On Synchronous and Asynchronous Monitor Instrumentation for Actor-based systems

Ian Cassar*
Computer Science, ICT, University of Malta.
ian.cassar.10@um.edu.mt

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

We study the impact of synchronous and asynchronous monitoring instrumentation on runtime overheads in the context of a runtime verification framework for actor-based systems. We show that, in such a context, asynchronous monitoring incurs substantially lower overhead costs. We also show how, for certain properties that require synchronous monitoring, a hybrid approach can be used that ensures timely violation detections for the important events while, at the same time, incurring lower overhead costs that are closer to those of an asynchronous instrumentation.

1 Introduction

Formally ensuring the correctness of component-based, concurrent systems is an arduous task, mainly because exhaustive methods such as model-checking quickly run into state-explosion problems; this is typically caused by the multiple thread interleavings of the system being analysed, and the range of data the system can input and react to. Runtime Verification (RV) [33] is an appealing compromise towards ensuring correctness, as it circumvents such scalability issues by verifying only the *current* system execution. Runtime Enforcement (RE) [21] builds on RV by automating recovery procedures once a violation is detected so as to mitigate or rectify the effects of the violation. Together, these runtime techniques can be used as a disciplined methodology for augmenting systems with self-adaptation functionality.

Most RV and RE frameworks work by synthesising *monitors* from properties specified in terms of a formal language, and then execute these monitors in tandem with the system. Online[1] monitoring, *i.e.*, the runtime analysis of a system from the *partial* execution trace generated thus far, usually comes in two *instrumentation* flavours. In *synchronous* runtime monitoring, system trace events are forwarded to the monitor while the system *is paused*, waiting for the monitor to acknowledge back before it can continue executing (until the next trace event is generated). By contrast, in *asynchronous* monitoring, the system *does not pause* when trace events are generated; instead, events are kept in a buffer and processed by the monitor at some later stage, thereby *decoupling* the execution of the system from that of the monitor.

Both forms of instrumentation have their merits. Synchronous monitoring guarantees *timely detection* of property violations/satisfaction since the system and monitor execute in lockstep; this facilitates the *runtime enforcement* of properties, where remedial action can be promptly applied to a system waiting for a monitor response. Although asynchronous monitoring may lead to late detections, it is *less intrusive* than its synchronous counterpart. Its instrumentation is easier to carry out and does not necessarily require access to the system code prior to execution. The associated *overheads* are also assumed to be *lower* than its synchronous counterpart, and it is more of a *natural fit* for settings with inherent notions of asynchrony, such as in distributed systems. Asynchronous monitoring also poses a lower risk of

*The research work disclosed in this publication is partially funded by the *Master it!* Scholarship Scheme (Malta).

[1] By contrast, *offline* monitoring typically works on *complete* execution traces, and occasionally going back and forth along this trace during analysis.[37]

Submitted to:
FOCLASA 2014

compromising system behaviour in the eventual case of an erroneous monitoring algorithm that forgets to acknowledge back or diverges (*i.e.,* enters an infinite internal loop) since, in asynchronous monitoring the system execution is decoupled from that of the monitor.

Issues relating monitor instrumentation are particularly relevant to actor-based [4, 35, 5] component systems. Synthesising asynchronous monitors as actors is in accordance with the actor model of computation, requiring independent computing entities to execute in decoupled fashion so as to permit scalable coding techniques such as fail-fast design patterns [9]; such code organisations (using monitor actors called *supervisors*) are prevalent in actor based languages and technologies such as Erlang [7, 9], Scala [27] and AKKA [1]. However, there are cases where tighter analyses through synchronous monitoring may be required, particularly when timely detections improve the effectiveness of subsequent recovery procedures. Crucially, the appropriate monitor instrumentation needs also to incur low runtime overheads for it to be viable.

In this paper we investigate issues related to monitor instrumentation in actor-based component systems constructed using Erlang, a mature, industry-strength language used to build fault-tolerant, self-adaptive systems with high degrees of resilience [7]. As a representative Erlang system for our experiments we consider Yaws [43, 30], a high performance HTTP webserver that makes extensive use of actors to handle multiple concurrent client connections. In order to show how this study can be used to verify and enforce correct behaviour of Erlang systems, we also employ detectEr [23, 24], a component-based RV tool designed to monitor for Erlang safety properties. Within this setting:

1. we *design and implement* mechanisms for *synchronous instrumentation*; asynchronous monitoring is natively supported through mechanisms offered by the Erlang Virtual Machine (EVM) — these are presently employed by detectEr, but also other tools such as [14, 20].

2. we *quantify* the relative computational overhead incurred by synchronous and asynchronous monitoring; it is generally accepted that asynchronous monitoring incurs less overheads, but we are not aware of any studies that attempt to quantify by how much.

3. we *devise* a novel *hybrid instrumentation technique* that guarantees timely detections while incurring overheads comparable to those of asynchronous monitoring. We also *asses* the effectiveness of the technique *wrt.* the other methods of instrumentation.

4. we *integrate* the different modes of instrumentation within detectEr, allowing one to specify correctness properties with multiple violation conditions where some violations are monitored asynchronously, some are monitored synchronously, and for others monitoring switches between synchronous and asynchronous modes. Although tools for synchronous and asynchronous monitoring exists, we are not aware of any that combine the two within the same monitor execution.

Although the work in this paper focusses on detection, it lays the necessary foundations for implementing effective enforcement mechanisms that are launched as soon as violations are detected, allowing us to augments systems with self-adaptive functionality.

The rest of the paper is structured as follows. Sec. 2 briefly outlines actors in Erlang and introduces Yaws. Sec. 3 describes the logic used by the tool detectEr and shows how this logic can specify safety properties for Yaws. In Sec. 4, we devise a technique for instrumenting synchronous monitors for an actor system and asses the relative overhead costs compared to the existing asynchronous instrumentation. Sec. 5 presents a novel hybrid setup where one can specify which events are to be monitored synchronously or asynchronously. We asses the overheads incurred by the new instrumentation technique in Sec. 6. Sec. 7 concludes by outlining future and related work.

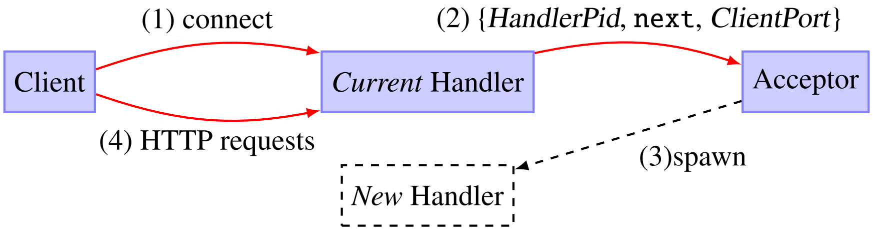


Figure 1: Yaws client connection protocol

2 Erlang Actors and the Yaws Webserver

Erlang [7] is an actor-based [5] programming language where *processes* (*i.e.*, actors) are threads of execution that are uniquely identified by a *process identifier* and own their own *local* memory. Erlang processes execute *asynchronously* to one another, interacting through *asynchronous messages*: instead of sharing data, processes explicitly send a *copy* of this data to the destination process (using the unique identifier as address); messages are received at a process *mailbox* (a form of message buffer) and can be exclusively read at a later stage by the process owning the mailbox. Processes may also *spawn* other processes at runtime, and communicate locally-known (*i.e.*, private) process identifiers as messages. Implementation wise, Erlang processes are relatively lightweight, and language coding practices recommend the use of concurrent processes wherever possible. These factors ultimately lead to systems made up of independently-executing components that are more scalable, maintainable, and use the underlying parallel architectures more efficiently [9].

At its core, Erlang is *dynamically-typed*, and function calls with mismatching parameters are typically detected at runtime. Function definitions are named and organised in uniquely-named modules. Within a particular module, there can be more than one function with the same name, as long as these names have different arities (*i.e.*, number of parameters). Thus, every function is uniquely identified by the triple *module_name:function_name:arity*.

Erlang offers a number of fault-tolerance mechanisms. Since asynchronous messages may reach a mailbox in a different order than the one intended, the (mailbox) read construct uses *pattern matching* to allow a process to retrieve the first message in the mailbox matching a pattern, possibly reading messages out of order. Erlang also offers a mechanism for *process linking*, whereby a process receives a notification message in its mailbox when a linked process fails *i.e.*, terminates abnormally. Erlang systems are often structured by the *supervisor* pattern which is built on linking: processes at the system fringes are encouraged to *fail-fast* when an error occurs (as opposed to handling the error locally), leaving error handling to linked supervisor processes [9].

2.1 YAWS: A webserver written in Erlang

Yaws[43, 30] is a high-performance, component-based HTTP webserver written in Erlang. For every client connection, the server assigns a dedicated (concurrent) handler that services HTTP client requests, thereby parallelising processing for multiple clients. Its implementation relies on the lightweight nature of Erlang processes to efficiently handle a vast number of simultaneous client connections.

The Yaws protocol for establishing client connections is depicted in Fig. 1. This protocol uses an *acceptor* component which, upon creation, spawns a *connection handler* to be assigned to the next client connection. Subsequently, the acceptor blocks waiting for messages in its mailbox, while the unassigned

$$\varphi,\psi \in \texttt{FRM} ::= \texttt{ff} \mid \varphi\texttt{\&}\psi \mid [\alpha]\varphi \mid X \mid \texttt{max}(X,\varphi) \mid \texttt{bool}\, b \texttt{=>} \varphi$$

$$\begin{aligned}
[\![\texttt{ff}]\!] &\overset{\textbf{def}}{=} \emptyset\\
[\![\varphi\texttt{\&}\psi]\!] &\overset{\textbf{def}}{=} [\![\varphi]\!] \cap [\![\psi]\!]\\
[\![\,[\alpha]\varphi]\!] &\overset{\textbf{def}}{=} \left\{ A \mid \left(A \overset{\beta}{\Longrightarrow} B \text{ and } \mathrm{match}(\alpha,\beta) = \sigma\right) \text{ implies } B \in [\![\varphi\sigma]\!] \right\}\\
[\![\texttt{max}(X,\varphi)]\!] &\overset{\textbf{def}}{=} \bigcup \{S \mid S \subseteq [\![\varphi\{X \mapsto S\}]\!]\}\\
[\![\texttt{bool}\, b \texttt{=>} \varphi]\!] &\overset{\textbf{def}}{=} \begin{cases} [\![\varphi]\!] & \text{if } b \Downarrow \mathsf{true}\\ \textsc{Actr} & \text{if } b \Downarrow \mathsf{false} \end{cases}
\end{aligned}$$

Figure 2: The Logic and its Semantics

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 *pid* 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 message, the *acceptor* unblocks, records the information sent by the handler for supervision purposes (*e.g.*, restarting the handler in case it crashes) and (3) spawns a *new* handler listening for future connection requests.

Once it is assigned a handler, the connected client then engages *directly* with it using (4) standard HTTP requests; these normally 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. 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 request and services the *resp.* HTTP request accordingly.

3 The Logic

We conduct our investigations using the component-based runtime verification tool called detectEr. In [23], the authors present a tool that synthesises concurrent monitors (as systems of Erlang processes) from a syntactic subset of the modal μ-calculus specifying *safety* properties for Erlang systems; the sublogic is called sHML [3]. In [23], these monitors *asynchronously* analyse the system so as to verify for runtime violations of the respective formulas. Actor based systems such as those constructed using Erlang typically grown and shrink in size as computation progresses.[2] Accordingly, the component-based monitors generated by detectEr are able to scale with the current size of the system being monitored.

The syntax of our logic is defined inductively using the BNF description in Fig. 2. The logic is an extension to that of [23], facilitating the expression of properties dealing with data. It is parametrised by a set of boolean expressions, $b,c \in \textsc{Bool}$, equipped with a *decidable* evaluation function, $b \Downarrow v$ where $v \in \{\mathsf{true},\mathsf{false}\}$, and a set of actions $\alpha,\beta \in \textsc{Act}$ that may universally quantify over data values. It assumes

[2] For instance, the Yaws webserver of Sec. 2.1 creates a new handler component for every new client request.

two distinct denumerable sets of *term variables*, $x,y,\ldots \in \textsc{Var}$, used in actions and boolean expressions, and *formula variables* $X,Y,\ldots \in \textsc{LVar}$, used to define recursive logical formulas; in what follows, we work up-to α-conversion of bound variables.[3] Formulas include falsity, $\texttt{ff}$, conjunctions, $\varphi \,\&\, \psi$, modal necessities, $[\alpha]\varphi$, and maximal fixpoints $\texttt{max}(X,\varphi)$ from [23]. One important extension to [23] is that actions, α, used in necessity formulas, $[\alpha]\varphi$, may contain term variables that *pattern-match* with actual (closed) actions. We also extend the logic with boolean guards, $\texttt{bool}\, b \texttt{=>} \varphi$, that may contain term variables introduced by necessity formulas. A formula $[\alpha]\varphi$ is thus a *binder* for the variables used in α in the subformula φ; similarly $\texttt{max}(X,\varphi)$ is a binder for X in φ. To improve readability, we sometimes denote term variables introduced by α in $[\alpha]\varphi$ that are not used in the subformula φ as underscores, _.

The semantics of the logic is defined over an arbitrary labelled transition system (LTS), $\langle \textsc{Actr}, \textsc{Act} \cup \{\tau\}, \rightarrow\rangle$, where $A,B \in \textsc{Actr}$ is the set of nodes denoting actor systems, $\textsc{Act} \cup \{\tau\}$ is a set of actions including a silent (internal) action τ, and $\rightarrow$ is a ternary relation of type $\textsc{Actr} \times (\textsc{Act} \cup \{\tau\}) \times \textsc{Actr}$. In [23], the authors show how Erlang programs can be given an LTS semantics and, in this paper, we conveniently adopt that semantics. In practice, however, other LTS semantics for the language, such as [40, 25] can be used instead. We write $A \xrightarrow{\alpha} B$ (*resp.* $A \xrightarrow{\tau} B$) in lieu of $(A,\alpha,B) \in\rightarrow$ (*resp.* $(A,\tau,B) \in\rightarrow$) and write $A \overset{\alpha}{\Longrightarrow} B$ to denote $A(\xrightarrow{\tau})^* \cdot \xrightarrow{\alpha} \cdot (\xrightarrow{\tau})^* B$.

Our logic semantics is presented in Fig. 2, through the denotational function $[\![-]\!] :: \texttt{FRM} \rightarrow \mathcal{P}(\textsc{Actr})$, defined inductively on the structure of the formula. The definition assumes *well-formed* formulas *i.e.*, formulas where all variables are *bound* and formula variables are *guarded* (appear under a necessity formula). We say A *satisfies* φ, denoted as $A \models \varphi$, whenever $A \in [\![\varphi]\!]$. The semantics follows that of [23]. No actor system satisfies $\texttt{ff}$, whereas actors satisfying $\varphi \,\&\, \psi$ must satisfy both φ and ψ. In our extension, the necessity formula is imbued with *pattern-matching* functionality, represented by the function match(α,β) matching a (possibly) open action α (*i.e.*, with term variables in it) with a closed action β, returning a substitution, $\sigma :: \textsc{Var} \rightharpoonup \textsc{Val}$, whenever successful.

$$\text{match}(\texttt{server}\,!\,\{x,\texttt{ack},y\}, \texttt{server}\,!\,\{5,\texttt{ack},\texttt{joe}\}) \quad = \{x \mapsto 5, y \mapsto \texttt{joe}\} \tag{1}$$

$$\text{match}(\texttt{server}\,!\,\{5,\texttt{ack},\texttt{joe}\}, \texttt{server}\,!\,\{5,\texttt{ack},\texttt{joe}\}) \quad = \{\} \tag{2}$$

$$\text{match}(\texttt{client}\,!\,\{x,\texttt{ack},y\}, \texttt{server}\,!\,\{5,\texttt{ack},\texttt{joe}\}) \quad \text{is undefined} \tag{3}$$

$$\text{match}(\texttt{server}\,?\,\{x,\texttt{ack},y\}, \texttt{server}\,!\,\{5,\texttt{ack},\texttt{joe}\}) \quad \text{is undefined} \tag{4}$$

For example, in (1) the open output action $\texttt{server}\,!\,\{x,\texttt{ack},y\}$ is successfully matched with the output action $\texttt{server}\,!\,\{5,\texttt{ack},\texttt{joe}\}$, where x and y are pattern matched with the values 5 and $\texttt{joe}$ *resp.* In (2) the two closed actions are matched (exactly), returning the empty substitution. The mismatch in (3) is due to mismatching destinations of the output actions *i.e.*, $\texttt{server}$ versus $\texttt{client}$, whereas the mismatch in (4) is because the input action $\texttt{server}\,?\,\{x,\texttt{ack},y\}$ cannot be pattern matched with actions of a different kind (*e.g.*, output). Necessity formulas $[\alpha]\varphi$ are satisfied by all actor systems A observing the condition that, *whenever* pattern-matchable actions β are performed (yielding substitution σ), the resulting actors B that are transitioned to *must* satisfy $\varphi\sigma$. Note that actors that *do not* perform any pattern-matchable actions trivially satisfy $[\alpha]\varphi$. Formula $\texttt{max}(X,\varphi)$ denotes the maximal fixpoint of the functional $[\![\varphi]\!]$; following standard fixpoint theory [42], this is characterised as the union of all post-fixpoints $S \in \mathcal{P}(\textsc{Actr})$. Guarded formulas $\texttt{bool}\, b \texttt{=>} \varphi$ equate to φ the whenever b evaluates to true but are trivially satisfied whenever b evaluates to false.

Example 3.1. Consider the formula

$$\texttt{max}(X, [\texttt{server}\,?\{\texttt{succ},x,y\}]\; [y\,!\,z]\; ((\texttt{bool}\,(z = x{+}1) \texttt{=>} X) \;\&\; (\texttt{bool}\,(z \neq x{+}1) \texttt{=>} \texttt{ff}))) \tag{5}$$

[3] detectEr renames duplicate variables accordingly during a pre-processing phase.

It states that a satisfying system repeatedly observes the condition, *i.e.*, $\texttt{max(}X\texttt{,}\ldots\texttt{)}$, that whenever it accepts an input at actor `server` with values matching the pattern $\{\texttt{succ},x,y\}$ — denoting a server request from client y for a successor computation, the `succ` message tag, for value x — followed by a reply output message sent to y with the answer z, then the answer is indeed an increment on value x. ■

Remark 3.1. Apart from input and output actions from the original tool presentation of [23], our logic extension also considers actions denoting function calls and returns, $\texttt{call(}\mathit{Pid},\{\mathit{module},\mathit{function},\mathit{values}\}\texttt{)}$ and $\texttt{ret(}\mathit{Pid},\{\mathit{module},\mathit{function},\mathit{arity},\mathit{values}\}\texttt{)}$ *resp.* These are needed in actual Erlang implementations because certain output and input actions may be abstracted away inside the function calls of system libraries, making them (directly) unobservable to the instrumentation mechanism. However, there are cases where we can still observe these actions (and the data associated with them) *indirectly*, through the calls and returns of the functions that abstract them.

3.1 Monitoring for safety properties in Yaws

The logic is expressive enough to express a number of safety properties for Yaws, including the Directory Traversal Vulnerability found in earlier versions of the software and reported on the reputable exploit-db website [29] . We here discuss a second safety property for the webserver.

If we assume the existence of a (decidable) predicate called *isMalicious*(), which can determine whether the client will engage in security-breaching activity from the 6 HTTP headers sent to the handler, we can use the logic of Fig. 2 to specify the safety property stated in (6). The property requires that, for *every* client connection established (determined from message (2) of Fig. 1 and denoted in property (6) as the action $\texttt{AcceptorPid}\,!\,\{\mathit{handID},\texttt{next},\_\}$) the following subproperty must hold: for *every* HTTP request, the respective headers communicated ($h1$ to $h6$) do not amount to a potentially security-breaching request (as determined by the predicate *isMalicious*()).

$$
\begin{array}{l}
\texttt{max(}X\texttt{,}\\
\quad\texttt{[AcceptorPid}\,!\,\{\mathit{handID},\texttt{next},\_\}\texttt{]}\\
\quad\big(X\ \texttt{\&}\\
\quad\ \ \texttt{max(}Y\texttt{,[ret(}\mathit{handID},\{\texttt{yaws},\texttt{do\_recv},3,\{\texttt{ok},\{\texttt{http\_req},\texttt{GET},\_,\_\}\}\}\texttt{)]}\\
\quad\quad\texttt{[ret(}\mathit{handID},\{\texttt{yaws},\texttt{do\_recv},3,\{\texttt{ok},h1\}\}\texttt{)]}\,\texttt{[ret(}\mathit{handID},\{\texttt{yaws},\texttt{do\_recv},3,\{\texttt{ok},h2\}\}\texttt{)]}\\
\quad\quad\texttt{[ret(}\mathit{handID},\{\texttt{yaws},\texttt{do\_recv},3,\{\texttt{ok},h3\}\}\texttt{)]}\,\texttt{[ret(}\mathit{handID},\{\texttt{yaws},\texttt{do\_recv},3,\{\texttt{ok},h4\}\}\texttt{)]}\\
\quad\quad\texttt{[ret(}\mathit{handID},\{\texttt{yaws},\texttt{do\_recv},3,\{\texttt{ok},h5\}\}\texttt{)]}\,\texttt{[ret(}\mathit{handID},\{\texttt{yaws},\texttt{do\_recv},3,\{\texttt{ok},h6\}\}\texttt{)]}\\
\quad\quad\left(\begin{array}{l}
(\texttt{bool}\,(\mathit{isMalicious}(h1,h2,h3,h4,h5,h6))\ \texttt{=>}\ \texttt{ff})\\
\texttt{\&}(\texttt{bool}\,(\neg\mathit{isMalicious}(h1,h2,h3,h4,h5,h6))\ \texttt{=>}\\
\qquad\qquad\texttt{[ret(}\mathit{handID},\{\texttt{yaws},\texttt{do\_recv},3,\{\texttt{ok},\texttt{http\_eoh}\}\}\texttt{)]}\,Y))
\end{array}\right)\\
\quad\big)\texttt{))}
\end{array}
\tag{6}
$$

The logical formula stated in (6) specifies this property by using two nested recursive formulas. The outer one, $\texttt{max(}X\texttt{,}\ldots\texttt{)}$, refers to each assigned handler by pattern-matching with the term variable *handID*, whereas the inner one, $\texttt{max(}Y\texttt{,}\ldots\texttt{)}$, uses this value to pattern match with the header term variables, *h1* to *h6*, for every iteration of the HTTP request protocol; boolean guarded formulas are then used to determine whether these HTTP requests violate the property or not. We note that, whereas the handler messages to the acceptor are observed *directly* (*i.e.*, the output action in the outer recursive formula), the client HTTP messages received by the handler have to be observed *indirectly* through the return values (of the form $\{\texttt{ok},\_\}$) of the invoked function `do_recv`.[4] Instrumentation allowing a direct observation of

[4] Defined in module `yaws` with arity 3.

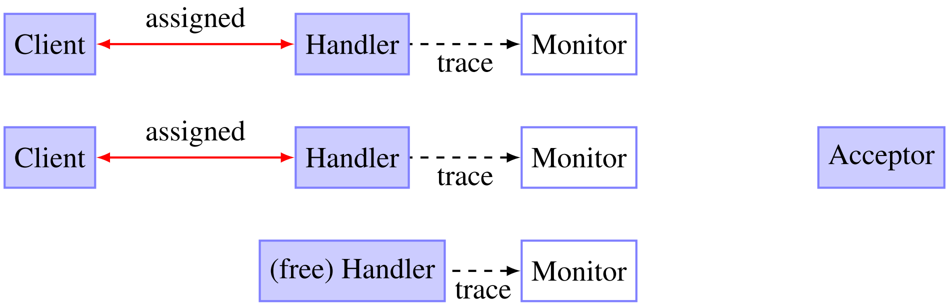


Figure 3: Component-based Monitoring

these actions is complicated by the fact that the client TCP messages are sent through functions from the `inet` Erlang library, which is part of the Erlang Virtual Machine kernel.

3.2 Component-based Monitor Generation

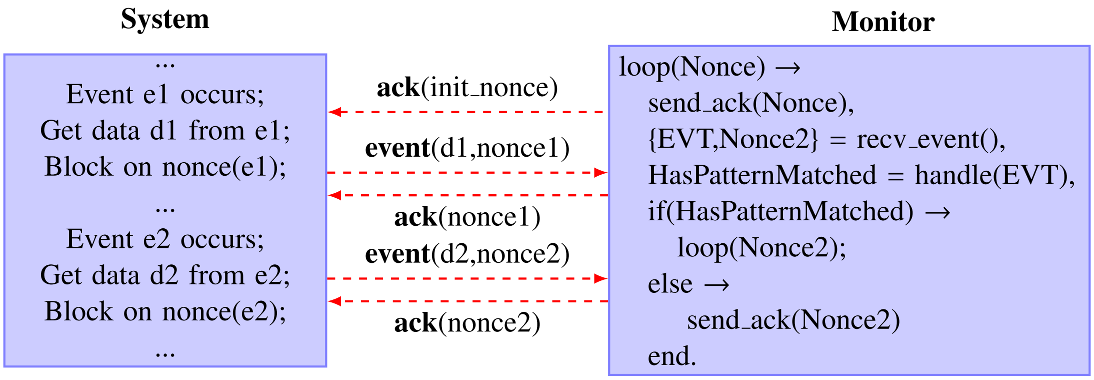
The monitors generated by detectEr are *not* monolithic, but consist of systems of concurrent (sub)monitors, each analysing different parts of a system under examination. For instance, in the case of property (6), detectEr first generates a submonitor that listens for the action $\texttt{AcceptorPid}\,!\,\{\mathit{handID},\texttt{next},\_\}$ from the unassigned (free) handler waiting on the TCP port. Once this action is detected, it spawns a new submonitor to listen for the new handler, while it continues monitoring for the handler that is now assigned to a client. Thus, after two client connections are accepted, we end up with the configuration depicted in Fig. 3 (see [23] for details). Note that the submonitors need not communicate with each other since a violation detected by one submonitor immediately means that the global property is violated.

4 Introducing Synchronous Instrumentation

In [23], the monitors generated by detectEr are exclusively asynchronous, using the tracing mechanism offered by the EVM OTP platform libraries [34] which do not require instrumentation at source-code level; instead VM directives generate trace messages for specified execution events that are sent to a specially-designated *tracer* actor. However, as was argued in Sec. 1, there are various cases where safety properties may need to be monitored synchronously.

There are a number of ways how one can layer synchronous monitoring atop of an asynchronous computational model. For instance, one can insert actual monitoring functionality *within* the sequential thread execution of each actor (in the style of [38, 11, 15]) and then have the monitoring code (scattered across independently executing actors) synchronise, as required, in a *choreographed* setup [38, 22]. Instead, we opt for an *orchestrated* solution, whereby individually monitored actors are only instrumented to report monitored actions to a (conceptually) centralised monitoring setup that receives all reported actions and performs the necessary checking.[5] There are a number of reasons for choosing such a setup. First, the instrumentation code at the system side is minimal, leaving the instrumented code close to the original. Moreover, monitoring is *consolidated* into a group of concurrent actors that are separate from the monitored system, improving manageability (*e.g.*, parts of the system may crash leaving the monitor

[5] Although conceptually centralised, the orchestrator monitor consists of independent, concurrent sub-monitors as discussed in Sec. 3.2.



```
...
Event e1 occurs;
Get data d1 from e1;
Block on nonce(e1);
...
Event e2 occurs;
Get data d2 from e2;
Block on nonce(e2);
...
```

```
loop(Nonce) →
  send_ack(Nonce),
  {EVT,Nonce2} = recv_event(),
  HasPatternMatched = handle(EVT),
  if(HasPatternMatched) →
    loop(Nonce2);
  else →
      send_ack(Nonce2)
  end.
```

Figure 4: A high level description of the synchronous event monitoring protocol.

(largely) unaffected). More importantly, it allows us to perform a like-with-like comparison with the existing asynchronous setup present in detectEr, thus obtaining a more precise comparison between the relative overheads of synchronous and asynchronous monitoring.

Fig. 4 depicts the synchronisation protocol between the system instrumented code and the monitor for two monitored events; synchrony is achieved through handshakes over asynchronous messages between the two parties. The monitoring loop starts by sending an acknowledgement message to the system, signalling it to execute until it produces the next monitored event. When this point is reached, the instrumentation code at the system side extracts the necessary data associated with the event, sends it as a trace message to the monitor and pauses by blocking on an unpause-acknowledgement message from the monitor. Since, the (acknowledgement) asynchronous messages may get reordered in transit (potentially interfering with this protocol) the instrumentation code generates a unique *nonce* for every monitored event and sends it with the event data. In return, the next time the monitor sends back an acknowledgement message, it includes this unique nonce; this allows the instrumentation code to pattern-match for (and possibly read out-of-order) mailbox inputs containing this nonce, and unblock to the corresponding acknowledgement. The monitoring loop outlined in Fig. 4 corresponds to the pattern-matching functionality required by a necessity formula $[\alpha]\varphi$. For instance, if the pattern is not matched, it acknowledges immediately to the system to continue executing and terminates the monitoring *i.e.*, command `send_ack(Nonce2)`. However, if the pattern matches, there is still a chance that a violation can be detected: it therefore delegates the acknowledgement (with the corresponding nonce) to the next part of the monitor (corresponding to φ in $[\alpha]\varphi$) *i.e.*, command `loop(Nonce2)`. Crucially, if $\varphi = \texttt{ff}$, the monitor *does not* send back the acknowledgement, blocking the offending system indefinitely while flagging the violation; in a runtime enforcement setup, the execution of a recovery procedure would substitute the violation flagging.

4.1 Instrumentation through Aspect-Oriented Weaving

The instrumentation was carried out by *extending* an Aspect Oriented Framework for Erlang [32] to add the necessary instrumentation in the system code; [32] did not support aspects for output and input events. Our aspect-based instrumentation uses a purpose built Erlang module called `advices.erl` containing the 3 type of advices used by our AOP injections. For send and function call events, the AOP weaves `before_advice` advices reporting the event data to the monitor. For function returns, the AOP weaves `after_advice` advices at the source location where the function is invoked; in this case, after advices are required since the return values are only known at that point. The weaving concerns mailbox reading events, performed through the `recieve` construct, was not as straightforward. Since a `recieve` may contain multiple pattern-matching guarded clauses

$$\texttt{recieve}\ \mathit{guard}_1\ \texttt{->}\ \mathit{expression}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{guard}_n\ \texttt{->}\ \mathit{expression}_n\ \texttt{end}$$

the AOP weaves `upon_advice` advice for each guarded expression (*i.e.*, after each `->`). At runtime, only *one* `recieve` guarded expression is triggered, at which point the necessary pattern-matched data of the event is known and can be reported to the monitor by the advice.

4.2 Preliminary results

We conducted a number of experiments to asses the relative overheads between synchronous and asynchronous monitoring. We synthesised numerous properties such as (6) for a Yaws server installation handling varying amounts of client connections and HTTP requests. The graphs obtained in Fig. 6 clearly show that synchronous monitoring incurs higher overheads than its asynchronous counterpart in terms of CPU Utilisation, memory consumption and the latencies it introduces; Fig. 6 shows substantial responsiveness degradation when handling typical loads of client requests. To rule out any gains obtained through the efficiencies of the OTP tracing platform, we also created our own version of asynchronous monitoring that uses aspect orientation (but without blocking the system); for certain measures, *e.g.*, memory consumption, the overhead discrepancies were even larger (see Fig. 6).

5 Hybrid Instrumentation

Despite the benefits of synchronous monitoring, the associated overheads obtained from our preliminary results are substantially higher so as to make it infeasible in practice. We therefore devise an alternative instrumentation strategy with the aim of guaranteeing timely violation detections while incurring lower overheads that are closer to those incurred by asynchronous instrumentation. The key insight is that, in order to attain timely detections, the instrumentation need not require the system to execute in lockstep with the monitor for *every* monitored event leading to the violation. Instead, (expensive) synchronous event monitoring can be limited to the *final* event *preceeding* a violation, letting the system execute in decoupled fashion otherwise. Intuitively, for the logic of Fig. 2 these final events and the necessity actions preceeding (directly or indirectly) a `ff` forumla.

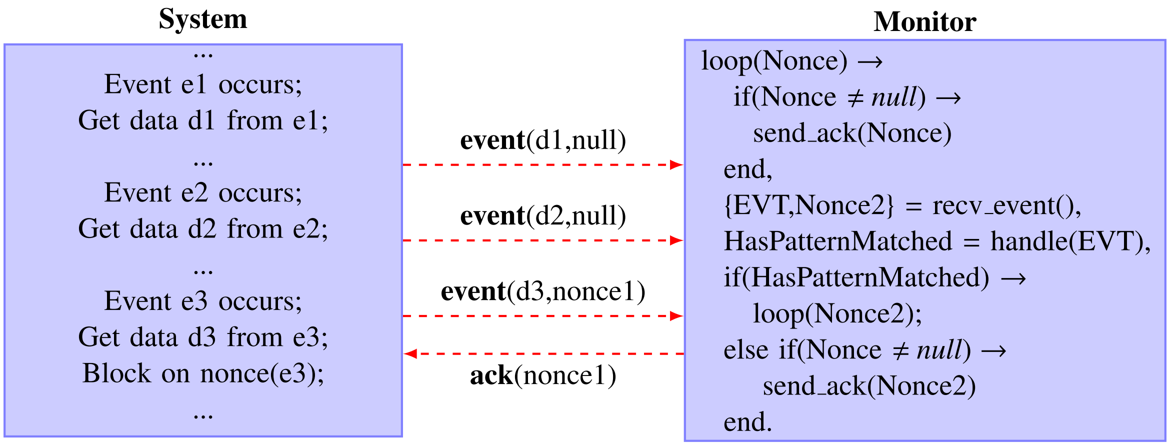
Example 5.1. Recall property (5) from Ex. 3.1. In order to synchronously detect a violation in this property, only action `[y!z]` needs to be synchronously monitored, since it precedes a `ff` forumla (interposed by a conjunction and a boolean guard). Action $\texttt{[server?\{succ,}x,y\texttt{\}]}$ can be asynchronously monitored, without affecting the timeliness of detections. ■

5.1 Logic Extensions

We extend the syntax of the logic introduced in Sec. 3 by two constructs: a *synchronous false* formula and a *synchronous necessity* formula with a semantics analogous to that of `ff` and $[\alpha]\varphi$ *resp.* (see Fig. 2)

$$\varphi,\psi \in \texttt{FRM} ::= \ldots \mid \texttt{sff} \qquad \textit{(synchronous false)} \mid [|\alpha|]\varphi \qquad \textit{(synchronous necessity)}$$

In the extended logic, formulas carry additional instrumentation information relating to how they need to be runtime-monitored: by default, all the monitoring is asynchronous, unless one specifies that a



```
...
Event e1 occurs;
Get data d1 from e1;
...
Event e2 occurs;
Get data d2 from e2;
...
Event e3 occurs;
Get data d3 from e3;
Block on nonce(e3);
...
```

```
loop(Nonce) →
   if(Nonce ≠ null) →
     send_ack(Nonce)
  end,
  {EVT,Nonce2} = recv_event(),
  HasPatternMatched = handle(EVT),
  if(HasPatternMatched) →
     loop(Nonce2);
  else if(Nonce ≠ null) →
     send_ack(Nonce2)
  end.
```

Figure 5: A high-level depiction of the hybrid monitoring protocol.

violation is to be synchronously monitored, $\texttt{sff}$, of that a particular event needs to be synchronously monitored, $\texttt{[|}\alpha\texttt{|]}\varphi$.[6]

Example 5.2. We can refine property (5) as shown below, whereby we distinguish between two kinds of violations, *i.e.*, return values z that are less than $x+1$ and return values that are greater than $x+1$, and require the latter violations to be detected in synchronous fashion.

$$\texttt{max}(X,\texttt{[server?}\{\texttt{succ},x,y\}\texttt{]}\ \texttt{[}y\,\texttt{!}\,z\texttt{]}\left(\begin{array}{l}(\texttt{bool}\,(z=x+1)\,\texttt{=>}X\,)\\ \quad\texttt{\&}\ (\texttt{bool}\,(z<x+1)\,\texttt{=>}\,\texttt{ff})\\ \quad\texttt{\&}\ \ (\texttt{bool}\,(z>x+1)\,\texttt{=>}\,\texttt{sff})\end{array}\right))\qquad\blacksquare$$

The new monitor synthesis algorithm requires a pre-processing phase to determine which events are to be synchronously monitored in order to implement a synchronous fail. For instance, formulas $[\alpha]\texttt{sff}$ and $\texttt{[|}\alpha\texttt{|]ff}$ are both monitored in the same way, in fact the pre-processing phase encodes the former into the latter. In general, however, determining which actions to synchronously monitor for implementing a synchronous fail is not as straightforward, since the $\texttt{sff}$ and the first necessity formula preceding this $\texttt{sff}$ may be interposed by intermediate formulas such as conjunctions and boolean guards (as in the property of Ex. 5.2). In such a case the compiler inspects each boolean guard and checks whether there exists at least *one* boolean guard that leads directly to a synchronous fail, i.e., $\texttt{bool}\,\texttt{b} \texttt{=>} \texttt{sff}$, and if so, the action specified in the necessity is synchronously monitored. In hybrid monitoring, both synchronous and asynchronous event monitoring require code instrumentation, as shown in Fig. 5. *Asynchronous* events inject advice functions sending a monitor message containing the event details and a *null* nonce, *without blocking* the system; upon receiving a null nonce, the monitor determines that it does not need to send an acknowledgment back to the system. *Synchronous* actions are implemented as before (see Sec. 4), where *fresh* nonces indicates the monitor that it needs to acknowledge back.

6 Evaluation

Using the extended syntax, we reformulate the security properties used for the evaluation of Sec. 4.2 and require violation detections to be synchronous *i.e.*, using $\texttt{sff}$ instead of $\texttt{ff}$. For instance, from (6) we

[6] Synchronous event monitoring can be used to engineer synchronisation points during periods where the system is not required to be immediately responsive at which point a monitor is allowed to catch up with a system execution, as in [16].

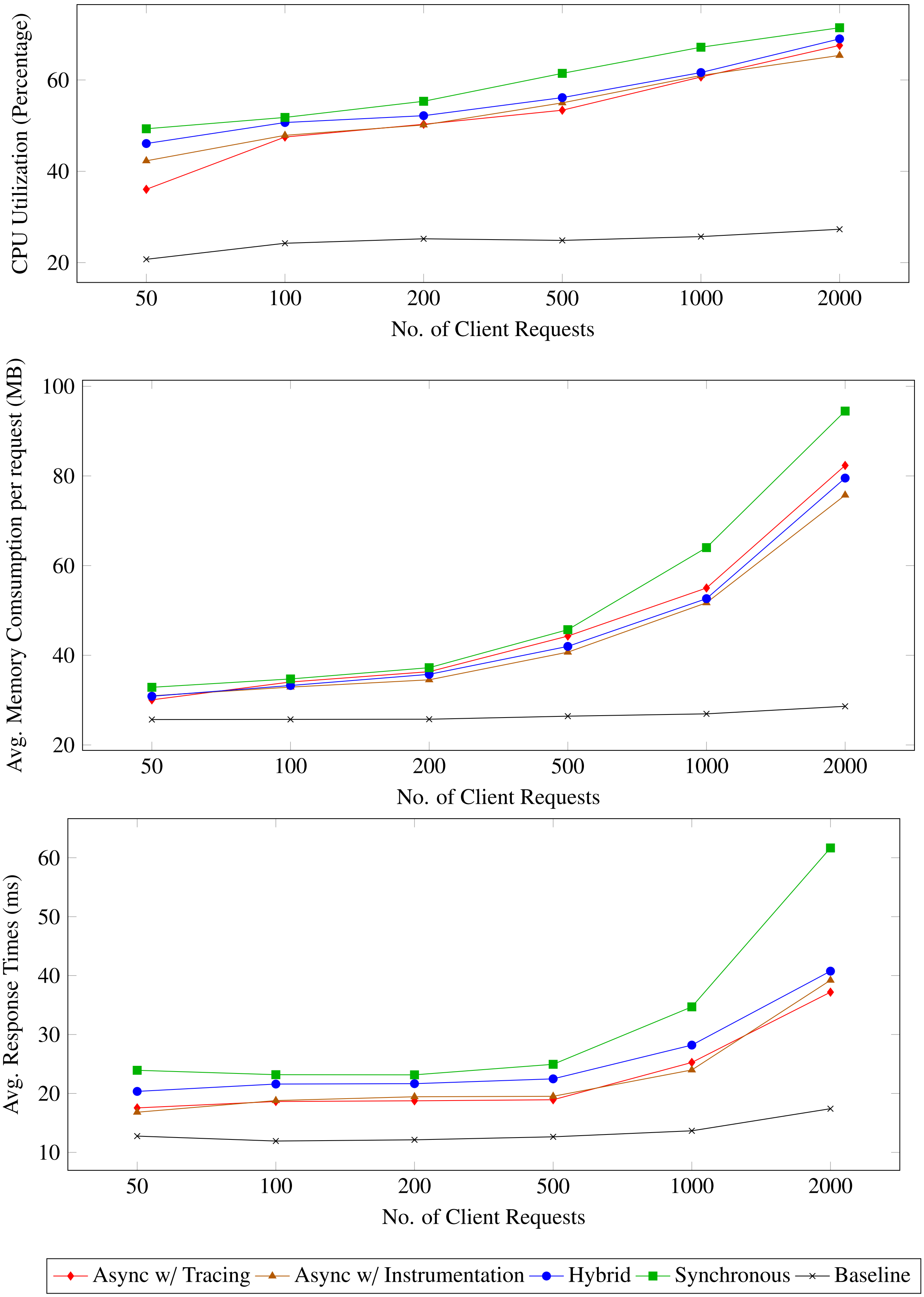


Figure 6: These graphs denote the average cpu time, memory utilization and response time when monitoring the system using different monitoring approaches.

obtain the property below.

$$\begin{array}{l}
\texttt{max}(X, \\
\quad \texttt{[|AcceptorPid!}\{\mathit{handlerPid},\texttt{next},\_\}\texttt{|]} \\
\quad \Big(X \ \&\ \texttt{max}(Y, \texttt{[ret(}\mathit{handlerPid},\{\texttt{yaws},\texttt{do\_recv},3,\{\texttt{ok},\{\texttt{http\_req},\texttt{GET},\_,\_\}\}\}\texttt{)]} \\
\qquad \texttt{[ret(}\mathit{handlerPid},\{\texttt{yaws},\texttt{do\_recv},3,\{\texttt{ok},h1\}\}\texttt{)]} \\
\qquad \vdots \\
\qquad \texttt{[ret(}\mathit{handlerPid},\{\texttt{yaws},\texttt{do\_recv},3,\{\texttt{ok},h6\}\}\texttt{)]} \\
\qquad \left(\begin{array}{l}
(\texttt{bool}\,(\mathit{isMalicious}(h1,h2,h3,h4,h5,h6)) \texttt{=>}\ \texttt{sff}\,) \\
\&(\texttt{bool}\,(\neg\mathit{isMalicious}(h1,h2,h3,h4,h5,h6)) \texttt{=>} \\
\qquad\quad \texttt{[ret(}\mathit{handlerPid},\{\texttt{yaws},\texttt{do\_recv},3,\{\texttt{ok},\texttt{http\_eoh}\}\}\texttt{)]}\,Y))
\end{array}\right) \\
\quad \Big)\texttt{))}
\end{array} \tag{7}$$

We measure the respective overheads resulting from the hybrid instrumentations over Yaws for varying client loads in terms of (*i*) the average CPU utilization required; (*ii*) the memory overheads per Yaws client request; and (*iii*) the average time taken for the (monitored) Yaws server to respond to batches of simultaneous client request.[7] 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 property and each client load, we take three sets of readings and then average them out. Since results do not show substantial variations when different properties were considered, we again average them across all properties and compile them in the graphs shown in Fig. 6.

The results show that the hybrid instrumentation yielded CPU utilisation and memory overheads that are substantially lower than those incurred by a synchronous instrumentation, comparable to those of asynchronous monitoring with code injections. The second graph even shows that the memory utilisation for both of these instrumentations is less than that for asynchronous monitoring performed through the EVM tracing of [23]. In the case of response-time latencies, where synchronous monitoring fared the worst (on average 30% higher overheads than its asynchronous counterpart) a hybrid approach managed to lower response times to overheads that are about 15% higher than asynchronous monitoring; see Fig. 6 bottom graph.

7 Conclusion

We studied various monitoring techniques for actor-based frameworks in the context of Erlang and integrated them within a tool for runtime verification of actor systems. Our contributions are:

- A novel hybrid instrumentation technique minimising the amount of (expensive) synchronous monitor instrumentations while still guaranteeing timely violation detections.
- An extension to detectEr, an RV tool for Erlang (actor-based) programs, that allows the verifier to control which violations to monitor synchronously and asynchronously within the same property.
- A case study demonstrating the applicability of the technique and tool to monitor safety properties for Yaws, a concurrent web-server written in Erlang.
- A systematic assessment of the relative overheads incurred by different instrumentation techniques within an actor setting.

[7] In the extended syntax, properties that are exclusively defined in terms of synchronous necessity formulas, $\texttt{[|}\alpha\texttt{|]}\varphi$, yield synchronous monitor instrumentations that are identical to those discussed in Sec. 4.

Related Work: Several verification and modeling tools [39, 26, 12, 36, 38] for actor-based component systems already exist. Rebeca [39, 2] is an actor-based modeling language designed with the aim of bridging the gap between formal verification approaches and real applications. It provides conversions to renowned model checkers like SMV and Promela, and extends the model with timing constraints; timed-rebeca models have also been translated into Erlang. McErlang [26] is a model-checker specifically targeting Erlang code; it uses a superset of our logic. We are however unaware of any extensions of these tools to RV. Apart from detectEr [23], other RV tools for actor based system exist. eLarva[14] is another Erlang monitoring tool that uses the EVM tracing mechanism to perform *asynchronous* monitoring; no facility for synchronous monitoring is provided. In [38], Sen *et al.* explore a decentralized (orchestrated) monitoring approach as a way to reduce the communication overheads that are usually caused by a centralized approach and implement it in terms of an actor-based tool called DiAna. Although their investigation is orthogonal to ours, it would be interesting to integrate this study within ours and systematically evaluate whether choreographed monitor arrangements yield further overhead gains.

By and large, most widely used online RV tools employ synchronous instrumentation [31, 13, 11, 19, 8]. There is also a substantial body of work commonly referred to as asynchronous RV [18, 17, 6]. However, the latter tools and algorithms assume *completed traces*, generated by complete program executions and stored in a database or a log file; as explained in the Introduction, we term these bodies of work as *offline*, and their aims are considerably different from the work presented here. There exist a few tools offering both synchronous and asynchronous monitoring, such as MOP [10, 11] and JPAX [28, 37]. Crucially, however, they do not provide the fine grained facility of supporting both synchronous and asynchronous monitoring at the same time, or switching between the two modes at runtime.

Rosu *et al.* [37] make a similar distinction to ours between offline, synchronous online and asynchronous online monitoring. However, their definitions of synchrony and asynchrony deals with the timeliness of detections. By contrast we focus on how instrumentation is carried out and show how hybrid instrumentation can be used to obtain timely detections for certain properties; this amounts to synchronous monitoring in [37]. A closer work to ours is [16]: they allow a decoupling between the system and monitor executions but provide explicit mechanisms for pausing the system while the lagging monitor execution catches up. In our case, this mechanism is handled *implicitly* when switching from asynchronous to synchronous monitoring. In [16] they do not provide an implementation of their constructs and do not asses the relative overhead costs incurred by different instrumentation strategies.

Talcott *et al.* in [41] compare and contrast three coordination models for actors which cover a wide spectrum of communication mechanisms and coordination strategies. The comparison focusses on the level of expressivity of each model, the level of maturity, the level of abstraction and the way user definable coordination behavior is provided. One of the analysed coordination models, ie. the Reo model, is a channel based language in which channels may be either synchronous or asynchronous. This model resembles the way our hybrid monitoring protocol interacts with the monitored system. In fact our monitoring language constructs $[\alpha]$ and $[|\alpha|]$, seem analogous to the asynchronous and synchronous channels in the Reo model.

Future Work: Our hybrid instrumentation technique can be extended to other inherently asynchronous computational models, such as monitoring for distributed systems [22]. This would require us to consider additional aspects such as the handling of multiple, partially-ordered traces and the use of alternative monitor organisations such as monitor choreographies. Our techniques can also be extended with enforcement mechanism [21], facilitated by the fact that corrective action can be carried out as soon as the system performs the violation. This would be worth exploring in the context of Erlang and detectEr.

References

[1] *AKKA website*. Available at `http://www.akka.io`.

[2] Luca Aceto, Matteo Cimini, Anna Ingolfsdottir, Arni Hermann Reynisson, Steinar Hugi Sigurdarson & Marjan Sirjani (2011): *Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca*. In: *FOCLASA*, *EPTCS* 58, pp. 1–19, doi:10.4204/EPTCS.58.1.

[3] Luca Aceto & Anna Inglfsdttir (1999): *Testing Hennessy-Milner Logic with Recursion*. In: *FoSSaCS*, *LNCS* 1578, Springer, pp. 41–55, doi:10.1007/3-540-49019-1_4.

[4] Gul Agha, Ian A. Mason, Scott F. Smith & Carolyn L. Talcott (1997): *A Foundation for Actor Computation*. *Journal of Functional Programming*, pp. 1–72, doi:10.1017/S095679689700261X.

[5] Gul A. Agha, Prasannaa Thati & Reza Ziaei (2001): *Formal Methods for Distributed Processing*. chapter Actors: A Model for Reasoning About Open Distributed Systems, Cambridge University Press, New York, NY, USA, pp. 155–176, doi:10.1.1.1.6356.

[6] James H. Andrews & Yingjun Zhang (2003): *General Test Result Checking with Log File Analysis*. *IEEE Trans. Software Eng.* 29, pp. 634–648, doi:10.1109/TSE.2003.1214327.

[7] Joe Armstrong (2007): *Programming Erlang*. The Pragmatic Bookshelf.

[8] Howard Barringer, Ylis Falcone, Klaus Havelund, Giles Reger & David E. Rydeheard (2012): *Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors*. In: *FM*, *LNCS* 7436, Springer, pp. 68–84, doi:10.1007/978-3-642-32759-9_9.

[9] Francesco Cesarini & Simon Thompson (2009): *ERLANG Programming*, 1st edition. O'Reilly.

[10] Feng Chen & Grigore Roşu (2005): *Java-MOP: A Monitoring Oriented Programming Environment for Java*. In: *TACAS'05*, *LNCS* 3440, Springer, pp. 546–550, doi:10.1007/978-3-540-31980-1_36.

[11] Feng Chen & Grigore Roşu (2007): *MOP: An Efficient and Generic Runtime Verification Framework*. In: *OOPSLA*, ACM press, pp. 569–588, doi:10.1145/1297027.1297069.

[12] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer & Carolyn Talcott (2007): *All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Springer-Verlag, Berlin, Heidelberg.

[13] C. Colombo, G.J. Pace & G. Schneider (2009): *LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper)*. In: *SEFM*, pp. 33–37, doi:10.1109/SEFM.2009.13.

[14] Christian Colombo, Adrian Francalanza & Rudolph Gatt (2011): *Elarva: A Monitoring Tool for Erlang*. In: *RV*, *LNCS* 7186, Springer, pp. 370–374, doi:10.1007/978-3-642-29860-8_29.

[15] Christian Colombo, Adrian Francalanza, Ruth Mizzi & Gordon J. Pace (2012): *polyLarva: Runtime Verification with Configurable Resource-Aware Monitoring Boundaries*. In: *SEFM*, pp. 218–232, doi:10.1007/978-3-642-33826-7_15.

[16] Christian Colombo & Gordon J. Pace (2012): *Fast-Forward Runtime Monitoring - An Industrial Case Study*. In: *RV*, *LNCS* 7687, Springer, pp. 214–228, doi:10.1007/978-3-642-35632-2_22.

[17] Marcelo d'Amorim & Klaus Havelund (2005): *Event-based Runtime Verification of Java Programs*. *SIGSOFT Softw. Eng. Notes* 30(4), pp. 1–7, doi:10.1145/1082983.1083249.

[18] B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra & Manna Z. (2005): *LOLA: Runtime Monitoring of Synchronous Systems*. In: *TIME*, IEEE, pp. 166–174, doi:10.1109/TIME.2005.26.

[19] Normann Decker, Martin Leucker & Daniel Thoma (2013): *jUnitRV - Adding Runtime Verification to jUnit*. In: *NASA Formal Methods*, *LNCS* 7871, Springer-Verlag Berlin Heidelberg, Springer, pp. 459–464, doi:10.1007/978-3-642-38088-4_34.

[20] Atilla Erdödi: *Exago: Property monitoring via log file analysis*. Available at `http://www.protest-project.eu/upload/slides/ErlangUserGroupMeeting-AtillaErdodi-Exago.pdf`.

[21] Yliès Falcone, Jean-Claude Fernandez & Laurent Mounier (2012): *What can you verify and enforce at runtime?* *STTT* 14(3), pp. 349–382, doi:10.1007/s10009-011-0196-8.

[22] Adrian Francalanza, Andrew Gauci & Gordon J. Pace (2013): *Distributed System Contract Monitoring*. *JLAP* 82(5-7), pp. 186–215, doi:10.1016/j.jlap.2013.04.001.

[23] Adrian Francalanza & Aldrin Seychell (2013): *Synthesising Correct Concurrent Runtime Monitors (Extended Abstract)*. In: *RV*, *LNCS* 8174, Springer, pp. 112–129, doi:10.1007/978-3-642-40787-1_7.

[24] Adrian Francalanza, Aldrin Seychell & Ian Cassar: *DetectEr*. Available at `https://bitbucket.org/casian/detecter-stable-releases`.

[25] Lars-Åke Fredlund (2001): *A Framework for Reasoning about Erlang Code*. Ph.D. thesis, Royal Institute of Technology, Stockholm, Sweden.

[26] Lars-Åke Fredlund & Hans Svensson (2007): *McErlang: a model checker for a distributed functional programming language*. ICFP '07, ACM, New York, NY, USA, pp. 125–136, doi:10.1145/1291151.1291171.

[27] Philipp Haller & Frank Sommers (2012): *Actors in Scala*. Artima Inc., USA.

[28] Klaus Havelund & Grigore Roşu (2001): *Monitoring Java Programs with Java PathExplorer*. In: *RV (in connection with CAV)*, *ENTCS* 55:2, pp. 200–217, doi:10.1016/S1571-0661(04)00253-1.

[29] Alejandro Hernandez (2010): *Yaws 1.89: Directory Traversal Vulnerability*. Available at `http://www.exploit-db.com/exploits/15371/`. Accessed on 6/6/2014.

[30] Zachary Kessin (2012): *Building Web Applications with Erlang: Working with REST and Web Sockets on Yaws*. O'Reilly Media.

[31] MoonZoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee & Oleg Sokolsky (2004): *Java-MaC: A Run-Time Assurance Approach for Java Programs*. *Formal Methods in System Design* 24(2), pp. 129–155, doi:10.1023/B:FORM.0000017719.43755.7c.

[32] Alexei Krasnopolski: *AOP for Erlang*. Available at `http://erlaop.sourceforge.net/`.

[33] Martin Leucker & Christian Schallhart (2009): *A brief account of Runtime Verification*. *JLAP* 78(5), pp. 293 – 303, doi:10.1016/j.jlap.2008.08.004.

[34] Martin Logan, Eric Merritt & Richard Carlsson (2011): *Erlang and OTP in Action*. Manning.

[35] Ian A. Mason & Carolyn L. Talcott (1999): *Actor languages their syntax, semantics, translation, and equivalence*. *Theoretical Computer Science* 220, pp. 409 – 467, doi:10.1016/S0304-3975(99)00009-2.

[36] Shangping Ren & Gul Agha (1995): *RTsynchronizer: Language Support for Real-Time Specifications in Distributed Systems*. In Richard Gerber & Thomas J. Marlowe, editors: *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pp. 50–59, doi:10.1145/216636.216656.

[37] Grigore Roşu & Klaus Havelund (2005): *Rewriting-Based Techniques for Runtime Verification*. *Automated Software Engg.* 12(2), pp. 151–197, doi:10.1007/s10515-005-6205-y.

[38] Koushik Sen, Abhay Vardhan, Gul Agha & Grigore Roşu (2004): *Efficient Decentralized Monitoring of Safety in Distributed Systems*. *ICSE*, pp. 418–427, doi:10.1109/ICSE.2004.1317464.

[39] Marjan Sirjani, Ali Movaghar, Amin Shali & Frank S. de Boer (2004): *Modeling and Verification of Reactive Systems Using Rebeca*. *Fundam. Inf.* 63(4), pp. 385–410.

[40] Hans Svensson, Lars-Åke Fredlund & Clara Benac Earle (2010): *A unified semantics for future Erlang*. In: *Erlang Workshop*, ACM, pp. 23–32, doi:10.1145/1863509.1863514.

[41] Carolyn Talcott, Marjan Sirjani & Shangping Ren (2008): *Comparing Three Coordination Models: Reo, ARC, and RRD*. *ENTCS* 194(4), pp. 39–55, doi:10.1016/j.entcs.2008.03.098.

[42] Alfred Tarski (1955): *A lattice-theoretical fixpoint theorem and its applications*. *Pacific Journal of Mathematics* 5(2), pp. 285–309.

[43] Steve Vinoski (2011): *Yaws: Yet another web server*. *IEEE Internet Computing* 15(4), pp. 90–94, doi:10.1109/MIC.2011.100.