On Implementing a Monitor-Oriented Programming Framework for Actor Systems

Ian Cassar and Adrian Francalanza

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

Abstract. We examine the challenges of implementing a framework for automating Monitor-Oriented Programming in the context of actor-based systems. The inherent modularity resulting from delineations induced by actors makes such systems well suited to this style of programming because monitors can surgically target parts of the system without affecting the computation in other parts. At the same time, actor systems pose new challenges for the instrumentation of the *resp.* monitoring observations and actions, due to the intrinsic asynchrony and encapsulation that characterise the actor model. We discuss a prototype implementation that tackles these challenges for the case of Erlang OTP, an industry-strength platform for building actor-based concurrent systems. We also demonstrate the effectiveness of our Monitor-Oriented Programming framework by using it to augment the functionality of a third-party software written in Erlang.

1 Introduction

Monitor-Oriented Programming (MOP) [9, 10] (also termed *monitoring* [40, 23]), is a code design principle advocating for the separation of concerns between the core functionality of a system and ancillary functionality that deals with aspects such as safety, security, reliability and robustness. MOP organises code in a layered onion-style architecture where the innermost core consists of the plain-vanilla system, and the outer layers are made up of *monitors* — software entities that observe the execution of the inner layers and react to these observations. Monitor actions typically include basic notifications of detected behaviour (to outer layers), the suppression of inner-layer observable behaviour, the filtering of stimuli coming from outer layers, and adaptation actions that affect the structure and future behaviour of the inner layers.

Software development and maintenance can benefit from MOP in various ways. For instance, MOP facilitates an *incremental deployment* strategy where outer layers may be added at a later stage, which may improve the time-to-market of a development process (*e.g.*, in the Simplex Architecture [42], monitoring was proposed as an automated method for upgrade-control systems). Arguably, this also fits better with real-world development processes, where requirements often become apparent at later stages of development. Monitoring may also be used as a means of software *customisation*, where every deployed system instance comes with its own auxiliary requirements in terms of security practices, privacy policies and robustness requirements that are handled by dedicated monitors [40, 16]. MOP is also used as a discipline for augmenting systems with a *last line of defense*, so as to improve execution correctness and robustness. For instance, they can shield the inner layers by filtering harmful external stimuli

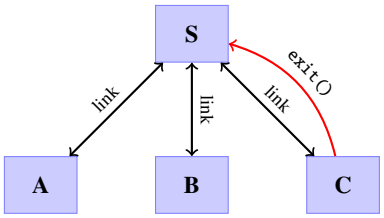

Fig. 1. Erlang actor linking

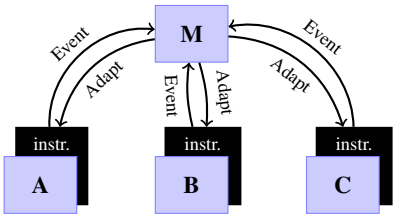

Fig. 2. Proposed monitoring extension

[6], or steer the execution of the inner system to remain within its *"stability envelope"*, from where a system can be controlled using safe and well-understood procedures [9]. In fact, monitors are the main mechanism used in formal techniques such as Runtime Verification (RV) [33, 10] and Security/Edit Automata [40, 34].

A restricted flavour of MOP is already used extensively in a number of actor-based technologies for building reactive systems, such as the Erlang [8] and Scala [27] programming languages, and the AKKA concurrency framework for Java [1]. In particular, these actor systems — collections of self-contained, asynchronously-executing, interacting processes called actors — are typically organised in hierarchical fashion, where *supervisor* actors monitor other actors at a lower layer through a mechanism called *process linking*. In the example of Fig. 1(left) the supervisor actor S is linked to three actors A, B and C; when either child actor fails, a special `exit()` notification is sent to S who is set to *trap* these *exit* messages[1] and react to them [8, 35]. Common coding practices for such technologies then advocate for the *fail-fast* design pattern, whereby inner-layer actors should focus on the core functionality of the system and not engage in defensive programming that attempts to anticipate and handle errors locally [8]; instead, actors should fail as soon as such errors are encountered, so as to allow their abnormal termination to be detected and handled by the *resp.* supervisor monitors. Once a (process) failure is detected, a supervisor may react in a number of ways: in the case of the Erlang language, a supervisor may reinstate the failed actor or replace it by a *"limp-home"* surrogate actor, terminate other actors at the same layer that are potentially *"infected"* by the error, or even fail themselves so as to allow the abnormal termination notification to percolate to monitors in outer layers that are better equipped to handle the error.

In [7], the authors propose an abstract formal model for extending this mechanism (based on supervision trees and process linking) to a more comprehensive MOP model:

1. They extend the events that are monitored for, from mere (actor) failures to sequences of actor events that include message communication and actor spawning. As depicted in Fig. 2, this allows monitors to react to a wider range of behaviour and take preemptive action *before* actors fail. As is often the case in MOP frameworks [37, 10], the authors use a formal logic to rigorously specify the actor behaviour of interest to the monitor, namely the logic presented in [26] and studied in [24].
2. They propose a range of adaptation actions that a monitor may take in response to some observed behaviour, but also argue that for such adaptations to be effective,

[1] Setting the `trap_exit` flag to false causes linked actors to fail upon receiving an exit message.

fine-tuned synchronisations between the monitor and a subset of the actors are required. Thus, they define language extensions to the logic of [26, 24] that permit the specification of *synchronisation strategies* and develop (sound) type-based analysis techniques to identity erroneous synchronisation procedures.

In this paper, we follow up on this work and study *implementability* aspects of the formal model proposed in [7]. In particular, we focus on one representative actor-based technology — the Erlang platform [8, 35] — and identify concrete instances of monitorable events and adaptation actions that are useful to MOP in such a setting. We then study the feasibility of such adaptation actions, together with the implementability of the synchronisation mechanisms designed in [7] *wrt.* the constraints of the runtime environment of the platform. In fact, we show that we can build a tool that *fully automates* the synthesis of monitors observing and reacting to the actor behaviour specified in the extended logic of [7]. Finally, we demonstrate the effectiveness and utility of the implemented framework by augmenting ancillary robustness functionality of a third-party software through our MOP framework.

To our knowledge, this is the first prototype implementation of a MOP framework for actor systems that allows programmers to add functionality in an incremental and disciplined manner through layers of monitors (implemented as actors themselves). Although the modular nature of actor-based systems facilitates the delineation of monitoring analysis and actions to a target subset of the system, the model poses new challenges to MOP. In particular, the encapsulated nature of actor state (as defined by formal models such as [2, 3] and attested by the Erlang implementation [35]) makes it hard for the monitor to access and change it. In addition, the asynchronous nature of actor executions complicates the task of synchronising observed behaviour with timely administration of monitor actions. In fact, our work appears to be one of the first to introduce synchronous monitoring atop an inherently asynchronous computing platform.

The rest of the paper is structured as follows. Sec. 2 reviews the logic used for specifying the monitor behaviour for our MOP framework. Subsequently, in Sec. 3 and Sec. 4 we discuss the implementation challenges for building an actor-centric MOP framework for this logic. Sec. 5 validates this framework by using it to administer MOP extensions to a third-party actor-based system. Sec. 6 discusses related work and concludes.

2 Monitor Specification Language

We adopt the specification language of [7] to describe monitor behaviour in our study, restated here as the abstract syntax of Fig. 3. There are mild cosmetic changes reflecting the syntax used in the implementation presented in this paper: *e.g.*, the guard constructs `[`p`]` `rel` $\vec{v}$`.` c and `*[`p`]` `rel` $\vec{v}$`.` c in Fig. 3 correspond to the *resp.* necessity formulas $[p]^{a}_{\vec{v}}\, c$ and $[p]^{b}_{\vec{v}}\, c$ of the formal logic (in [7], the qualifiers a and b differentiate between asynchronous (a) and blocking (b) pattern matching), and the termination constructs `flag` and `end` correspond to the *resp.* logic formulas ff and tt. In spite of these syntactic changes, the construct semantics is identical to that in [7].

The logic is defined over streams of visible events, α, generated by the monitored system made up of *actors* — independently-executing processes that are uniquely-identifiable by a process identifier, have their own local memory, and can either spawn

$c, d \in \text{SPEC} ::=$	$\texttt{flag}$	(detect)	$\mid \texttt{end}$	(terminate)
	$\mid\ c \,\&\, d$	(conjunction)	$\mid \texttt{if}\, b\, \texttt{then}\, c\, \texttt{else}\, d$	(branch)
	$\mid\ \texttt{rec}\, X.c$	(recursion)	$\mid X$	(recursive call)
	$\mid\ \texttt{[}p\texttt{]}\, \texttt{rel}\, \vec{v}.\ c$	(guard)	$\mid \texttt{*[}p\texttt{]}\, \texttt{rel}\, \vec{v}.\ c$	(blocking guard)
	$\mid\ A\texttt{(}x\texttt{)}\, \texttt{rel}\, \vec{v}.\ c$	(asyn. adaptation)	$\mid S\texttt{(}x\texttt{)}\, \texttt{rel}\, \vec{v}.\ c$	(sync. adaptation)

Fig. 3. Monitor Specification Syntax

other actors or interact with other actors in the system through asynchronous messaging; we use $i, j, h \in \textsc{Pid}$ to denote the unique identifiers. For the Erlang implementation we discuss in this paper, events monitored include the sending of messages, $i > j\ !\ v$, (containing the value v from actor with identifier i to actor j), the receipt of messages, $i\ ?\ v$, (containing the value v received by actor i), function calls, $\texttt{call(}i, \{m, f, l\}\texttt{)}$, (at actor i for function f in module m with argument list l) and function returns, $\texttt{ret(}i, \{m, f, a, v\}\texttt{)}$ (at actor i for function f in module m with argument arity a and return value v). Event patterns, $p, q \in \textsc{Pat}$, follow a similar structure to that of events, but may contain term variables $x, y, z \in \textsc{Var}$ (in place of values) that are bound to values $v, u \in \textsc{Val}$ (where $\textsc{Pid} \subseteq \textsc{Val}$), at runtime through pattern matching (we use $\vec{v}$ to denote lists of values).

Example 1. The pattern $x > j\ !\ \{y\texttt{,true}\}$ describes an output event from an *arbitrary* actor x to a *specific* actor j, carrying a tuple value where the first item y is unspecified but the second item must be the value $\texttt{true}$. It can match with the event $i > j\ !\ \{\texttt{5,true}\}$ returning the substitution $\{^{i,\,5}/_{x,\,y}\}$. However, the same pattern does *not* match with either $i\ ?\ \{\texttt{5,true}\}$ (different type of event) or $i > h\ !\ \{\texttt{5,false}\}$ (same event type but the event argument j conflicts with h, as does $\texttt{true}$ with $\texttt{false}$). ■

In addition to term variables, the abstract syntax in Fig. 3 also assumes a distinct denumerable set of *formula variables* $X, Y, \ldots \in \textsc{LVar}$, used to define recursive specifications. It is also parameterised by a set of *decidable* boolean expressions, $b, c \in \textsc{Bool}$, and the aforementioned set of event patterns. Monitor specifications include commands for flagging violations, $\texttt{flag}$, and terminating (silently), $\texttt{end}$, conjunctions, $c_1 \,\&\, c_2$, recursion, $\texttt{rec}\, X.c$, and conditionals to reason about data, $\texttt{if}\, b\, \texttt{then}\, c_1\, \texttt{else}\, c_2$. The specification syntax in Fig. 3 includes *two* guarding constructs, $\texttt{[}p\texttt{]}\, \texttt{rel}\, \vec{i}.\ c$ and $\texttt{*[}p\texttt{]}\, \texttt{rel}\, \vec{i}.\ c$, instructing the *resp.* monitor to observe system events that match pattern p, and progressing as c if the match is successful. Following [7], these constructs encompass directives for *blocking* and *releasing* actor executions, depending on the events observed. The guarding construct $\texttt{*[}p\texttt{]}\, \texttt{rel}\, \vec{i}.\ c$ is *blocking*, meaning that it *suspends* the execution the actor whose identifier is the *subject* of the event matched by the pattern (*e.g.,* actor i is the subject in the events $i > j\ !\ v$, $i < j\ ?\ v$, $\texttt{call(}i, \{m, f, l\}\texttt{)}$ and $\texttt{ret(}i, \{m, f, a, r\}\texttt{)}$). By contrast, the guarding construct $\texttt{[}p\texttt{]}\, \texttt{rel}\, \vec{i}.\ c$ does not block any actor when its pattern is matched. However, for both constructs $\texttt{[}p\texttt{]}\, \texttt{rel}\, \vec{i}.\ c$ and $\texttt{*[}p\texttt{]}\, \texttt{rel}\, \vec{i}.\ c$, pattern *mismatch* terminates monitoring, but *also* releases all the blocked actors in the list of identifiers $\vec{i}$. The syntax in Fig. 3 also specifies two adaptation constructs, $A\texttt{(}j\texttt{)}\, \texttt{rel}\, \vec{i}.\ c$ and $S\texttt{(}j\texttt{)}\, \texttt{rel}\, \vec{i}.\ c$. Both constructs instruct the monitor to admin-

ister an adaptation action (A and S) on actor j, releasing the (blocked) actors in $\vec{i}$ afterwards, then progressing as c. The only difference between these two constructs is that the adaptation in $S(j)$ `rel` $\vec{i}$`.` c, namely S, expects the target actor j to be *blocked* (*i.e.*, synchronised with the monitor) when the adaptation is administered, and must therefore be blocked by some preceding guarding construct.

Example 2. Consider the monitor script below. It instructs the monitor to analyse two output events, first from actor i and then from actor j, sent to the same destination x (which is pattern-matched and determined at runtime). If the outputted values sent are equivalent, $y{=}{=}z$, monitoring terminates. Otherwise, the monitor terminates the execution of the recipient actor x, restarts the two sender actors i and j, and recurses.

$$\texttt{rec}\ X\texttt{.*[}i > x\ \texttt{!}\ y\texttt{] rel [].*[}j > x\ \texttt{!}\ z\texttt{] rel [}i\texttt{]. if}\ y{=}{=}z\ \texttt{then end else}$$
$$\mathit{kill}(x)\ \texttt{rel [].}\ \mathit{restart}(i)\ \texttt{rel [].}\ \mathit{restart}(j)\ \texttt{rel [}i\texttt{,}\ j\texttt{].}\ X$$

The *restart* adaptation action is synchronous, requiring the actors i and j to be blocked (the *kill* adaptation is not). Therefore, the script specifies an *incremental strategy* for synchronising with actors i and j before the *resp.* adaptations are administered: matching with pattern $i > x\ \texttt{!}\ y$ blocks actor i, whereas pattern-matching with $j > h\ \texttt{!}\ z$ (for some actor h instantiated for x in $j > x\ \texttt{!}\ z$ by the previous match) blocks actor j. However, mismatching with pattern $j > h\ \texttt{!}\ z$ *releases* the previously blocked actor i, thereby allowing it to continue executing as normal because the monitor would terminate and the adaptation would not be administered. Importantly, if we assume that actor j's behaviour does not depend on communications from actor i, the temporary pause of actor i does not visibly affect computation since actors execute asynchronously *wrt.* to one another. See [7] for a complete formal description of the synchronisation mechanism.■

3 Instrumenting Actors

In Erlang, actors limit the sharing of data by explicitly sending copies of this data to the destination actor; identifiers act as unique actor addresses. These *asynchronous* messages are received at the destination actor's *mailbox* (a message queue buffer) and can be exclusively read by this actor using pattern-matching, which retrieves the first message in the mailbox matching a specified pattern; this two-step communication mechanism allows the recipient actor to prioritize certain messages over others by potentially reading them out-of-order of arrival. Asynchronous actor execution is one of the tenets of the actor model and, in the case of Erlang, has lead to systems that are more scalable, maintainable and resilient — asynchronous actor computation is inherently modular, easier to understand in isolation, and its failure can be readily quarantined [8, 35].

By contrast, monitors (expressed as actors) require tighter synchronisations *wrt.* the execution of actors they observe. Adequate MOP would occasionally need to momentarily pause the execution of an actor — typically after observing an event generated by it — while continuing to observe behaviour generated by other (independently executing) actors; in the event that an aggregate behaviour is detected, the monitor could then either issue notifications involving the paused actor (thereby attaining timelier detections) or else administer adaptations on the paused actor. Complex adaptations consisting of multiple operations often require adaptee actors to be inactive for their correct

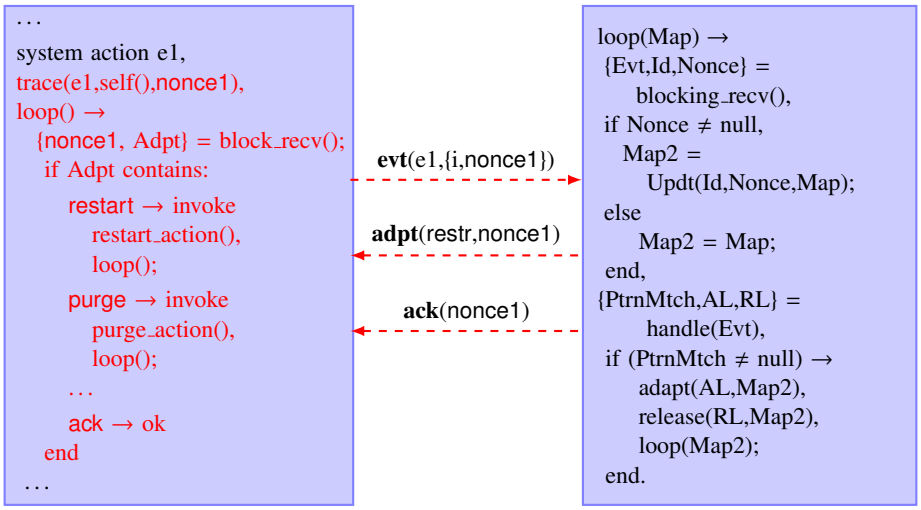

Fig. 4. The Runtime Adaptation protocol between a System Actor (left) and the Monitor (right).

administration. In our case, the specifications of Fig. 3 necessitate an incremental synchronisation mechanism whereby actors are cumulatively synchronised to (and desynchronised from) a monitor during their execution, based on the observed behaviour.

The implementability of this synchronisation mechanism hinges on the capability of externally interrupting the execution of an actor. In order to encapsulate the execution of an actor, the Erlang OTP libraries [35] (the layer of abstraction provided by the Erlang Virtual Machine) specifically limit external actor interventions to either actor killing[2] or asynchronous messaging. Neither method provides the desired functionality: (actor) killing is too coarse of an intervention, whereas sending an interrupt message to an actor does not guarantee that it will be picked up or handled adequately by the receiving actor.

Our solution was to engineer an implementation that uses an Aspect-Oriented Programming (AOP) framework to instrument injections at specific points of interest in the the monitored actors' code, and then use messaging (from the monitor) to trigger synchronisation procedures at specific stages of the monitored actor's computation; see Fig. 4, where the red code constitutes the code injected on the instrumented actors. The points of interest required by our aspect-based instrumentation are derived automatically from the patterns of the guarding constructs used in the specification scripts of Fig. 3. In particular, these patterns provide the necessary information to generate advices for AOP injections that match events at specific parts in the monitored system's source code and report back these events to the monitor for processing (first line of the injected code in Fig. 4). In the case of a blocking guard, further code is injected implementing the synchronisation protocol (injected code in Fig. 4, second line onwards).

In the actor code shown in Fig. 4 (left), specification script non-blocking guards (Fig. 3) translate into reported events with `null` nonces whereas blocking guards gen-

[2] This may be either explicit using the BIF `exit/2` or implicit through process linking[8].

erate a fresh nonce uniquely identifying a blocking session (an actor may be blocked multiple times during the course of a monitored execution). Once the monitor — the code in Fig. 4 (right) — receives an event with a non-null nonce, it creates a map entry linking the *resp.* actor ID to that nonce, and uses it to send directives during that blocking session. The monitor may send two kinds of directives: *adaptation directives*, instructing the actor to execute some predefined function (*cf.* Sec. 4), or *resumption directives* which unblock the monitored actor. After a blocking event (*i.e.,* containing a non-null nonce) is reported, the injected instrumentation code on the system-side enters a loop, waiting for directive messages from the monitor: whereas adaptation directives (*e.g.,* `restart` and `purge`) cause the monitored actor to stay in this blocking loop, the resumption directive (denoted by `ack` in Fig. 4) instructs the loop to be exited.

Remark 1. We extended an AOP Framework for Erlang [32] to carry out the necessary instrumentation (the tool did not support aspects for sends and receives). Our instrumentation thus requires an *aspect file* that specifies the actions requiring instrumentation, along with a purpose built module called `advices.erl` containing three types of advices used by the AOP injections, namely before_advice, after_advice and upon_advice advices. Function call events specified in the aspect file generate before_advice advices woven *before* the function invocation, whereas for outputs and function returns, the AOP weaves after_advice advices (after_advice are necessary for function returns, since return values are only known *after* the return of the call). For mailbox reading, the *resp.* Erlang `receive` construct may contain multiple pattern-guarded clauses *i.e.,*

$$\texttt{recieve}\ g_1\texttt{->}exp_1\texttt{;}\ g_2\texttt{->}exp_2\texttt{;}\ g_3\texttt{->}exp_3\texttt{;}\ldots\texttt{;}g_n\texttt{->}exp_n\ \texttt{end}.$$

The AOP thus weaves upon_advice advice for every guarded expression matching the message pattern defined by the receive aspects, as specified in the aspect file. *E.g.,*

$$\texttt{recieve}\ g_1\texttt{->upon\_advice(..)},exp_1\texttt{;}\ g_2\texttt{->}exp_2\texttt{;}\ g_3\texttt{->upon\_advice(..)},exp_3\texttt{;}\ldots\texttt{end}.$$

4 Implementing Adaptations

The instrumentation setup outlined in Sec. 3 enables the implementation of a wide range of adaptation actions that can be administered on individual actors using their unique actor ID. We here discuss a number of these that were successfully implemented as predefined adaptations by our prototype implementation. Following [7], these adaptations fall under two main categories, namely *asynchronous* and *synchronous* adaptations.

Asynchronous adaptations may be applied to actors whose execution need not necessarily be synchronised to that of the *resp.* effectuating monitor at the time of the adaptation. This is permissible because the *resp.* administration can execute correctly independently of the status of the adaptee's execution, typically because the execution environment provides the necessary interface for the adaptation to be effectuated *externally* from the monitor. Erlang OTP prioritises actor encapsulation and provides a *limited interface* for external interference. Accordingly, our prototype implementation offers the following predefined asynchronous adaptations: actor *killing*, using the OTP `exit()` library function, actor registering and deregistering with a global name,

using `register()` and `deregister()` OTP functions, actor memory optimisation using the OTP `garbage_collect()` function, exit message un/trapping setting using the OTP `process_flag()` function, and a composite adaptation that terminates the execution of all the actor linked to an actor (apart from itself), defined in terms of the `process_info()` and `exit()` OTP functions. These adaptations are generic in nature and agnostic to the instrumentation infrastructure discussed in Sec. 3 — in fact, they can also be used in asynchronous monitoring setups such as that of [26]. There are however scenarios where asynchronous adaptations would need to be applied to synchronised actors (*e.g.,* suspending the execution of an actor before killing it may guarantee a more timely monitor intervention); our prototype implementation allows this as well.

By contrast, synchronous adaptations require the adaptee's execution to be synchronised to that of the effectuating monitor (*i.e.,* temporarily suspended), as outlined in Sec. 3. In the case of the Erlang, one major reason for this requirement is the limited set of handles offered to externally affect the adaptee's execution — apart from the OTP functions mentioned above, messaging is the only other way of influencing an actor's execution. However, for a MOP framework to be effective, some adaptations would ideally have access an actor's internal state, even though the OTP restricts this to the owning actor exclusively. In our particular context (*i.e.,* Erlang), the only plausible method of carrying out such adaptations is that of sending a message instructing the recipient actor to carry out the adaptation itself. Note, however, that sending such a message to an actor that is not synchronised may either (*i*) be ignored by an adaptee that does not block to perform a mailbox read, or be not picked up since messages may be read out-of-order (*ii*) interfere abnormally with an actor's execution, either because the recipient actor does not know how to interpret the message directive, or because the directive-message reaches the actor at an execution point where it was expecting another type of message. The instrumentation in Sec. 3 avoids these pitfalls by forcing the actor to (autonomously) relinquish control (at specific execution points) to the observing monitor, which then sends it a message with the appropriate directive. Synchronisations are required for other reasons apart from those relating to Erlang OTP constraints. For instance, an adaptation may consist of a number of smaller actions that need to appear as one atomic action. Again, the instrumentation of Sec. 3 yields a straightforward implementation for this by suspending the adaptee's execution at the beginning and releasing it once the full list of sub-actions is completed. As a proof-of-concept, our implementation offers the pre-defined synchronous actions below:

- `purge(x)`: This adaptation requires access to (part of) the internal state of an actor (*i.e.,* its mailbox). It is implemented as a loop of non-blocking receives (using the `receive after 0` construct) consuming all the messages in the mailbox.
- `silent_kill(x)`: This composite adaptation terminates the execution of the argument actor `x` *without* informing the sibling actors to which it is linked. It is implemented by first obtaining the list of actor IDs to which it is currently linked (using `process_info(self(), links)`) and then unlinking it from this list of actors (using `unlink()`) and finally killing the adaptee once it is completely severed.
- `restart(x)`: The main complication when implementing this adaptation is that of preserving the identifier of the restarted adaptee, since other actors may be using it; a naive implementation using killing and spawning would yield a fresh iden-

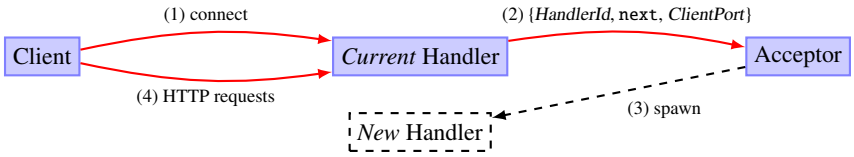

Fig. 5. Yaws client connection protocol

tifier for the restarted actor. Our implementation keeps the adaptee alive, empties its mailbox and process dictionary [8, 35] can then calls the original function with which it was spawned initially. This requires modifying spawn functions through AOP instrumentation) so as to record the actor spawn information (*i.e.,* the function spawned and its arguments) in its process dictionary; this information is then retrieved when the restart adaptation is invoked.

– `untrace(x)`: This action makes events from actor `x` unmonitorable. It extends the instrumented code of Fig. 4 with a flag indicating whether an actor should report events or not. By default, the flag is set to true whereas the action inverts it.

Remark 2. Other pre-defined adaptations can be added to the existing suite. For instance, one can define a runtime-enforcement *deletion* operation in the form of *synchronous* adaptations that intercept specific messages, using the message consumption mechanism of the `purge()` adaptation discussed above but refined for specific message patterns. One can also have an application-specific *asynchronous* adaptation that sends messages as *insertion* operations in a runtime-enforcement setup [34]. Since Erlang is higher-order and treats functions as first-class citizens, the framework can also be easily extended to handle dynamic adaptations that are not part of the predefined suite. ■

5 Augmenting Functionality through MOP

As a representative system for our evaluation we consider Yaws [43, 30], a third-party, (open source) HTTP webserver that uses actors to handle multiple client connections. For every client connection, the server assigns a dedicated (concurrent) handler that services HTTP client requests, thereby parallelising processing for multiple clients.

Fig. 5 depicts the Yaws protocol for establishing client connections. Upon creation, an *acceptor* component spawns a *connection handler* to be assigned to the next client connection. The acceptor component waits for client connection requests while the unassigned handler waits for the next TCP connection request. Clients send connection requests through standard TCP ports (1), which are received as messages in the *handler*'s mailbox. The current handler accepts these requests by reading the *resp.* message from its mailbox and (2) sending a message containing its own *Id* and the *port* of the connected client to the *acceptor*; this acts as a notification that it is now engaged in handling the connection of a specific client. Upon receiving the connection request message, the *acceptor* records the information sent by the handler and (3) spawns a *new* handler listening for future connection requests. Once it is assigned a handler, the connected client interacts *directly* with it using (4) standard HTTP requests; these normally

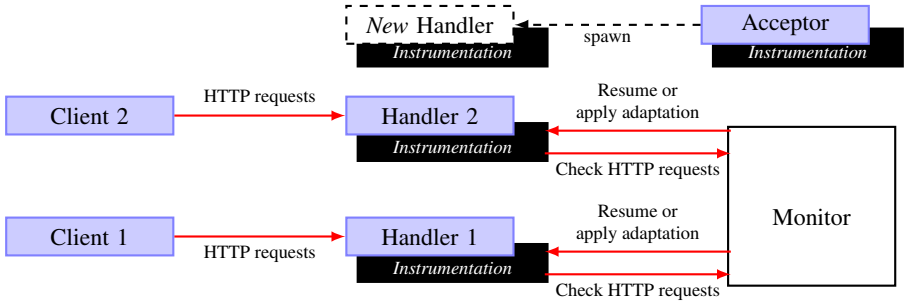

Fig. 6. Reinforced Yaws client connection protocol

consist of six (or more) HTTP headers containing the information such as the client's User Agent, Accept-Encoding and the Keep-Alive flag status. In Yaws, HTTP request information is *not* sent in one go but follows a protocol of messages: it starts by sending the `http_req`, followed by six `http_header` messages containing client information, terminated by a final `http_eoh` message. The dedicated connection handler inspects the client information received in the headers, and services the HTTP request accordingly.

To asses the effectiveness of our framework, we used our MOP tool to define Yaws extensions that augment its functionality. We here showcase one such extension, strengthening Yaws against *dot-dot-slash* attacks that exploit a directory traversal vulnerability [29]. Through additional monitor layers, the extended Yaws can detect malicious client requests (by comparing the requested URLs against a *white-list*) and take the necessary remedial actions. For our exposition, we define the monitoring script below assuming the following simplifications: (*i*) we consider a simple white-list with two files (*i.e.*, pic.png and site.html) and (*ii*) we only vet the first request of every new client. Intuitively, the script specifies that every time a client connects, and the handler actor assigned by the server receives an HTTP `GET` request for a file stored on the server, followed by 6 HTTP headers ($h1$ to $h6$) and the end-of-headers notification, then the requested file can only refer to either for pic.png or site.html. If not the *handler* is *killed*, and the *mailbox* contents of the server's *acceptor* actor is *purged*.

```
1  rec X.(
2    *[acc?{hId,next,_}] rel [].
3    [ret(hId,{yaws,do_recv,3,{ok,{http_req,'GET',{abs_path,path},_}}})] rel [acc].
4    [ret(hId,{yaws,do_recv,3,{ok,h1}})] rel [acc].
5    ...
6    [ret(hId,{yaws,do_recv,3,{ok,h6}})] rel [acc].
7    *[ret(hId,{yaws,do_recv,3,{ok,http_eoh}})] rel [acc].
8    if (path == ''/pic.png'' orelse path == ''/site.html'')
9      then untrace(hId) rel [acc, hId]. X
10     else silent_kill(hId) rel []. purge(acc) rel [acc,hId]. X
11 )
```

Through pattern-matching, the script binds the assigned handler with variable *hId* (line 2), which is then used for pattern-matching with the HTTP `GET` request, the 6 HTTP head-

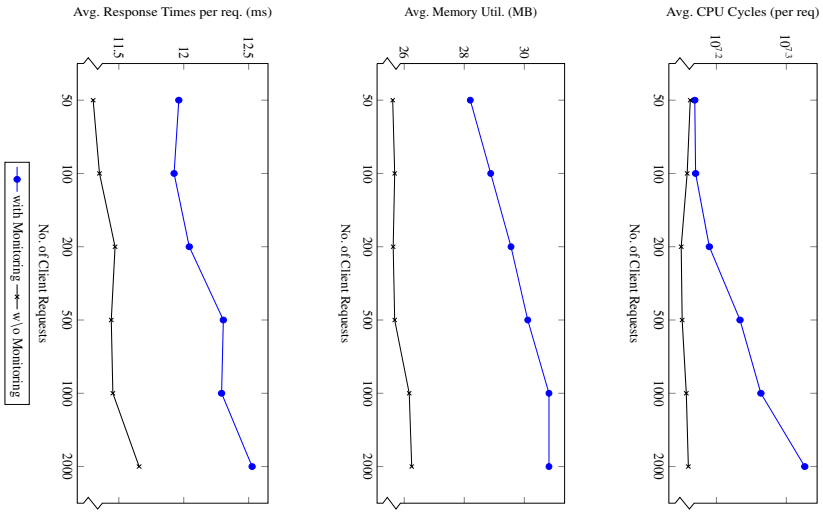

Fig. 7. MOP performance overheads

ers, and the ending header `http_eoh` (lines 3 to 7)[3]. On line 3, the file requested is bound to the variable *path* and checked against the white-list (line 8). The guard commands on lines 2 and 7 block `acc` and *hId resp.* (whereas `acc` may be known prior deployment, the Id bound to *hId* can only be determined at runtime). If the white-list check is successful, the script removes *hId* from the list of traceable actors, releases it together with `acc`, and recurs on the script variable *X* (line 9). Otherwise, a synchronous kill action is applied on *hId*, the mailbox contents of the `acc` actor are purged, and the two adaptees are released before recursing (line 10). If the HTTP message sequence is not matched at any point, the blocked actor `acc` is also released (lines 3 to 7).

From this script, our prototype implementation generates the augmented system depicted in Fig. 6. Our tool automates the necessary instrumentation required for the acceptor actor and every dynamically created handler actor. This instrumentation reports events to a monitor actor, also synthesised from the above script, which processes events and reacts by administering adaptation actions accordingly.

We also examined the overheads introduced by our MOP framework in terms of our Yaws case study. We considered a number of monitor scripts (similar to the one discussed earlier) and calculated the relative overheads when subjecting the resulting (augmented) webserver to varying client loads (measured as number of server requests) in terms of (*i*) the average CPU utilization; (*ii*) the memory required per client request; and (*iii*) the average time taken for the server to respond to batches of simultaneous client request. The experiments were carried out on an Intel Core 2 Duo T6600 processor with 4GB of RAM, running Microsoft Windows 7 and EVM version R16B03. For each script and client load, we average out three sets of readings; since the varia-

[3] These input operations are encapsulated by OTP library functions that are part of the Erlang VM. To keep the VM standard, we instead instrumented on the call returns of these functions.

tion between different monitor scripts was not substantial we again averaged the results and reported them in Fig. 7. The overheads obtained are at an acceptable level, especially since that monitoring is not merely observing the system but adding functionality (*e.g.*, at the worst level, the Memory overhead averaged at 17.4%. Fig. 7 does show a sharp increase in CPU overheads (46.7% at 2000 requests). This is in part attributed to the code serialisations introduced by the monitor synchronisations, which create inevitable bottlenecks and wasted CPU cycles when processing multiple requests (*e.g.*, in the previous script, blocking the acceptor process prohibits it from servicing other client requests in waiting). However, such steep overheads where not reflected in the average response times per client request (*e.g.*, we recorded 7.4% overheads at 2000 requests).

6 Conclusion

We present implementability results for a MOP framework targeting actor-based systems of a representative, industry-strength platform. The concrete contributions are:

1. A prototype [4] implementation that can *fully automate* the synthesis and instrumentation of monitors from formal descriptions specifying the system behaviour to be observed and the monitor actions to take in response. The implementation gives fine-grained control for non-trivial monitor actions to be carried out while imposing few system-monitor synchronisations (in accordance with the actor computational model), affecting only the sub-system targeted by the monitor actions.
2. A validation of the generality and effectiveness of the approach. We show that the functionality of third-party software can indeed be extended (with relative ease) by our framework, thereby attaining the MOP separation of concerns described in Sec. 1. Moreover, we give evidence that this can also be done feasibly, maintaining reasonable overheads when the extended system is subjected to varying stress loads.

The implementation is backed up by a formal model describing the monitor behaviour and a type system guaranteeing that synchronous monitor actions are only applied to blocked actors, as previous presented in [7]. For future work, we plan to incorporate techniques for lowering the monitor overheads (*e.g.*, code inlining [22]), and to extend our incremental synchronisation mechanisms to other monitor specification logics.

Related Work. Monitoring can be either inlined [22, 41, 11] or consolidated a separate code unit; we opted for the latter option. In multithreaded settings, inlining of inter-thread monitoring requires a *choreographed* setup [41, 25] whereas we could afford an *orchestrated* solution whereby a *centralised* monitor analyses events and issues remedial actions. Monitor inlining tends to yield lower overheads and is generally more expressive because it has full access of the system code [22]. By contrast, having monitoring as a separate unit minimally alters the code of the monitored system (all the decision branching is performed inside the monitor), is less error-prone (orchestration tends to be easier to program than monitor choreographies), allows monitor computation to be offloaded to other machines [14], and facilitates compositional analysis whereby monitors are treated in isolation [23, 24].

[4] The implementation can be downloaded from `https://bitbucket.org/casian/adapter`.

As opposed to *offline* monitoring, which assumes complete execution traces (logs) and executes *after* the system terminates its computation (*e.g.,* [18, 17, 4]), *online* monitoring executes alongside the system and has the ability to influence its computation. The prevalently used online monitoring frameworks typically employ synchronous instrumentation [31, 11, 19, 14, 5]. However, there are a few tools relying exclusively on asynchronous monitoring [13, 26, 12], which is easier to instrument since system components can be treated as black-boxes. In fact, if the monitor adaptations of Sec. 4 are limited to the asynchronous ones, then the less intrusive instrumentation setup of [26] (based on the tracing mechanism offered by the Erlang VM [35]) would suffice.

There are also frameworks offering *both* synchronous and asynchronous monitoring, such as MOP [10, 11], JPAX [28, 39] and DB-Rover [21, 20]; in these tools, the specifier can choose whether to monitor synchronously or asynchronously for a property. By contrast, we offer finer-grained control that allows a monitor to *switch* between synchronous and asynchronous modes (and vice-versa) within the same property. We are aware of one other work that studies these fine-grained monitor controls [15], proposing a model where decoupling between system and monitor executions can be inserted, together with *explicit* mechanisms for pausing the system while the lagging (asynchronous) monitor execution catches up. There are nevertheless key differences between our work and that of [15]: (*i*) they treat the monitored system as one monolithic entity whereas we have the facility of introducing synchronisations with parts of the system; (*ii*) they assume a synchronous monitoring setup and introduce asynchrony at certain points of the computation whereas, contrarily, our setting starts off with a completely decoupled system-monitor setup and introduces synchronisations when needed. Also, we study adaptations in this setting whereas [15] limit themselves to detections.

MOP frameworks that support monitor adaptations typically lean more towards giving full flexibility [37, 36] by allowing the specifier to define recovery procedures in the host language of the monitored system (*e.g.,* Java code in the case of JavaMOP[10]). Our current framework takes a different approach, offering only a finite subset of predefined adaptations that are classified into two groups (synchronous and asynchronous). Although less expressive, our approach allows for a cleaner separation between the monitor specification logic and the implementation of the system (our adaptations are implementation-agnostic abstract actions as opposed to actual Erlang code) which, in turn, facilitates the analysis of monitor scripts (*e.g.,* the type system presented in [7]).

EnforceMOP [36] is a JavaMOP extension for monitoring multithreaded computation, where they also use a centralised monitor for analyses across threads. However, as opposed to our setting, this centralised monitor does not have its own thread of control and is implemented as a static Java object that is invoked by inlined code in the *resp.* threads. Event reporting is thus necessarily synchronous, whereas our non-blocking event reporting is asynchronous and free of deadlock errors (the two-way handshake protocol of our blocking events amount to synchronous monitoring). Since they give full expressive power when defining remedial monitor actions, EnforceMOP employs additional runtime checks to avert errors introduced by the monitor itself; by contrast we offer predefined monitor actions and check for errors prior to deployment.

The implementation solutions discussed in this paper can be potentially applied to other MOP frameworks targeting asynchronous component-based systems, such as

Enterprise Service Bus (ESB) architectures [12, 38]. BusMOP [38] is an instance of the MOP suite of tools [37] where monitoring is used for component-based systems (COTS - Components Off The Shelf) made up of uniquely-identifiable devices connected to a bus. The tool treats components as black-boxes which limits monitor actions that can be taken. On the contrary, our framework adopts more of a grey-box approach for actors which allows for more powerful instrumentation mechanism and a wider range of adaptation actions. The monitoring in [38] is also completely synchronous and at a lower level of abstraction than ours (*e.g.,* they can monitor for low-level events such as memory reads and writes on the bus). The work in [12] is another example of a black-box monitor treatment of components; they study RV instrumentation alternatives on an ESB; the instrumentations considered are exclusively asynchronous and monitoring is limited to detections (*i.e.,* they do not support monitor adaptations).

References

1. Akka website. `http://www.akka.io`.
2. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.
3. G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, pages 1–72, 1997.
4. J. H. Andrews and Y. Zhang. General test result checking with log file analysis. *IEEE Trans. Software Eng.*, 29:634–648, 2003.
5. H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *FM*, volume 7436 of *LNCS*, pages 68–84. Springer, 2012.
6. N. Bielova and F. Massacci. Do you really mean what you actually enforced?: Edited automata revisited. *Int. J. Inf. Secur.*, 10(4):239–254, Aug. 2011.
7. I. Cassar and A. Francalanza. Runtime Adaptation for Actor Systems. In *RV*, volume 9333 of *LNCS*, pages 38–54. Springer, 2015.
8. F. Cesarini and S. Thompson. *ERLANG Programming*. O'Reilly, 1st edition, 2009.
9. F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *ENTCS*, pages 106–125. Elsevier, 2003.
10. F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *TACAS'05*, volume 3440 of *LNCS*, pages 546–550. Springer, 2005.
11. F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *OOPSLA*, pages 569–588. ACM press, 2007.
12. C. Colombo, G. Dimech, and A. Francalanza. Investigating instrumentation techniques for ESB runtime verification. In *SEFM*, volume 9276 of *LNCS*, pages 99–107. Springer, 2015.
13. C. Colombo, A. Francalanza, and R. Gatt. Elarva: A monitoring tool for erlang. In *RV*, volume 7186 of *LNCS*, pages 370–374. Springer, 2011.
14. C. Colombo, A. Francalanza, R. Mizzi, and G. J. Pace. polylarva: Runtime verification with configurable resource-aware monitoring boundaries. In *SEFM*, pages 218–232, 2012.
15. C. Colombo and G. J. Pace. Fast-forward runtime monitoring - an industrial case study. In *RV*, volume 7687 of *LNCS*, pages 214–228. Springer, 2012.
16. M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Self-adaptive monitors for multiparty sessions. In *PDP*, pages 688–696. IEEE Computer Society, 2014.
17. M. d'Amorim and K. Havelund. Event-based runtime verification of java programs. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

18. B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and M. Z. LOLA: Runtime Monitoring of Synchronous Systems. In *TIME*, pages 166–174. IEEE, 2005.
19. N. Decker, M. Leucker, and D. Thoma. jUnitRV - Adding Runtime Verification to jUnit. In *NFM*, volume 7871 of *LNCS*, pages 459–464. Springer, 2013.
20. N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Softw. Eng.*, 30(12):859–872, 2004.
21. D. Drusinsky. Monitoring temporal rules combined with time series. In *CAV*, volume 2725 of *LNCS*, pages 114–117. Springer, 2003.
22. U. Erlingsson. *The Inlined Reference Monitor approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2004.
23. A. Francalanza. A Theory of Monitors (Extended Abstract). In *FoSSaCS*, 2016. (to appear).
24. A. Francalanza, L. Aceto, and A. Ingolfsdottir. On Verifying Hennessy-Milner Logic with Recursion at Runtime. In *RV*, volume 9333 of *LNCS*, pages 71–86. Springer, 2015.
25. A. Francalanza, A. Gauci, and G. J. Pace. Distributed System Contract Monitoring. *JLAP*, 82(5-7):186–215, 2013.
26. A. Francalanza and A. Seychell. Synthesising Correct concurrent Runtime Monitors. *FMSD*, 46(3):226–261, 2015.
27. P. Haller and F. Sommers. *Actors in Scala*. Artima Inc., USA, 2012.
28. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *RV*, volume 55:2 of *ENTCS*, pages 200–217, 2001.
29. A. Hernandez. Yaws 1.89: Directory traversal vulnerability. Online at `http://www.exploit-db.com/exploits/15371/`, 2010. Accessed on 1/12/2015.
30. Z. Kessin. *Building Web Applications with Erlang: Working with REST and Web Sockets on Yaws*. O'Reilly Media, 2012.
31. M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *FMSD*, 24(2):129–155, 2004.
32. A. Krasnopolski. AOP for Erlang. `http://erlaop.sourceforge.net/`.
33. M. Leucker and C. Schallhart. A brief account of Runtime Verification. *JLAP*, 78(5):293 – 303, 2009.
34. J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *IJIS*, 4(1-2):2–16, 2005.
35. M. Logan, E. Merritt, and R. Carlsson. *Erlang and OTP in Action*. Manning, 2011.
36. Q. Luo and G. Roşu. EnforceMOP: A Runtime Property Enforcement System for Multi-threaded Programs. In *ISSTA*, pages 156–166, New York, NY, USA, 2013. ACM.
37. P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *STTT*, 14(3):249–289, 2012.
38. R. Pellizzoni, P. O. Meredith, M. Caccamo, and G. Rosu. Hardware Runtime Monitoring for Dependable COTS-Based Real-Time Embedded Systems. In *RTSS*, pages 481–491. IEEE, 2008.
39. G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engg.*, 12(2):151–197, 2005.
40. F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, Feb. 2000.
41. K. Sen, A. Vardhan, G. Agha, and G. Roşu. Efficient decentralized monitoring of safety in distributed systems. *ICSE*, pages 418–427, 2004.
42. L. Sha, R. Rajkumar, and M. Gagliardi. A Software Architecture for Dependable and Renewable Industrial Computing Systems. In *Int. Conference on Process Control*. IEEE, 1995.
43. S. Vinoski. Yaws: Yet another web server. *IEEE Internet Computing*, 15(4):90–94, 2011.