# Improving Runtime Overheads for detectEr

Ian Cassar *
CS Dept, University of Malta.
ian.cassar.10@um.edu.mt

Adrian Francalanza
CS Dept, University of Malta.
adrian.francalanza@um.edu.mt

Simon Said
CS Dept, University of Malta.
ssai0008@um.edu.mt

We design monitor optimisations for detectEr, a runtime-verification tool synthesising systems of concurrent monitors from correctness properties for Erlang programs. We implement these optimisations as part of the existing tool and show that they yield considerably lower runtime overheads when compared to the unoptimised monitor synthesis.

## 1 Introduction

Runtime Verification (RV) [10] is a lightweight verification technique mitigating the scalability issues associated with exhaustive verification techniques such as model checking. Low overheads are an important requirement for the viability of any RV framework, where the additional computation introduced by the monitors should ideally be kept to a minimum.

detectEr[3, 7] is an RV tool for analysing the correctness of concurrent Erlang programs — the analysis of concurrent programs is notoriously hard and often leads to state explosion problems. From a safety correctness property (defined through a formal logic), detectEr generates a system of monitors that execute concurrently with the system under scrutiny, analysing its execution trace, and raising an alert as soon as a violation to the *resp.* correctness property is detected. In [7], it is shown that the monitors generated by the tool are indeed correct (*e.g.*, they only raise an alert when the system violates the *resp.* property) whereas in [3] the authors study the relationship between synchronous and asynchronous instrumentation in this setting, establishing (amongst other things) that asynchronous monitoring consistently yields the lowest level of overheads.

In this paper we study optimisation techniques for further lowering the overheads of the tool's asynchronous monitors.[1] The monitor synthesis defined in [7] uses *concurrent* monitors to parallelise the runtime analysis as much as possible and exploit better the underlying hardware architecture (which nowadays typically includes multiple computing cores). However, in order to simplify the correctness proofs, this synthesis is kept as *regular* as possible: the monitor-generation strategy is the same for every logical construct and does not take into consideration the syntactic context of where that logical construct appears in the correctness property. Moreover, the communication organisation of the generated concurrent monitors is also kept *static* throughout the execution of the program, even though certain monitor subsystems become redundant during the runtime analysis. In this work we address these two potential sources of inefficiency by defining *fine-tuned* organisations of concurrent monitors specifically tailored to different forms of logical formulas; in addition, these monitors are also able to perform a degree of *reconfiguration* during the runtime analysis. We incorporate the new strategies into the existing tool and show that the generated monitors produce lower overheads than the existing monitor translations.

The rest of the paper is structured as follows. § 2 introduces the tool whereas § 3 identifies inefficiencies and proposes solutions. § 4 discusses performance improvements and § 5 concludes.

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

[1] These optimisations may also be extended to synchronous monitors and should also yield lower overheads to that setting.

B. Buhnova, L. Happe, J. Kofroň (Eds.):
Formal Engineering approaches to Software Components and Architectures (FESCA'15).
EPTCS 178, 2015, pp. 1–8, doi:10.4204/EPTCS.178.1

$$\varphi,\psi \in \texttt{FRM} ::= \texttt{tt} \mid \texttt{ff} \mid \varphi\,\&\,\psi \mid [\alpha]\varphi \mid X \mid \texttt{max}\,X.\varphi \mid \texttt{if}\,b\,\texttt{then}\,\varphi\,\texttt{else}\,\psi$$

$$[\![\texttt{tt}]\!] \stackrel{\text{def}}{=} \textsc{Actr} \qquad [\![\texttt{ff}]\!] \stackrel{\text{def}}{=} \emptyset \qquad [\![\varphi\,\&\,\psi]\!] \stackrel{\text{def}}{=} [\![\varphi]\!] \cap [\![\psi]\!]$$

$$[\![[\alpha]\varphi]\!] \stackrel{\text{def}}{=} \left\{ A \mid \left( A \stackrel{\beta}{\Longrightarrow} B \text{ and } \text{match}(\alpha,\beta) = \sigma \right) \text{ implies } B \in [\![\varphi\sigma]\!] \right\}$$

$$[\![\texttt{max}\,X.\varphi]\!] \stackrel{\text{def}}{=} \bigcup \{ S \mid S \subseteq [\![\varphi\{X \mapsto S\}]\!] \} \qquad [\![\texttt{if}\,b\,\texttt{then else}\,\varphi\,\psi]\!] \stackrel{\text{def}}{=} \begin{cases} [\![\varphi]\!] & \text{if } b \Downarrow \mathsf{true} \\ [\![\psi]\!] & \text{if } b \Downarrow \mathsf{false} \end{cases}$$

Figure 1: The Logic and its Semantics

## 2 detectEr Primer

**The Logic:** Correctness properties in detectEr are expressed through the logic sHML [1] (a syntactic subset of the modal $\mu$-calculus). The sHML syntax is defined inductively using the BNF in Fig. 1. 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 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.[2] Formulas include truth and falsity, `tt` and `ff`, conjunctions, $\varphi$ `&` $\psi$, modal necessities, $[\alpha]\varphi$, maximal fixpoints to describe recursive properties, `max` $X.\varphi$, and conditionals to reason about data, `if` $b$ `then` $\varphi$ `else` $\psi$. A necessity formula, $[\alpha]\varphi$, may contain term variables in $\alpha$ that *pattern-match* with actual (closed) actions, thus acting as a *binder* for these variables in the subformula $\varphi$; similarly `max` $X.\varphi$ is a binder for $X$ in $\varphi$.

The semantics of the logic is defined for *closed* formulas, over Erlang programs interpreted as a Labelled Transition Systems (LTSs), as shown in [7]. In our case, an LTS takes the form $\langle \textsc{Actr}, \textsc{Act} \cup \{\tau\}, \rightarrow \rangle$, where $A, B \in \textsc{Actr}$ are nodes denoting actor systems, $\textsc{Act} \cup \{\tau\}$ are actions including a silent (internal) action $\tau$, and $\rightarrow$ is a ternary relation of type $\textsc{Actr} \times (\textsc{Act} \cup \{\tau\}) \times \textsc{Actr}$; we write $A \xrightarrow{\alpha} B$ in lieu of $(A, \alpha, B) \in \rightarrow$ and use $A \stackrel{\alpha}{\Longrightarrow} B$ to denote $A(\xrightarrow{\tau})^* \cdot \xrightarrow{\alpha} \cdot (\xrightarrow{\tau})^* B$. The semantics is given in Fig. 1 and follows that of [7]. No actor system satisfies `ff`, whereas all actors satisfying `tt`. Actors satisfying $\varphi$ `&` $\psi$ must satisfy both $\varphi$ and $\psi$. Necessity formulas $[\alpha]\varphi$ are satisfied by all actor systems $A$ observing the condition that, *whenever* pattern-matchable actions $\beta$ are performed (yielding substitution $\sigma :: \textsc{Var} \rightharpoonup \textsc{Val}$), 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 `max` $X.\varphi$ denotes the maximal fixpoint of the functional $[\![\varphi]\!]$ and allow the logic to be defined over actors with *infinite* behaviour; following standard fixpoint theory [13], this is characterised as the union of all post-fixpoints $S \in \mathcal{P}(\textsc{Actr})$ (in Fig. 1, $\{X \mapsto S\}$ denotes the substitution of $S$ for $X$). Finally, a conditional, `if` $b$ `then` $\varphi$ `else` $\psi$, equates to $\varphi$ whenever $b$ evaluates to $\mathsf{true}$ and to $\psi$ when $b$ evaluates to $\mathsf{false}$.

**Example 2.1.** Consider an Erlang system implementing a *predecessor server* receiving messages of the form *(n, clientID)* and returning $n-1$ back to *clientID* whenever $n > 0$, but reporting the offending client to an error handler, `err`, whenever $n = 0$. It may also announce termination of service by sending a

[2] Although we here work up-to $\alpha$-conversion, detectEr accordingly renames duplicate variables during pre-processing.

message to end. A safety correctness property in our logic would be:

$$\texttt{max}X.\texttt{[srv?}\{x,y\}\texttt{]}\left(\begin{array}{l}(\texttt{[end!\_]ff}) \;\;\&\;\; (\texttt{[err!}z\texttt{]}(\texttt{if}\ (x\neq 0 \vee y\neq z)\ \texttt{then ff else}\ X)) \\ \quad \&\;\; (\texttt{[}y\texttt{!}z\texttt{]}(\texttt{if}\ z=(x-1)\ \texttt{then}\ X\ \texttt{else ff}))\end{array}\right) \quad (1)$$

It is a recursive formula, $\texttt{max}\,X.(\ldots)$, stating that, *whenever* the server implementation receives a request (input action), [srv?{x,y}]..., with value $x$ and return (client) address $y$, then it should *not*:

1. terminate the service (before handing the client request), [end!_]ff.
2. report an error, [err!z]..., when $x$ is not 0, or with a client other than the offending one, $y \neq z$.
3. service the client request, [y!z]..., with a value other than $x-1$.

These conditions are *invariant*; maximal fixpoints capture this invariance for server implementations that *may never terminate* (they are considered correct as long as the conditions above are not violated). ■

**The Monitor Synthesis:** The synthesis algorithm of [7] aims to be *modular*: it generates independently executing monitor combinators for each logical construct, interacting with one another through message passing.[3] For instance, the synthesis *parallelises* the analysis of the subformulas $\varphi_1$ and $\varphi_2$ in a conjunction $\varphi_1 \& \varphi_2$ by (*i*) synthesising concurrent monitor systems for $\varphi_1$ and $\varphi_2$ *resp.* and (*ii*) creating a conjunction monitor combinator that receives trace events and forks (forwards) them to the independently-executing monitors of $\varphi_1$ and $\varphi_2$. Since the submonitor systems for $\varphi_1$ and $\varphi_2$ may be arbitrarily complex (needing to analyse a stream of events before reaching a verdict), the conjunction monitor is *permanent* in the monitor organisation generated, so as to fork and forward event streams of arbitrary length. It is also worth noting that the synthesis algorithm assumes formulas to be *guarded*, where recursive variables appear under necessity formulas; this is required to generate monitors that implement *lazy* unrolling of recursive formulas, thereby minimizing overheads (see [7] for details).

**Example 2.2.** From formula (1), the monitor organisation $m$ (depicted in Fig. 2) is generated, consisting of one process acting as the combinator for the necessity formula $\texttt{[srv?}\{x,y\}\texttt{]}\varphi$. If an event of the form $\texttt{srv?}\{v,c\}$ is received, the process pattern matches it with srv?{x,y} (mapping variables $x$ and $y$ to the values $v$ and $c$ *resp.*) and spawns the (dashed) monitor system shown underneath it in Fig. 2, instantiating the variables $x,y$ with $v,c$ *resp.* The subsystem consisting of three monitor subsystems, one for each subformula guarded by $\texttt{[srv?}\{x,y\}\texttt{]}$ in (1), connected by two conjunction forking monitors. When the next trace event is received, *e.g.*, c!v-1 (a server reply to client $c$ with value $v-1$), the conjunction monitors replicate and forward this event to the three monitor subtrees. Two of these subtrees do not pattern match this event and terminate; the third subtree (submonitor $m2$) pattern-matches it however, instantiating $z$ for $(v-1)$, evaluating the conditional and unfolding the recursive variable $X$ to monitor $m$. If another server request event is received, $\texttt{srv?}\{v',c'\}$ (with potentially different client and value arguments $c'$ and $v'$), the conjunction monitors forward it to $m$, pattern matching it and generating a subsystem with two further conjunction combinators as before. ■

# 3 Optimizations

Formula (1) is a pathological example, highlighting two inefficiencies introduced by the synthesis algorithm of § 2. First, conjunction monitors mirror closely their syntactic counterpart and can only handle

[3] Every combinator is implemented as a (lightweight) Erlang process (actor) [4], uniquely identifiable by its process ID. Messages sent to a process are received in its dedicated mailbox, and may be read selectively using (standard) pattern-matching.

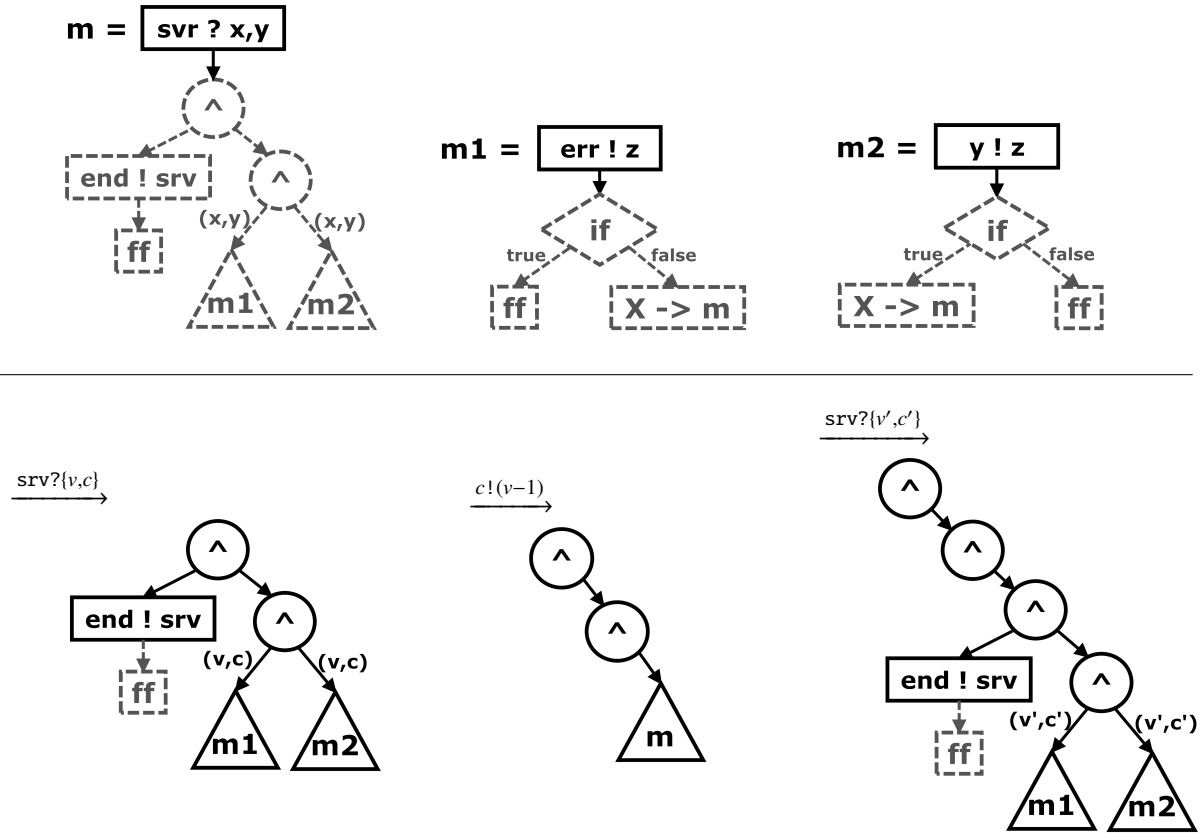


Figure 2: Monitor Generation for Formula (1) and its execution *wrt.* trace $\texttt{srv?}\{v,c\}.c\,!\,(v-1).\texttt{srv?}\{v',c'\}$

forwarding to *two* sub-monitors. As a result, formulas with *nested* conjunctions — as in formula (1) — translate into organisations of *cascading conjunction monitors* that are inefficient at forwarding trace events. For instance, the two cascading conjunction monitors of $m$ in Fig. 2 replicate the trace event c!v-1 *four* times in order to forward it to three sub-monitor systems; the problem is accentuated for higher numbers of nested conjunctions and repeated forwarding.

Second, the current monitor implementation does not perform any monitor reorganisations at runtime. When a conjunction formula $\varphi_1 \& \varphi_2$ is translated, the conjunction combinator monitor organisation is kept permanent throughout its execution because it is assumed that the *resp.* sub-monitors for $\varphi_1$ and $\varphi_2$ are long-lived. This heuristic however does not apply in the case of formula (1), where two out of the three sub-monitors terminate after a single event is received. This feature, in conjunction with recursive unfolding, creates *chains of indirections* through conjunction monitors with only one child, as shown in Fig. 2 (bottom row).

**Proposed Solutions:** The first optimisation we introduce is that of conjunction monitor combinators that fork-out to an *arbitrary number* of monitor subsystems. For instance, the corresponding monitor formula (1) would translate into a monitor organisation consisting of *one* conjunction combinator with *three* children (instead of two combinators with two children each) as shown in Fig. 3(left). This is more

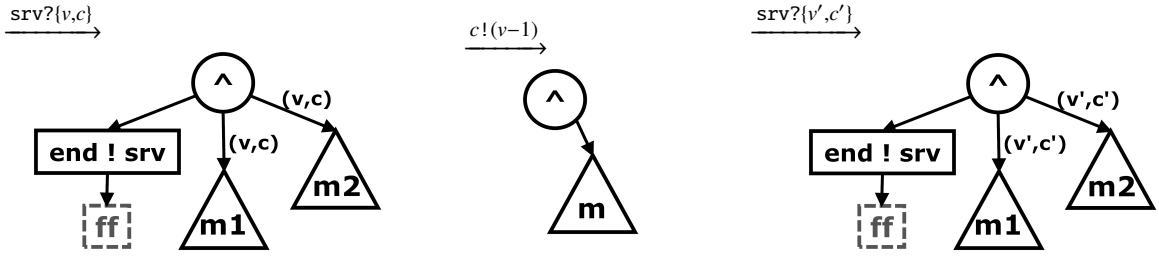


Figure 3: Optimised Synthesis for Formula (1) and its execution *wrt.* $\texttt{srv?}\{v,c\}.c\,!\,(v-1).\texttt{srv?}\{v',c'\}$

efficient from from the point of view of processes created, but also in terms of the number of replicated messages required to perform the necessary event forwarding to monitor subsystems. For example, the conjunction combinator of Fig. 3 generates *three* message replications to forward an event to the three sub-monitors (as opposed to the *four* messages of Fig. 2, as discussed earlier).

The second optimisation considered is that of allowing conjunction monitor combinators to *dynamically reorganise* the monitor configuration so as to keep the event flow structure as efficient as possible. In order to keep overheads low, this reconfiguration operation should be kept *local*, where unaffected monitor subsystems should continue with their runtime analysis while the restructuring is in process. Stated otherwise, the monitor reorganisation happens while trace events are *still being received*, and the operation needs to guarantee that (*i*) no trace events are lost (*ii*) trace events are not reordered.

Reorganisations are carried out by conjunction combinators, which are now allowed to *add* and *delete* monitor subsystems from their internal list of children. For instance, when an event causes a child sub-monitor to terminate, the parent (conjunction) monitor is sent a termination message which allows it to remove the terminated sub-monitor from its child-list.

The reconfiguration protocol is kept local (*i.e.,* other parts of the monitor graph are unaffected), and is carried by two (multi-child) conjunction combinators in a parent-child setup. It proceeds as follows:

1. When an event causes a child sub-monitor to become a system with a conjunction combinator at its root, it sends a *merge-request* to its parent. In the meantime the child sub-monitor may start receiving events from its parent and forwards them to its children.
2. When parent conjunction combinator reads the merge request, it sends a *merge-ack* back to child monitor and waits for a *merge-msg* from this child; while waiting for this merge message, the parent monitor stops retrieving further trace events from its mailbox, effectively using it as a buffer for future events that may keep on being received.
3. As soon as the child monitor receives the *merge-ack* message, it forwards all the remaining events in its mailbox to its children. Once it empties its mailbox, it sends a *merge-msg* back to the parent with a list of its children sub-monitors and waits for a *merge-final* message.
4. Upon receiving *merge-msg* the parent removes the child sub-monitor sending the message and, instead, adds the sub-monitors sent by this child to its own list. It then sends a *merge-final* to the child monitor and waits for a *merge-complete* message.
5. When the child receives *merge-final*, it retrieves any possible merge requests sent by its former children, forwards them in order to its parent, followed by a *merge-complete* message, and terminates.

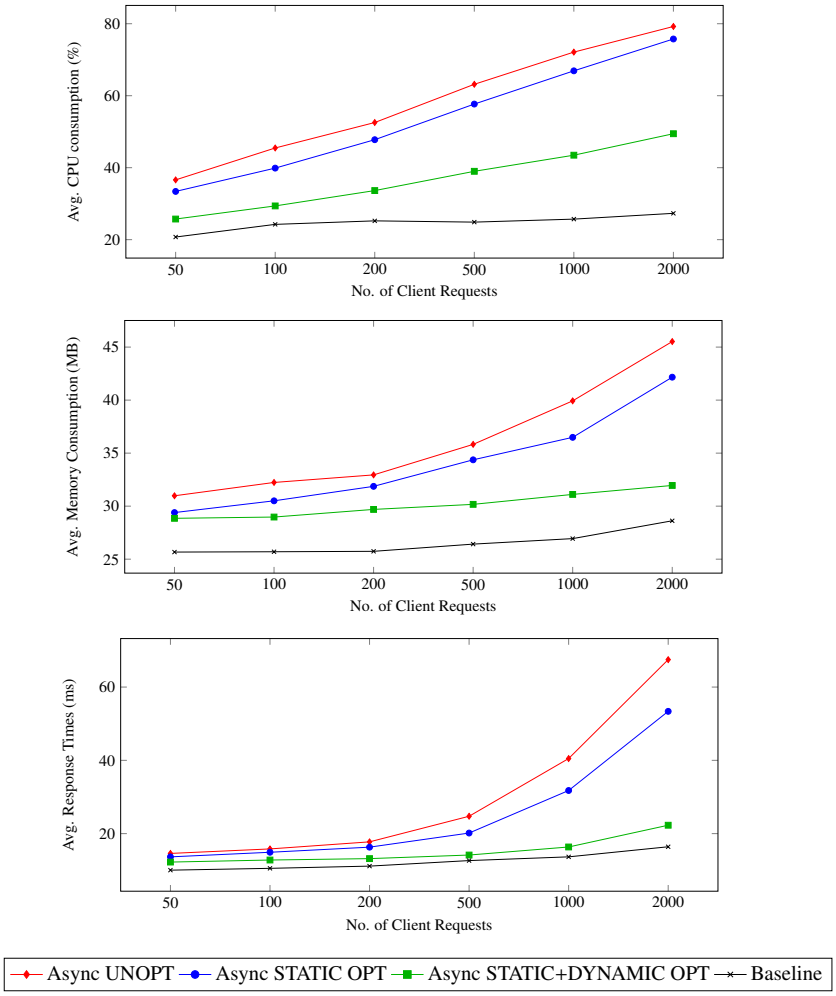


Figure 4: Evaluation results

6. When the parent receives *merge-complete*, it reverts back to its normal operation of trace forwarding.

The monitor restructuring protocol discussed yields monitor organisations with only *one* (eventual) conjunction node at the root, and a list of monitor subsystems processing the forwarded events (a spider-like configuration). For instance, for the event trace $\texttt{srv?}\{v,c\}.c!(v-1).\texttt{srv?}\{v',c'\}$, the synthesised monitor for formula (1) yields the evolution shown in Fig. 3.

# 4 Evaluation

Through a series of empirical tests, we verify whether the monitor optimisations of § 3 yield the expected overhead improvements. In particular, such gains are not obvious for the second optimisation presented in § 3, where reconfigurations introduce additional computation that may offset the lower overheads obtained from addressing the inefficiencies of redundant monitor code discussed in § 2.

The tests are carried out on a third-party commercial software called Yaws [9], an HTTP webserver written in Erlang. In order to keep high levels of throughput, the Yaws server assigns a dedicated (concurrent) handler servicing HTTP requests for every client connection, thereby parallelising processing for multiple clients. The evaluation is based on a variety of safety properties for the Yaws server implementation, expressed in terms of the logic discussed in § 2. The tests carried out employ three synthesis algorithms to obtain monitors from these properties, namely (*i*) the unoptimised synthesis (presented in [7]), (*ii*) a monitor synthesis employing multi-child conjunction combinators (without reorganisation) and (*iii*) a synthesis employing *both* optimisations of § 3 (including dynamic reorganisations); these are compared to baseline readings, *i.e.*, the unmonitored system.

The tests measure the respective overheads for these three synthesis algorithms, and compare the respective overheads induced *wrt.* the baseline (unmonitored) Yaws execution for varying client loads. Overheads are calculated in terms of (*i*) the average CPU utilization; (*ii*) the memory overhead per client request; and (*iii*) the average time taken for the server to respond to batches of simultaneous client request. The experiments are 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 the average of three sets of readings. Since results did not yield substantial variations for the different properties synthesised, we present averaged readings across all properties in the graphs shown in Fig. 4.

The results show that just using multi-child conjunction combinators yield modest yet consistent gains in terms of CPU usage, memory consumption and average response times, when compared with the two-pronged conjunction combinator of [7]. However, such monitors still appear to induce non-linear overhead increases, probably created by the chains of monitor indirections created for recursive formula unfolding (see discussion in § 3). This problem however seems to be rectified for monitors with dynamic reorganisations, as can be seen from the graphs in Fig. 4. In particular, overheads appear to be comparable to the baseline execution for memory consumption.

## 5 Conclusion

We present monitor optimisations for detectEr, an RV tool synthesising concurrent monitors for Erlang correctness properties. We implement these optimisations as part of the existing tool and demonstrate that they yield considerably lower runtime overheads when compared to the original monitor synthesis presented in [7]. We conjecture that similar overhead improvements should be observed if these optimisation techniques are applied to the synchronous monitoring studied in [3].

**Related Work:** Several verification and modeling tools [12, 8, 11] for actor-based component systems already exist. Rebeca [12] is an actor-based modeling language providing automated translation to renowned model checkers like SMV and Promela; timed-rebeca models have also been translated into Erlang. McErlang [8] is a model-checker specifically targeting Erlang code that uses a superset of our logic; to the best of our knowledge the tool does not consider any verification post-deployment, as in the case of RV. As far as we are aware, eLarva [5] is the only other RV tool for Erlang programs. Similar to the setup studied in this work, it synthesises monitors that use the Erlang Virtual Machine tracing mechanism to obtain system trace events in *asynchronous* fashion. Apart from the source logic used (eLarva properties are described as automata-based specifications), a key difference between this tool and detectEr is that eLarva produces *monolithic* monitors, as opposed to the component based monitor systems described in this paper; as a result, the optimisation techniques discussed do not apply. In [11], Sen *et al.* explore a decentralized (choreographed) monitoring approach as a way to reduce the com-

munication overheads that are usually caused by a centralized approach and implement it in terms of an actor-based tool called DiAna. It would be interesting to explore to what extent the optimisations presented in this work can be extended to the distributed setting of DiAna, and whether these optimisations would yield comparable overhead gains. Our techniques may also be relevant to lower overheads in other component-based monitor synthesis algorithms such as in [2] (which has a fixed monitor organisation) or in [14] (which supports a level of dynamic reorganisation as updatable distributed tables). Similar investigations could also be carried out for the distributed monitoring approaches studied in [6].

# References


[1] 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.

[2] Andreas Bauer & Ylies Falcone (2012): *Decentralised LTL Monitoring*. In: *Formal Methods (FM)*, *LNCS* 7436, Springer, pp. 85–100, doi:10.1007/978-3-642-32759-9_10.

[3] Ian Cassar & Adrian Francalanza (2014): *On Synchronous and Asynchronous Monitor Instrumentation for Actor-based systems*. In: *FOCLASA*, *EPTCS* 175, pp. 54–68, doi:10.4204/EPTCS.175.4.

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

[5] 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.

[6] 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.

[7] Adrian Francalanza & Aldrin Seychell (2014): *Synthesising Correct Concurrent Runtime Monitors*. *Formal Methods in System Design (FMSD)*, pp. 1–36, doi:10.1007/s10703-014-0217-9.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] Alfred Tarski: *A lattice-theoretical fixpoint theorem and its applications.* *Pacific Journal of Mathematics* 5(2), pp. 285–309, doi:10.2140/pjm.1955.5.285.

[14] Wenchao Zhou, Oleg Sokolsky, Boon Thau Loo & Insup Lee (2009): *DMaC: Distributed Monitoring and Checking*. In: *RV*, *LNCS* 5779, Springer, pp. 184–201. Available at `http://dblp.uni-trier.de/db/conf/rv/rv2009.html#ZhouSLL09`.