsilon \left(\ldots [z : \mathsf{lid} \triangleright y\,!\,\texttt{err}]^b_i \ \ldots \left(\begin{array}{r} [i?\{\texttt{inc}, x' : \mathsf{dat}, y' : \mathsf{uid}\}]^a_\epsilon \\ (\ldots [z' : \mathsf{lid} \triangleright y\,!\,\texttt{err}]^b_i \ \ldots \ \varphi') \end{array} \right) \right)$$

where the outer bindings x, y, z are distinct from the inner bindings x', y', z'. More importantly, however, the scope of these bindings extends until the *next* fixpoint unfolding, and are *not used again* beyond that point, *e.g.*, x, y, z above are not used beyond the *resp.*

$$\textsc{rMax}\ \frac{}{\langle \Gamma,\, \Delta,\, \mathsf{max}\, X.\, \varphi\rangle \xrightarrow{\tau} \langle \Gamma,\, \Delta,\, \varphi[(\mathsf{clr}(X)\mathsf{max}\, X.\, \varphi)/X]\rangle}$$

$$\textsc{rClr}\ \frac{}{\langle \Gamma,\, \Delta,\, \mathsf{clr}(X)\varphi\rangle \xrightarrow{\tau} \langle \Gamma,\, \{i{:}\kappa \,\big|\, i{:}\kappa \in \Delta \wedge X \notin \kappa\},\, \varphi\rangle}$$

$$\textsc{rNc1}\ \frac{\begin{array}{c}\mathsf{mtch}(e,\alpha) = \sigma \qquad \mathsf{subj}(\alpha) = i \qquad \Gamma' = \mathsf{bnd}(e)\sigma \\ |\mathsf{lin}(\Gamma')| = |\mathsf{lin}(\mathsf{bnd}(e))| \qquad \mathrm{dom}(\Delta) \cap \mathrm{dom}(\Gamma') = \emptyset\end{array}}{\langle \Gamma,\, \Delta,\, [e_\kappa]^b_w\, \varphi\rangle \xrightarrow{\alpha} \langle (\Gamma, \Gamma'),\ (\Delta \cup \{i{:}\kappa \,\big|\, \Gamma'(i) = \mathsf{lid}\}),\ \mathsf{blk}(i)\, \varphi\sigma\rangle}$$

$$\textsc{rNc2}\ \frac{\mathsf{mtch}(e,\alpha) = \sigma \quad \Gamma' = \mathsf{bnd}(e)\sigma \quad |\mathsf{lin}(\Gamma')| = |\mathsf{lin}(\mathsf{bnd}(e))| \quad \mathrm{dom}(\Delta) \cap \mathrm{dom}(\Gamma') = \emptyset}{\langle \Gamma,\, \Delta,\, [e_\kappa]^a_w\, \varphi\rangle \xrightarrow{\alpha} \langle (\Gamma, \Gamma'),\ (\Delta \cup \{i{:}\kappa \,\big|\, \Gamma'(i) = \mathsf{lid}\}),\ \varphi\sigma\rangle}$$

$$\textsc{rCn1}\ \frac{\langle \Gamma,\, \Delta,\, \varphi\rangle \xrightarrow{\alpha} \langle \Gamma',\, \Delta',\, \varphi'\rangle \quad \langle \Gamma,\, \Delta,\, \psi\rangle \xrightarrow{\alpha} \langle \Gamma'',\, \Delta'',\, \psi'\rangle \quad \mathrm{dom}(\Delta) = (\mathrm{dom}(\Delta') \cap \mathrm{dom}(\Delta''))}{\langle \Gamma,\, \Delta,\, \varphi \,\&\, \psi\rangle \xrightarrow{\alpha} \langle (\Gamma', \Gamma''),\, \Delta' \cup \Delta'',\, \varphi' \,\&\, \psi'\rangle}$$

Fig. 5. Dynamically Typed Adaptation-Script Rules (main rules)

adaptations of the first unfolding. Thus, one possible method for allowing a finer dynamic analysis for adaptation-scripts, *esp.* relating to linearity and aliasing violations, is to employ a mechanism that keeps track of which bindings *are still in use*. In the example above, this would allow us to bind k twice — once with z during the first iteration, and another time with z' — safe in the knowledge that by the time the second binding occurs (z'), the first binding (z) is not in use anymore, *i.e.*, there is no aliasing.

To implement this mechanism, our formalisation uses three additional components. First, the operational semantics for adaptation-scripts uses an extra environment, $\Delta \in \textsc{Pid} \rightharpoonup \mathcal{P}(\textsc{LVar})$, keeping track of the *recursion variables under which an identifier binding is introduced* by associating that identifier to a set of formula variables, $\kappa \in \mathcal{P}(\textsc{LVar})$. Environment Δ keeps track of the linear identifiers that are currently in use. Second, to facilitate updates to environment Δ, the patterns in necessity formulas are *decorated by sets of formula variables*, denoting their *resp.* recursion scope: *e.g.*, formula $\mathsf{max}\, X.\, [e]^\rho_r \ldots \mathsf{max}\, Y.\, [e']^{\rho'}_w\, \mathsf{ff}$ is decorated[3] as $\mathsf{max}\, X.\, [e_{\{X\}}]^\rho_r \ldots \mathsf{max}\, Y.\, [e'_{\{X,Y\}}]^{\rho'}_w\, \mathsf{ff}$. Third, the runtime syntax uses an additional construct, $\mathsf{clr}(X)\varphi$, when unfolding a recursive formula: the new runtime construct demarcates the *end of an unfolding* and, upon execution, removes all identifier entries in Δ with X in their *resp.* set of formula variables so as to record that they are not in use anymore.

Fig. 5 describes the main transition rules for typed adaptation-scripts, defined over triples $\langle \Gamma, \Delta, \varphi\rangle$. Together with the system and instrumentation rules of Fig. 3 (adapted to triples $\langle \Gamma, \Delta, \varphi\rangle$), they form the complete operational semantics. By contrast to the rule in Fig. 2, rMax in Fig. 5 unfolds a recursive formula to one prefixed by a clear construct, $\mathsf{clr}(X)\mathsf{max}\, X.\, \varphi$. Rule rClr removes all entries in Δ containing X. The new version of rNc1 in Fig. 5 implicitly checks for type mismatch incompatibilities by requiring that

[3] Decoration is easily performed through a linear scan of the script.

the environment extension, (Γ, Γ'), is still a map — conflicting entries *e.g.*, $i : \mathsf{uid}, i : \mathsf{lid}$ would violate this condition. It also checks that the new bindings, $\mathrm{dom}(\Gamma')$ are distinct from the linear identifiers currently in use, $\mathrm{dom}(\Delta)$, as these constitute aliasing incompatibilities, and that pattern matching does not introduce aliasing for linear variables itself, *i.e.*, $|\mathsf{lin}(\Gamma')| = |\mathsf{lin}(\mathsf{bnd}(e))|$. Finally, it transitions by updating Δ accordingly. Rule RNc2 is analogous. Rule RCn1 performs similar checks *e.g.*, it ensures that linear aliasing introduced along separate branches do not overlap, $\mathrm{dom}(\Delta) = (\mathrm{dom}(\Delta') \cap \mathrm{dom}(\Delta''))$. If any of the conditions for RNc1, RNc2 and RCn1 are not satisfied, the adaptation-script blocks and is terminated in an instrumented setup using rule ITrm from Fig. 3, *i.e.*, it aborts as soon as type incompatibilities are detected.

Using a straightforward extension of Def. 1, we prove type soundness *wrt.* the dynamic semantics of typed adaptation-scripts: configurations with typed scripts (and initial Δ) never transition to an erroneous configuration (for any trace t).

Theorem 2 (Type Soudness). *Whenever* $\Gamma \vdash \varphi$ *then, for initial* $\Delta_{init} = \{i : \emptyset \mid \Gamma(i) = \mathsf{lid}\}$*:*

$$s \triangleright \langle \Gamma, \Delta_{init}, \varphi \rangle \overset{t}{\Longrightarrow} s' \triangleright \langle \Gamma', \Delta', \varphi' \rangle \quad \textit{implies} \quad \neg \mathsf{error}(s' \triangleright \langle \Gamma', \Delta', \varphi' \rangle)$$

7 Conclusion

We have designed language extensions for an RV tool monitoring actor systems, *cf.* Sec. 3. These extensions weave synchronisation and adaptation directives over behavioural specifications expressed in the tool logic. We then formalised the *resp.* behaviour of these new constructs (Fig. 3 and Fig. 5). Through the formalisation, we also identified execution errors that may be introduced by the synchronisation and adaptation directives. Subsequently, we defined a type system for assisting the construction of such adaptation-scripts, Sec. 5, and proved soundness properties for it, Thm. 2. We conjecture that our techniques and methodologies are generic enough to be applied, at least in part, to other RA extensions of existing RV logics and tools.

Related Work: Perhaps the closest work to ours is [26], where an extension to the logic LTL called A-LTL is developed so as to describe properties of self-adaptive systems. In [13] the authors also implement an RV tool that checks for these adaptation properties at runtime. A crucial difference between this work and ours is that in [26, 13] systems are assumed to be self-adaptive already; by contrast, we take (normal) systems and *introduce* degrees of adaptation through monitoring. We also spend substantial effort contending with the specific issue of partial monitor synchronisation in the context of inherently asynchronous (actor) systems.

In [4], the authors explore an interplay between static and dynamic typechecking in a message passing setting through monitoring. This framework of synthesising monitors from (session) types is further extended in [8] to carry out degrees of adaptations for security purposes. No type checking is carried out on the synthesised monitors in either of these works. Runtime Adaptation through monitoring are also explored in [23, 19] for C programs to attain "failure-oblivious computing" that can adapt to errors such as null-dereferencing through a technique called reverse shepherding. Again, no static analysis is performed on the monitors themselves. Finally, in [6, 5], the authors extend

Aspect-J with dependent advices, and subsequently perform static analysis on these RV scripts (using typestates) in order to determine optimisations in monitor instrumentations. However, the static analysis does not consider aspects relating to monitor safety.

References

1. L. Aceto and A. Ingólfsdóttir. Testing hennessy-milner logic with recursion. In *FoSSaCS*, volume 1578 of *LNCS*, pages 41–55. Springer, 1999.
2. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.
3. J. Armstrong. *Programming Erlang*. The Pragmatic Bookshelf, 2007.
4. L. Bocchi, T.-C. Chen, R. Demangeon, K. Honda, and N. Yoshida. Monitoring networks through multiparty session types. In *FORTE*, volume 7892 of *LNCS*, pages 50–65, 2013.
5. E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. ICSE, pages 5–14. ACM, 2010.
6. E. Bodden and P. Lam. Clara: Partially evaluating runtime monitors at compile time. In *RV*, volume 6418 of *LNCS*, pages 74–88. Springer, 2010.
7. I. Cassar and A. Francalanza. On synchronous and asynchronous monitor instrumentation for actor-based systems. In *FOCLASA*, volume 175, pages 54–68, 2014.
8. I. Castellani, M. Dezani-Ciancaglini, and J. A. Pérez. Self-adaptation and secure information flow in multiparty structured commun.: A unified perspective. In *BEAT*, pages 9–18, 2014.
9. F. Cesarini and S. Thompson. *ERLANG Programming*. O'Reilly, 1st edition, 2009.
10. C. Colombo, A. Francalanza, and R. Gatt. Elarva: A monitoring tool for erlang. In *RV*, volume 7186 of *LNCS*, pages 370–374. Springer, 2011.
11. Y. Falcone, M. Jaber, T.-H. Nguyen, M. Bozga, and S. Bensalem. Runtime verification of component-based systems. In *SEFM*, volume 7041 of *LNCS*, pages 204–220. Springer, 2011.
12. A. Francalanza and A. Seychell. Synthesising Correct concurrent Runtime Monitors. *Formal Methods in System Design (FMSD)*, pages 1–36, 2014.
13. H. Goldsby, B. Cheng, and J. Zhang. AMOEBA-RT: Run-time verification of adaptive software. In *MSE*, volume 5002 of *LNCS*, pages 212–224. Springer, 2008.
14. P. Haller and F. Sommers. *Actors in Scala*. Artima Inc., USA, 2012.
15. F. Irmert, T. Fischer, and K. Meyer-Wegener. Runtime adaptation in a service-oriented component model. SEAMS, pages 97–104. ACM, 2008.
16. M. A. Kalareh. *Evolving Software Systems for Self-Adaptation*. PhD thesis, University of Waterloo, Ontario, Canada, 2012.
17. S. Kell. A survey of pract. software adaptation techniques. *J. UCS*, 14:2110–2157, 2008.
18. D. Kozen. Results on the propositional μ-calculus. *TCS*, 27:333–354, 1983.
19. F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic runtime error repair and containment via recovery shepherding. *SIGPLAN Not.*, 49:227–238.
20. R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.
21. P. Oreizy, N. Medvidovic, and R. N. Taylor. Runtime software adaptation: Framework, approaches, and styles. ICSE Companion, pages 899–910. ACM, 2008.
22. B. C. Pierce, editor. *Advanced Topics in Types and Prog. Languages*. MIT Press, 2005.
23. M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and W. Beebee. Enhancing Availability & Security through Failure-oblivious Computing. OSDI, pages 303–316. USENIX, 2004.
24. G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Eng.*, 12:151–197, 2005.
25. K. Sen, A. Vardhan, G. Agha, and G. Roşu. Efficient decentralized monitoring of safety in distributed systems. *ICSE*, pages 418–427, 2004.
26. J. Zhang and B. H. Cheng. Using temporal logic to specify adaptive program semantics. *JSS*, 79:1361 – 1369, 2006.