Better Late than Never or: Verifying Asynchronous Components at Runtime*

Duncan Paul Attard[✉,1,2], Luca Aceto[2,3], Antonis Achilleos[2], Adrian Francalanza[1], Anna Ingólfsdóttir[2], and Karoliina Lehtinen[4]

[1] University of Malta, Msida, Malta {duncan.attard.01,afra1}@um.edu.mt
[2] Reykjavík University, Reykjavík, Iceland
{duncanpa17,luca,antonios,annai}@ru.is
[3] Gran Sasso Science Institute, L'Aquila, Italy luca.aceto@gssi.it
[4] CNRS, Aix-Marseille University and University of Toulon, LIS, Marseille, France
lehtinen@lis-lab.fr

Abstract. This paper presents detectEr, a runtime verification tool for monitoring asynchronous component systems. The tool synthesises executable monitors from properties expressed in terms of the safety fragment of the modal μ-calculus. In this paper, we show how a number of useful properties can be flexibly runtime verified via the three forms of instrumentation—inline, outline, and offline—offered by detectEr to cater for specific system set-up constraints.

Keywords: Runtime Verification · Instrumentation · Monitoring

1 Do You Want To Know A Secret

In the Cockaigne of software development, programs are verified using a smorgasbord of *pre-deployment* techniques, and executed only when their *correctness is ascertained*. Reality, however, tells a different story. Mainstream verification practices, including testing [47], only reveal the *presence* of errors [29]. Exhaustive approaches like model checking [41] are laborious to use, *e.g.* building effective program models is non-trivial, and known to suffer from state explosion problems [27]. Other methods such as type systems [48] are intentionally lightweight to prevent disrupting the software development lifecycle; this, in turn, limits their precision since type-based analyses occasionally rule out well-behaved programs. Present-day software poses even more challenges. Static verification often relies on having access to the program source code, which is not necessarily available when software is constructed from libraries or components subject to

* Supported by the doctoral student grant (No: 207055-051) and the MoVeMnt project (No: 217987-051) under the Icelandic Research Fund, the BehAPI project funded by the EU H2020 RISE under the Marie Skłodowska-Curie action (No: 778233), the ENDEAVOUR Scholarship Scheme (Group B, national funds), and the MIUR project PRIN 2017FTXR7S IT MATTERS.

third-party restrictions. Moreover, modern applications are increasingly developed in decentralised fashion, where the constituent parts are not always known pre-deployment. This tends to increase both the complexity of software *and* the resources required to verify it, while at the same time, decreasing the time available to conduct its verification. Lately, with the availability of large data volumes, cutting-edge software components rely on machine learning to adapt their behaviour without the need to be explicitly programmed. Analysing these types of software artifacts statically is difficult, not least because their internal representation is notoriously hard to understand.

Although the proverbial correctness cake cannot be had and eaten, slices of it may still be savoured *after* the program has been deployed. In certain cases, post-deployment techniques such as Runtime Verification (RV) [20,37] can be used instead of, or in tandem with, static techniques to increase correctness assurances about a program or System under Scrutiny (SuS). RV uses *monitors*—computational entities consisting of logically-distinct *instrumentation* and *analysis* units—to observe the execution of the SuS. Analysers, *i.e.,* sequence recognisers [13], are typically synthesised automatically from formal descriptions of correctness properties expressed in a specification logic.

When devising a RV tool, substantial effort is focussed on the specification language that is used to describe correctness properties, and the synthesis procedure which generates the analysis that runtime checks these properties [6,45,20]. Arguably, less attention is given to the instrumentation aspect, particularly, how the SuS is equipped to run with monitors, and the manner in which the computation of the SuS is *extracted* and *reported* for analysis. There is no one-size-fits-all solution to these challenges. For instance, inline instrumentation—the *de facto* technique employed by state-of-the-art RV [20,34]—relies on access to the program source or unobfuscated binary, thus obliging the RV monitors to be expressed in the same language as that of the SuS. The hallmark of a *flexible* RV tool is, therefore, its ability to support various instrumentation techniques to cater for the different scenarios where RV is used. The RV tool should also provide a *common interface* for describing *what* properties should be verified, agnostic of the underlying instrumentation mechanism dealing with the technicalities of *how* the verification is performed. This contributes to lowering the learning curve of the tool and facilitate its adoption.

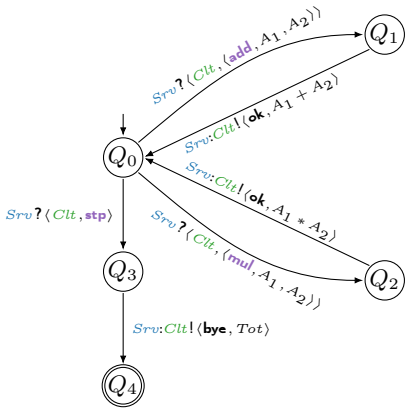
This paper presents the RV tool detectEr that addresses the *analysis* and *instrumentation* aspects of runtime monitoring. Our tool targets asynchronous component systems. It automatically synthesises *correct* analyser code from properties expressed in terms of the monitorable *safety* fragment of the modal μ-calculus. Since the correctness of the synthesised analysers is studied in prior work (see [39,1,5,17]), our account elaborates on the usability and instrumentation aspects of the tool. detectEr, developed on top of the Erlang [15,26] ecosystem, supports three instrumentation methods to cater for different SuS set-ups:

Inline: targets programs written in the Erlang language;
Outline: accommodates program binaries that are compiled for and run on the Erlang Virtual Machine (EVM), but whose source code is unavailable;

```
1  start() → spawn(calc_server, loop, [0]).
```

```
2  loop(Tot) →
3    receive
4      {Clt, {add, A1, A2}} →
5        Clt ! {ok, A1 + A2},
6        loop(Tot + 1);
7
8      {Clt, {mul, A1, A2}} →
9        Clt ! {ok, A1 * A2},
10       loop(Tot + 1);
11
12     {Clt, stp} → % Stop service.
13       Clt ! {bye, Tot}
14   end.
```

(a) `calc_server` implementation in Erlang



(b) Server-client interaction model

Fig. 1: Our calculator server and its abstraction in terms of symbolic actions

Offline: analyses recorded runs of programs that may execute outside the EVM.

We show how, from the same correctness specifications, detectEr is able to runtime monitor system components using these different instrumentation methods.

The paper is structured as follows. Sec. 2 introduces our running example that captures the typical interaction between concurrent processes, along with useful properties one may wish to runtime check on such systems. Secs. 3 and 4 focus on the specification logic used by detectEr and how this is synthesised into executable analysis code. Sec. 5 summarises the role the instrumentation has with respect to the runtime analysis, and the mechanism detectEr employs to identify the SuS components in need of monitoring. Secs. 6–8 overview the three instrumentation methods mentioned above, while sec. 9 concludes.

2 A Day in the Life

We consider an idiomatic calculator server that handles client requests for arithmetic computation. Our server can be naturally expressed as an actor (process) [12] that blocks, and waits for client requests sent as *asynchronous* messages. These messages are addressed to the server using its unique process ID (PID), and deposited in its *mailbox* that buffers multiple client requests. The server unblocks upon consuming a message from the mailbox. In our client-server protocol, messages contain the *type* of operation to be executed on the server, its *arguments* (if applicable), and the client PID to whom the corresponding server reply is addressed.

Our calculator server is implemented as the Erlang module, `calc_server`, in fig. 1a. The server logic is encapsulated in the function `loop(`*Tot*`)` that is *forked*

to execute as an independent process by the launcher invoking `start()`, line 1. Processes in Erlang are forked via the built-in function `spawn()`, parametrised on line 1 by the *module name*, `calc_server`, the name of the *function* to spawn, `loop`, and the list of *arguments* accepted by `loop`, `[0]`. The server process reads messages from its mailbox (line 3), and *pattern-matches* against the three types of operations requested by clients, *Clt*: *(i)* addition (add) and multiplication (mul) requests carry the operands A_1 and A_2, lines 4 and 8, and, *(ii)* stop (stp) requests that carry no arguments, line 12. Pattern variables *Clt*, A_1 and A_2 in fig. 1a are instantiated to concrete data in client request messages via pattern matching. Every request fulfilled by the server results in a corresponding reply that is sent to the PID of the client instantiated in variable *Clt*, lines 5, 9 and 13. Server replies carry the status *tag*, `ok` or `bye`, and the result of the requested operation. Parameter *Tot* of `loop()` is used by the server to track the number of client requests serviced, and is returned in reply to a stp request. The server loops on add and mul requests, incrementing *Tot* before recursing, lines 6 and 10; a stp request does not loop and *terminates* the server computation.

In the sequel, we focus on a system set-up consisting of one server and client to facilitate our exposition. The forked `loop(`*Tot*`)` function for some initial service count *Tot* induces a server runtime behaviour that can be abstractly described by the state transition model in fig. 1b. Transitions between the states of fig. 1b denote the computational steps that produce (visible) *program events* (*e.g.* event *Srv*:*Clt*`!{bye ,` *Tot*`}` that carries the *concrete* payload values *Srv*, *Clt* and *Tot*). There are a number of correctness properties we would like such behaviour to observe. For instance, we do not control the value *Tot* that the server loop is launched with and, therefore, could require the invariant

"The service request count returned on shutdown is never negative." (P_1)

Similarly, one would expect the safety properties

"Replies are always sent to the client indicated in the request" (P_2)

and *"A request for adding two numbers always returns their sum"* to hold, amongst others. The properties are data-dependent, which makes them hard to ascertain using static techniques such as type systems. Besides properties that reason on data, the implementation in fig. 1a is expected to comply with control properties, such as,

"Client requests are never serviced more than once", (P_3)

that describe the message exchanges between the server and client processes. All these properties are hard to ascertain without access to the source code.

3 I Want to Tell You

We overview the detectEr specification syntax, sHML [38,4,10], which is the *safety* logical fragment of the modal μ-calculus [43,44], and show how a selection of the properties in sec. 2 can be formally specified in this logic.

The logic. Specifications in sHML are defined over the states of transition models (such as the one of fig. 1b), and are generated from the following grammar:

$$\varphi \in \text{sHML} ::= \mathsf{tt} \;\; \text{(truth)} \quad | \quad \mathsf{ff} \;\; \text{(falsehood)} \quad | \quad x \;\; \text{(fix-point variable)}$$
$$| \; \textstyle\bigwedge_{i \in I} [p_i, c_i] \varphi_i \;\; \text{(conj. necessities)} \quad | \quad \mathsf{max} x.\varphi \;\; \text{(max. fix-point)}$$

sHML expresses recursive properties as maximal fix-point formulae $\mathsf{max} x.\varphi$, that bind free occurrences of x in φ. A central construct to sHML is the universal modal operator, $[p,c]\varphi$. To handle reasoning over event data, sHML modalities are augmented with *symbolic actions* [10], consisting of event patterns $p \in \textsc{Pat}$, and *decidable* constraints, $c \in \textsc{BExp}$. This is similar to how sets of actions are expressed in tools such as CADP [40] and mCRL 2 [22]. The pattern p contains data variables, $A,B,\ldots \in \textsc{Var}$, that bind free data variables in c, along with any other free variables in constraints of the *continuation* φ. The pair (p,c) describes a *concrete set* of actions, $a \in \textsc{Act}$ (*i.e.,* program events). An action a is in this set when: *(i)* p *matches* the shape of a, and maps the variables in p to the payload data in a as the substitution σ, and *(ii)* the *instantiated* constraint $c\sigma$ of p also holds. A state Q of the SuS (model) satisfies $[p,c]\varphi$ if the following holds: *whenever* Q transitions to state Q' with action a that is included in the set described by (p,c) with σ, then Q' *must* satisfy the instantiated continuation formula $\varphi\sigma$.

The logical variant [4,10] we use for detectEr combines necessities and conjunctions into one construct, $\bigwedge_{i \in I} [p_i,c_i]\varphi_i$, to denote $[p_1,c_1]\varphi_1 \wedge \ldots \wedge [p_n,c_n]\varphi_n$, $I = \{1,\ldots,n\}$ being a finite index set. Conjunctions assume that every pair (p_i,c_i) describes a *disjoint* set of actions to facilitate the generation of deterministic monitors [35,36]. detectEr supports the five action patterns of tbl. 1 that capture the lifecycle of, and interactions between the processes of the SuS. A `fork` action is exhibited by a process when it creates a child process; its dual, `init`, is exhibited by the corresponding child upon initialisation. Process `exit` actions signal termination, while `send` and `recv` describe interaction. The labelled state transition model of fig. 1b uses the actions `send` and `recv` from tbl. 1.

Example 1. Recall the SuS behaviour in fig. 1b. Formula φ_0 with symbolic action (p,c) describes a property requiring that a state does *not* exhibit an output event that consists of $\langle Ack, Tot \rangle$, acknowledged with bye and a negative total, Tot.

$$\bigwedge [\overbrace{Srv{:}Clt\,!\langle Ack, Tot\rangle}^{\text{pattern } p}, \overbrace{Ack = \mathsf{bye} \wedge Tot < 0}^{\text{constraint } c}]\,\mathsf{ff} \qquad (\varphi_0)$$

The universal modality states that, for *any* event satisfying the symbolic action (p,c) from a state Q, the state Q' it transitions to must then satisfy the continuation formula. No state can satisfy the continuation ff, and formula φ_0 can only be satisfied when Q *does not* exhibit the event described by (p,c). All the states in fig. 1b *trivially* satisfy this property (as there are no outgoing state transitions on (p,c) of formula φ_0) with the exception of Q_3. If this state exhibits the *concrete* event $\mathtt{pid_1{:}pid_2}\,!\langle \mathsf{bye}, -1\rangle$, it matches the pattern p, yielding

Action a	Action pattern p	Variables	Description
`fork` `init`	$P_1 \rightarrow P_2, M{:}F(A)$ $P_1 \leftarrow P_2, M{:}F(A)$	P_1	PID P_1 of the parent process forking P_2
		P_2	PID P_2 of the child process forked by P_1
		$M{:}F(A)$	Function signature forked by P_1
`exit`	$P_1 \star\star Dat$	P_1	PID P_1 of the terminated process
		Dat	Exit data, *e.g.* termination reason, *etc.*
`send`	$P_1{:}P_2!Req$	P_1	PID P_1 of the process issuing the request
		P_2	PID P_2 of the recipient process
		Req	Request payload, *e.g.* integers, tuples, *etc.*
`recv`	$P_2?Req$	P_2	PID P_2 of the recipient process
		Req	Request payload, *e.g.* integers, tuples, *etc.*

Tbl. 1: Trace event actions capturing the behaviour exhibited by the SuS

the substitution $\sigma = \{Srv \mapsto \texttt{pid}_1, Clt \mapsto \texttt{pid}_2, Ack \mapsto \mathsf{bye}, Tot \mapsto -1\}$. As $c\sigma$ also holds, then we can conclude that Q_3 violates formula φ_0. The formula φ_1 below extends φ_0 to one that is invariant for any state reachable from the current state; this formalises property P_1 from sec. 2.

$$\mathsf{max}\underline{x}.\bigwedge\left(\begin{array}{l}\overbrace{[Srv?Req,\ \top]\underline{x}}^{①},\ \overbrace{[Srv{:}Clt!\langle Ack, Tot\rangle,\ Ack = \mathsf{bye} \wedge Tot < 0]\mathsf{ff}}^{②},\\ \underbrace{[Srv{:}Clt!\langle Ack, Ans\rangle,\ Ack = \mathsf{ok} \vee (Ack = \mathsf{bye} \wedge Ans \geq 0)]\underline{x}}_{③}\end{array}\right) \qquad (\varphi_1)$$

Whereas ② corresponds to formula φ_0, ① and ③ cover the other possible actions produced in fig. 1b, recursing on the fix-point variable x. ■

Note that the formula variables *Srv*, *Clt*, *Tot*, *etc.* in eg. 1 are different to the program variables of fig. 1a bearing the same name. In our setting, program behaviour is observed as events, and formulae variables are used to pattern-match and reason about data in these events. We adopt the convention of naming formulae and program variables identically, merely to indicate the link between program and event data to readers.

The tool. The syntax used by detectEr deviates minimally from sHML. Concretely, the comma symbol delimiting patterns and constraints is dropped in favour of the when keyword, whereas vacuous constraints, *i.e.,* when $\top$, may be omitted. The tool also supports a shorthand notation for patterns to specify atomic values directly; these are *implicitly matched* against action data, *e.g.* subformula ② from eg. 1 can be abbreviated to $[Srv{:}Clt!\langle \mathsf{bye}, Tot\rangle$ when $Tot < 0]$. Moreover, redundant data variables can be replaced by the 'don't care' pattern,

(_), that matches *arbitrary* data values. This sugaring enables us to rewrite φ_1 from eg. 1 as:

$$\mathsf{max}\,\underline{x}.\bigwedge\left(\begin{array}{l}[\_?\_]\underline{x},\ [\_:\_!\langle \mathsf{bye}, Tot\rangle \ \mathsf{when}\ \ Tot<0]\mathsf{ff},\\ [\_:\_!\langle Ack, Ans\rangle \ \mathsf{when}\ \ Ack=\mathsf{ok}\vee(Ack=\mathsf{bye}\wedge Ans\geq 0)]\underline{x}\end{array}\right)$$

Example 2. Property P_2 from sec. 2 describes a fragment of the client-server interaction, asserting that server replies are always addressed to the clients issuing them. Unlike φ_1, this property induces *data dependency* across *nested formulae*.

$$\mathsf{max}\,\underline{x}.\bigwedge\left(\overbrace{[Srv_1?\langle Clt_1,\_\rangle]}^{①}\bigwedge\left(\begin{array}{l}\overbrace{[Srv_2{:}Clt_2!\_ \ \mathsf{when}\ \ Srv_1=Srv_2\wedge Clt_1\neq Clt_2]\mathsf{ff}}^{②},\\ \underbrace{[Srv_2{:}Clt_2!\_ \ \mathsf{when}\ \ Srv_1=Srv_2\wedge Clt_1=Clt_2]\underline{x}}_{③}\end{array}\right)\right) \quad (\varphi_2)$$

P_2 can be formalised as the formula φ_2, where the data dependency is expressed via the binders Srv_1 and Clt_1 in ①, which are then used in the constraint of sub-formulae ② and ③. The constraint $Srv_1 = Srv_2$ scopes our reasoning to a *single server* instance. Formula φ_2 is violated when $Clt_1 \neq Clt_2$ (since the continuation would need to satisfy ff), and recurs on x otherwise. Recall that the aforementioned comparisons between variable instantiations is possible since the substitution σ obtained from matching the symbolic action of modality ① *extends* to the context of sub-formulae ② and ③. ■

Example 3. Property P_3 specifies a control aspect of the client-server interaction, demanding that requests issued by clients are never serviced more than once. Formula φ_3 expresses this requirement via a guarded fix-point that recurs on x for sequences of `send`-`recv` actions; this captures normal server operation that corresponds to sub-formulae ① followed by ②, and then ④ followed by ②.

$$\bigwedge\overbrace{[\_?\_]}^{①}\mathsf{max}\,\underline{x}.\bigwedge\left(\underbrace{[Srv_1{:}Clt_1!\_]}_{②}\bigwedge\left(\begin{array}{l}\overbrace{[Srv_2{:}Clt_2!\_ \ \mathsf{when}\ \ Srv_1=Srv_2\wedge Clt_1=Clt_2]\mathsf{ff}}^{③},\\ \underbrace{[\_?\_]\underline{x}}_{④}\end{array}\right)\right) \quad (\varphi_3)$$

Formula φ_3 is violated when a `send` action matched by ② is followed by a second `send` action that is matched by ③. The constraint $Clt_1 = Clt_2$ in sub-formula ③ ensures that duplicate `send` actions concern the same recipient. ■

Our earlier formula φ_2 of eg. 2 does not account for the case where the server interacts with more than one client. It disregards the possibility of other interleaved events, that are inherent to concurrent settings where processes are unable to control when messages are received. For instance, while sub-formula ① matches an initial `recv` action, a second `recv` action (*e.g.* due to a second client

C_2 that interacts with the server) matches neither ② nor ③. This does not reflect the requirement of our original property P_2. The problem can be addressed by augmenting formulae with clauses that 'eat up' non-relevant actions.

$$\mathsf{max}\underline{x}.\bigwedge\left(\begin{array}{l}[Srv_1?\langle Clt_1,\_\rangle]\overbrace{\mathsf{max}\underline{y}.\bigwedge}^{①}\\ \qquad\left(\begin{array}{l}[Srv_2{:}Clt_2!\_ \text{ when } Srv_1 = Srv_2 \wedge Clt_1 \neq Clt_2]\mathsf{ff},\\ [Srv_2{:}Clt_2!\_ \text{ when } Srv_1 = Srv_2 \wedge Clt_1 = Clt_2]\underline{x},\\ \underbrace{[\_?\_]\underline{y}}_{②}\end{array}\right)\end{array}\right)$$

As the refinement of φ_2 above shows however, this bloats specifications, which is why we chose to scope our exposition to a single client-server set-up for the benefit of readers. Introducing the nested maximal fix-point ① and sub-formula ② filters `recv` actions by recursing on variable y; the rest of φ_2 is unaltered.

4 What Goes On

Often, post-deployment verification techniques such as RV, do *not* have access to the entire execution graph of a SuS, *e.g.* the transition model in fig. 1b. Instead, these are limited to the *trace* of (program) events that is generated by the *current execution* of the SuS. For instance, an execution might generate the trace of events '$\texttt{pid}_1?\langle\texttt{pid}_2,\mathsf{stp}\rangle.\texttt{pid}_1{:}\texttt{pid}_2!\langle\mathsf{bye},-1\rangle$', that corresponds to the (finite) path traversal $Q_0 \to Srv?\langle Clt,\mathsf{stp}\rangle \to Q_3 \to Srv{:}Clt!\langle\mathsf{bye},Tot\rangle \to Q_4$ in the transition model of fig. 1b. In traces, events consist of concrete values instead of variable placeholders, *e.g.* $\texttt{pid}_1$ instead of Srv, *etc.* This limitation can be problematic when verifying specifications that reason about entire SuS transition models, *e.g.* properties expressed in the μ-calculus [43,44,11], CTL [41,19], and other branching-time logics. Recent studies show that finite traces suffice to adequately verify a practically-useful subset of these properties, as long as the verification is confined to *either* determining satisfaction *or* violation [38,39,1,5,7] (not both). This is more commonly referred to as *specification monitorability* [39]. sHML, used in sec. 3 to encode properties P_1–P_3, has been shown to be a maximally-expressive subset of the μ-calculus for the runtime analysis of violations. This means that *(i)* any program that violates a property expressed as a sHML formula can be detected at runtime, *(ii)* any μ-calculus property whose violations can be detected at runtime can be expressed as a sHML formula.

From specification to analysis. detectEr synthesises *automata-like* analysers in Erlang from sHML; these inspect trace events *incrementally* and reach *irrevocable* verdicts. An analyser flags a *rejection* verdict when it processes a trace exhibiting the program behaviour that *violates* a property of interest—crucially, it never flags verdicts associated with the *satisfaction* of the property [39,1,7]. Intuitively, this is because the trace observed at runtime can never provide *enough*

information to rule out the existence of violating behaviour in *other* execution branches of the program. The synthesised analyser code embeds this reasoning: when a trace event is *not* included in the set of actions denoted by the symbolic action of a necessity modality, an *inconclusive verdict* is flagged.

Our synthesis translates a sHML specification to Erlang code encoded as a higher-order function, tasked with the analysis of trace events. This function accepts an event as input, and returns a new function of the same kind that performs the residual analysis following the event just processed. The synthesis, $[\![-]\!]$, that maps sHML constructs to Erlang syntax is as follows:

$$[\![\mathsf{ff}]\!] \triangleq \texttt{(fun (\_) -> io:format("Rejection") end) ()}$$

$$[\![\max x.\varphi]\!] \triangleq \texttt{(fun x() ->} [\![\varphi]\!] \texttt{ end) ()} \qquad [\![x]\!] \triangleq \texttt{x()}$$

$$[\![\bigwedge_{i\in I}[p_i,c_i]\varphi_i]\!] \triangleq \begin{cases} \texttt{fun(}p_1\texttt{) when }c_1 \texttt{ ->} [\![\varphi_1]\!] \\ \quad\vdots \\ \texttt{(}p_n\texttt{) when }c_n \texttt{ ->} [\![\varphi_n]\!] \\ \texttt{(\_) -> (fun (\_) -> io:format("Stopped") end) ()} \\ \texttt{end} \end{cases}$$

In $[\![-]\!]$, ff is translated into an anonymous function that flags rejection, modelling formula violations; tt is not synthesised into analysis code since it can never be violated. Maximal fix-point formulae are translated to *named* functions that can be referenced by $[\![x]\!]$. The conjunction of necessities, $\bigwedge_{i\in I}[p_i,c_i]\varphi_i$, maps naturally to a sequence of function clauses, where the pattern p_i matches the shape of the trace event, and the constraint c_i—expressed as an Erlang guard [26]—operates on variables bound in p_i *and* those instantiated in the parent function scope. Inconclusive verdicts are modelled via the catch-all clause `(_)` that matches any events *other than* those described by the clauses, `fun(`p_i`) when` c_i. The order of clauses in Erlang *does* matter, and affects our synthesis in two ways: *(i)* it conveniently allows us to handle the inconclusive verdict case using a catch-all clause at the end of function definitions; *(ii)* the mutually-exclusive symbolic actions in a necessity conjunction allows us to synthesise them in the order specified, without affecting the commutativity of conjunctions. Note that the synthesis applies the generated functions, *i.e.,* `()`, to unfold them once.

Fig. 2 depicts the behaviour of the analyser that is synthesised from formula φ_2 of eg. 2. It consists of two states, Q_0, Q_1, the rejection verdict state ✗, and the inconclusive verdict state *?*. The transition from Q_0 to Q_1 in fig. 2 corresponds to the modality $[Srv_1?\langle Clt_1,\_\rangle]$, ① in formula φ_2, while the transitions between Q_1 and ✗, Q_1 and Q_0 express sub-formulae ② and ③. The auxiliary transitions from states leading to *?* correspond to the catch-all `(_)` clause inserted by the synthesis for conjuncted necessities. Fig. 2 illustratively labels these transitions by the complement of the set of actions from a given state. For example, the symbolic action set $\textsc{Pat}\setminus\{\_?\langle\_,\_\rangle\}$ from Q_0 to *?* matches *anything but* `recv` events; `recv` events are, in turn, matched by $Srv_1?\langle Clt_1,\_\rangle$ labelling the transition between Q_0 and Q_1. Verdict irrevocability, a prevalent RV require-

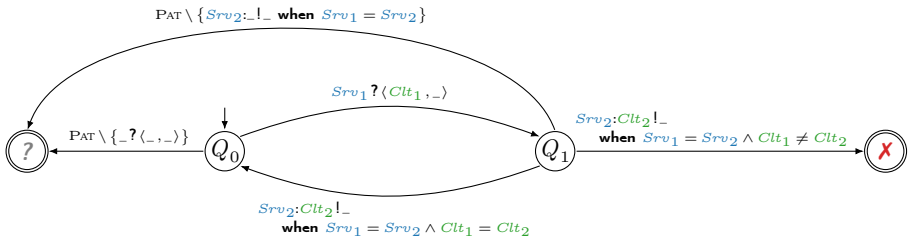


Fig. 2: Abstract model of the analyser synthesised from formula φ_2

ment [6], is modelled by the detectEr synthesis in terms of final states (✗ and *?* in fig. 2). The analysis stops when a final state is reached.

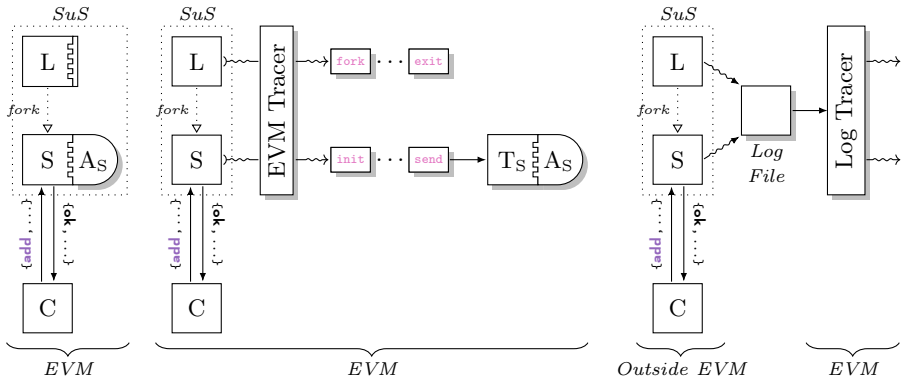
Specification, in practice. detectEr processes sHML formulae specified in plain text files. The syntax follows the one given in sec. 3, albeit with two adaptations: *(i)* the keyword `and` is used in lieu of $\bigwedge$, and, *(ii)* we adopt the Erlang operators for writing boolean constraint expressions, *e.g.* `=:=` instead of $=$, `andalso` instead of $\wedge$, `orelse` instead of $\vee$, *etc.* Analysers resulting from the synthesis, $[\![-]\!]$, are compiled to binaries to be packaged with the SuS executables.

5 The Magical Mystery Tour

Instrumentation is central to RV. It refers to the extraction of the computation of interest in the form of a sequence of trace events from an executing program, *and* its reporting to the runtime analysis discussed in sec. 3. Formulae can be rendered unverifiable at runtime when the program events they assume cannot be extracted and reported by the instrumentation. The instrumentation also plays a role in dropping *extraneous* events that can infiltrate the trace being observed and potentially, interfere with the analysis.

What to monitor. We provide the meta keywords with and monitor to target the SuS component of interest for a particular specification. The with keyword picks out the signature of the function that is forked whereas the monitor keyword defines the property to be analysed. For example, to runtime verify the behaviour of the calculator server of fig. 1a against formula φ_2, we write:

$$
\begin{array}{l}
\mathsf{with} \\
\quad \texttt{calc\_server:loop}(\_) \\
\mathsf{monitor} \\
\quad \mathsf{max}\underline{x}.\bigwedge\left(\begin{array}{l}
[Srv_1?\langle Clt_1,\_\rangle]\bigwedge \\
\qquad \left(\begin{array}{l}
[Srv_2{:}Clt_2!\_ \text{ when } Srv_1 = Srv_2 \wedge Clt_1 \neq Clt_2]\mathsf{ff}, \\
[Srv_2{:}Clt_2!\_ \text{ when } Srv_1 = Srv_2 \wedge Clt_1 = Clt_2]\underline{x}
\end{array}\right)
\end{array}\right)
\end{array}
\qquad (\varphi_2')
$$



(a) Weaving (b) Tracing via the EVM (c) Tracing via log file

Fig. 3: Inline, outline and offline instrumentation methods offered by detectEr

From an instrumentation standpoint, with establishes the set of trace events corresponding to the SuS component it targets, thus enabling the specification to *abstract* from the events generated by other components. This helps to keep the size of specifications compact whenever possible. In using with, formula φ_2' need not account for superfluous events (*e.g.* those of another server component) that tend to make the specification exercise tedious and error-prone.

How to monitor. detectEr offers three instrumentation methods, inline (sec. 6), outline (sec. 7), and offline (sec. 8), to cater for different situations where the RV is conducted. These methods are depicted by the three set-ups of fig. 3 that are instantiated with our calculator server, labelled by S, and its parent launcher process, labelled by L. Inlining *statically* instruments system components with the analyser, A_S, which then executes as *part of* the SuS. Outline instrumentation *decouples* the SuS components from the extraction and analysis of trace events by way of the tracing infrastructure provided by the EVM. The offline set-up extends the latter notion to a SuS that (possibly) executes *outside* the EVM, mirroring the same architecture of fig. 3b to enjoy the same outline arrangement (elided from fig. 3c). The with keyword directs detectEr to instrument the analyser code over the relevant SuS components regardless of the instrumentation method used, to ensure that the same set of trace events is reported to the runtime analysis. This makes the analysers generated by our synthesis of sec. 4 agnostic of the underlying instrumentation.

6 Come Together

Inlining [34] is the most efficient instrumentation method detectEr offers. While it assumes access to the source code of the SuS, it carries advantages such as

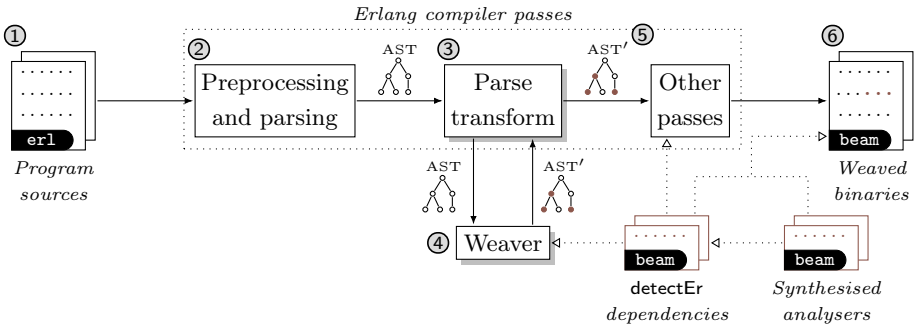


Fig. 4: Instrumentation pipeline for inlined program monitoring via weaving

low runtime overhead [32,31] and immediate detections [20]. detectEr instruments invocations to synthesised analysers via *code injection* by manipulating the program abstract syntax tree (AST). This procedure is depicted in fig. 4. In step ①, the Erlang program source code is preprocessed and parsed into the corresponding AST, step ②. The Erlang compilation pipeline includes a *parse transformation* phase [26], step ③, that offers an optional hook to allow the AST to be processed externally, prior to code generation. Our custom-built weaver leverages this mechanism to transform the program AST in step ④ and produce the modified AST in step ⑤; this is subsequently compiled by the Erlang compiler into the program binary, step ⑥. The compilation phase depends on the detectEr core modules *and* analyser binaries, as does the SuS, once it executes.

Instrumentation, in one go. Step ④ in fig. 4 performs two transformations on the program AST. The first transformation initialises an analyser. It weaves code instructions that *store* the function encoding of the synthesised analyser (refer to sec. 4) in the process dictionary (PD) of the instrumented process (PDs are process-local, mutable key-value stores with which Erlang actors are initialised). The weaver identifies `spawn()` calls that carry the function signature to be executed as a process. It then *replaces* the `spawn()` call with a counterpart which accepts an *anonymous wrapper* function that *(i)* stores the analyser function in the PD, and, *(ii)* applies the function specified inside the original `spawn()` call. Fig. 5a recalls the function `start()` that forks our calculator server loop, line 1. The corresponding weaved version of its AST—given as Erlang code for illustration in fig. 5a—performs the initialisation of *(i)* and *(ii)*. Line 3 contains (omitted) boilerplate logic that determines whether a particular `spawn()` call should be instrumented. The meta keyword with from sec. 5 is used to this end: it results in the synthesis of auxiliary code that enables the weaver to effect this judgement. For example, the specification '`with calc_server:loop(_)...`' of formula φ_2' informs the weaver to initialise the analyser only for the function name `loop` forked by the invocation of `spawn()` on line 1 in fig. 5a. In line 8, the encoding of the analyser function, `AnlFun0`, is stored in the PD. The signature

```
1 start() → spawn(calc_server, loop, [0]).
```

```
2 start() →
3   AnlFun0 = ... % Load analysis logic.
4   P1 = self(),
5   MFA = {Mod0, Fun0, Args0},
6   P0 = spawn(
7     fun() →
8       anl:embed(AnlFun0),
9       anl:dsp({init, self(), P1, MFA}),
10      apply(Mod0, Fun0, Args0)
11    end)
12  anl:dsp({fork, self(), P0, MFA}),
13  P0.
```

(a) Server initialised with analyser function

```
1 loop(Tot) →
2 receive
3   M2 = {Clt, {add, A1, A2}} →
4     anl:dsp(recv, self(), M2),
5     (P1 = Clt) ! M1 = {ok, A1 + A2},
6     anl:dsp(send, self(), P1, M1),
7     loop(Tot + 1);
8
9   M4 = {Clt, {mul, A1, A2}} →
10    ...
11
12  M6 = {Clt, stp} → % Stop service.
13    ...
14 end.
```

(b) Weaved analysis code in server loop

Fig. 5: Transformations to the AST of the `calc_server` program (shown as code)

used in the original `spawn()` call on line 1 is applied on line 10, where `Mod0`, `Fun0`, and `Args0` are respectively instantiated to values `calc_server`, `loop`, and `[0]` by the boilerplate logic on line 3 (omitted).

The second transformation decorates the program AST with calls at points of interest: these correspond to the actions catalogued in tbl. 1. Each call constructs an intermediate trace event description that is *dispatched* to the analyser. Lines 9 and 12 in fig. 5a construct events `init` and `fork`, and dispatch them to the analyser using the function `anl:dsp()` exposed by the core `detectEr` modules. The events `recv` and `send` are analogously handled on lines 4 and 6 in fig. 5b.

Our weaver performs the two transformations outlined above regardless of whether monitoring is required by the SuS. This induces a *modular design* where the SuS is weaved *once*, while the analyser binaries may be *independently* regenerated, *e.g.* to refine or add sHML specifications. Updates in these binaries can afterwards be put into effect by restarting the weaved SuS. To determine whether to analyse a trace event, the dispatcher implementation `anl:dsp()` internally checks against the PD whether an analyser function has been initialised for the instrumented process. When the analyser function is initialised, `anl:dsp()` *applies* the function to the event, and *saves* the resulting unfolded analyser function back to the PD; otherwise, `anl:dsp()` discards the event. An irrevocable verdict is reached by the analyser function once its application to an event returns the internal value that encodes ✗ or ?.

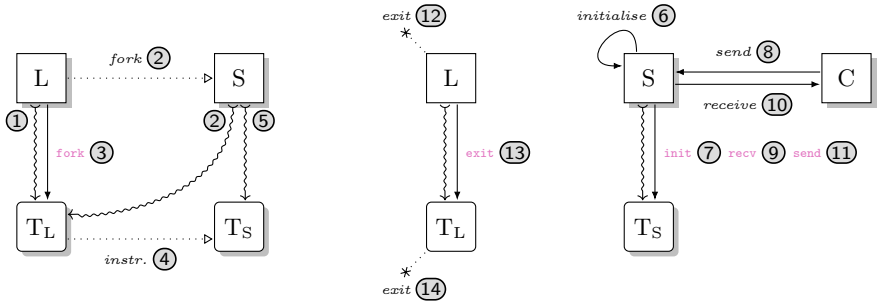
Weaving makes it difficult to extract `exit` trace events, since abnormal termination due to crashes cannot be easily anticipated. This limits the ability to runtime check correctness properties concerning process termination. An instrumentation approach via external observation easily sidesteps this restriction.

7 Tell Me What You See

Outlining *externalises* the acquisition and analysis of SuS trace events. It relies on the tracing infrastructure provided by the EVM [26], and supports any software component that is developed for the EVM ecosystem, *e.g.* Erlang, Elixir [42] and Clojerl [33]. Fig. 3b in sec. 5 shows the outlined set-up for our calculator server example. Outlining uses *tracers*, actor processes tasked with the handling of trace events exhibited by the SuS. Tracers *register* with the EVM tracing infrastructure to be notified of process events in connection to the actions of tbl. 1. Our outlining algorithm instruments tracers *on-demand*, depending on what processes need to be analysed. This approach departs considerably from inline instrumentation in sec. 6, and rather than weaving the SuS statically, outlining defers the decision of what to instrument until runtime.

While outline instrumentation tends to induce higher runtime overhead, it offers a number of benefits over inlining. It takes a *non-invasive* approach that leverages the EVM to trace components without modification, making it easy to enable and disable the runtime analysis without the need of restarts or redeployments. By decoupling the SuS and tracer components, outlining induces a degree of *partial failure*—a faulty analyser does not compromise the running system, nor does a crashed system component affect the external tracer. As a result of this arrangement, `exit` trace events can be detected, giving us the full expressiveness with respect to the system actions of tbl. 1. The implementation of an adequate outline monitoring set-up comes with its own set of challenges. For example, the instrumentation should be engineered to *scale* in line with the SuS, while the runtime analysis of trace events is underway. It has to contend with the race conditions (*e.g.* trace event reordering) that arise from the asynchronous execution of the SuS and tracer components. Scalability requires the instrumentation to *explicitly* manage garbage collection, where redundant tracer processes are discarded to minimise resource consumption. Inline instrumentation is spared these complications since the analysis logic is weaved directly in SuS processes. Although our outline instrumentation algorithm handles these aspects, we refrain from providing further detail in this presentation. Interested readers are encouraged to consult [8] for more details.

Instrumentation, as we go. The EVM tracing infrastructure enables processes to register their interest in receiving trace event messages from other processes. Erlang provides the built-in function `trace()`, that processes may invoke to enable and disable process tracing dynamically at runtime. Our tracers from fig. 3b leverage this functionality to fork other tracers, and scale the RV set-up as the SuS executes. We configure the EVM tracing to *automatically assign* the tracer of an already-traced SuS process to the children it forks [26]. Using this as a default setting allows us to analyse *groups* of processes as one component. The with keyword guides the targeting of which processes tracers need to track and analyse. By contrast to inlining—where the set of trace events of a component is *implicitly* determined as a byproduct of weaving—outlining must *actively isolate* processes from a group to assign dedicated tracers. Recall the specification



(a) S forked by L and traced by T_S (b) C interacting with S; events received by T_S

Fig. 6: Outline instrumentation for the calculator server (analysers omitted)

'with `calc_server:loop(_)...`' of formula φ_2'. This instructs our outline instrumentation to set up an independent tracer process for the calculator server loop forked by `spawn()` on line 1 in fig. 5a.

Tracers are programmed to react to `fork` and `exit` events in the trace. Fig. 6 illustrates how the process creation sequence of the SuS is exploited to instrument a dedicated tracer for our calculator server. A tracer instruments other tracers whenever it encounters `fork` events. The initial RV configuration is shown in fig. 6a, where the root tracer, T_L, is assigned to the launcher process, L, in step ①. L forks the server function `loop()` to execute as the process S which is automatically assigned the same tracer T_L, as steps ② both indicate. Subsequently, T_L instruments a *new* tracer, T_S, when it processes the `fork` trace event due to L in step ③. The data carried by `fork` contains the PID of the forked process (see tbl. 1) that designates the SuS process to be instrumented, S, in this case. At this point, T_S takes over the tracing of S from the root tracer T_L by invoking `trace()` to handle S independently of T_L, steps ④ and ⑤. T_S resumes its analysis of S, receiving the `init` event in step ⑦; this is followed by `recv` in step ⑨ as a result of the service request issued by the client, C, in step ⑧. In a similar way, the service reply sent by the server to C in step ⑩ results in `send` being exhibited by S and received by T_S in step ⑪. Process L eventually exits after the fork completes, step ⑫. The ensuing `exit` event in step ⑬ is interpreted by the root tracer T_L as the cue to self-terminate in step ⑭. This garbage collection measure maintains the lowest possible runtime overhead.

8 I'm Only Sleeping

We extend the notion of outline instrumentation to the offline case where the SuS may potentially run outside the EVM. To support offline instrumentation, detectEr implements a middleware that emulates the EVM tracing infrastructure, while preserving the configuration mentioned in sec. 7, *i.e.,* where forked system

processes automatically inherit the tracer assigned to their parent. This enables detectEr to employ the *same* outline instrumentation algorithm for offline monitoring. Offline set-ups are generally the slowest in terms of verdict detection, by comparison to the inline and outline forms of instrumentation. This stems from the dependence outline instrumentation has on the timely availability of pre-recorded runtime traces that are subject to external software entities such as files, databases, and the SuS itself. However, the outline set-up and SuS can reside on different hardware since they are mutually detached. Such an arrangement makes overhead issues secondary.

Fig. 3c from sec. 5 overviews our offline arrangement. It mirrors the set-up in fig. 3b: the only difference lies in how the offline tracing infrastructure obtains events. Our Log Tracer component in fig. 3c exposes a `trace()` function, providing the same EVM feature subset relevant to outlining. The implementation relies on log files as the medium through which the SuS can communicate trace events to the offline set-up. It can process log files with complete system executions, or *actively* monitor files for changes to dynamically dispatch events to tracers while the SuS executes and writes events to file. Offline tracing supports the event actions in tbl. 1; these carry the event data and are assumed to follow a pre-defined format. For instance, the offline event description `fork(`$\texttt{pid}_1$`,`$\texttt{pid}_2$`,{calc_server,loop,[0]})` is mapped to the action $\texttt{pid}_1 \rightarrow \texttt{pid}_2, \texttt{calc\_server:loop([0])}$ by the Log Tracer of fig. 3c. Our file-based approach to collecting SuS events is motivated by the fact that file logging is widely-adopted in practice, and is offered by popular frameworks such as Lager [21] for Erlang, Log4J 2 [16] for Java [46], and the Python [49] logging facility. Besides logging, events may also be extracted from the SuS via other tracing frameworks, *e.g.* DTrace [23], LTTng [28], and OpenJ9 Trace [30].

9 Here, There and Everywhere

This paper presents detectEr, a RV tool that analyses program correctness *post-deployment* against properties expressed in a logic that has been traditionally used for *static* verification [22,40,14]. Secs. 5–8 describe how detectEr can flexibly runtime check the same specifications via three instrumentation methods. The tool can be found at `https://duncanatt.github.io/detecter`.

Future work. We intend to asses the merits of our three instrumentation methods in terms of the multi-faceted overhead metrics proposed by [9]. We also plan to extend detectEr to handle sHML specifications where conjunctions and universal modalities can be treated as separate logical constructs [4,3,39,18,17,24,25]. This facilitates the composition of properties via conjunctions, *e.g.* formulae φ_1–φ_3 from sec. 3 can be combined as $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$ to synthesise one global monitor. Although detectEr focusses on properties that are known to be runtime monitorable, new results argue that monitoring can be systematically extended to the entire class of regular properties, albeit, with possibly weakened detection guarantees [7,2]. We aim to incorporate these results within this tool.

References

1. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A.: Monitoring for silent actions. In: FSTTCS. LIPIcs, vol. 93, pp. 7:1–7:14 (2017)
2. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A.: A Framework for Parameterized Monitorability. In: FoSSaCS. LNCS, vol. 10803, pp. 203–220 (2018)
3. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Kjartansson, S.Ö.: On the Complexity of Determinizing Monitors. In: CIAA. LNCS, vol. 10329, pp. 1–13 (2017)
4. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Kjartansson, S.Ö.: Determinizing Monitors for HML with Recursion. JLAMP **111**, 100515 (2020)
5. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: Adventures in Monitorability: From Branching to Linear Time and Back Again. Proc. ACM Program. Lang. **3**(POPL), 52:1–52:29 (2019)
6. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: An Operational Guide to Monitorability with Applications to Regular Properties. Softw. Syst. Model. **20**(2), 335–361 (2021)
7. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: The Best a Monitor Can Do. In: CSL. LIPIcs, vol. 183, pp. 7:1–7:23 (2021)
8. Aceto, L., Attard, D.P., Francalanza, A., Ingólfsdóttir, A.: A Choreographed Outline Instrumentation Algorithm for Asynchronous Components. CoRR **abs/2104.09433** (2021)
9. Aceto, L., Attard, D.P., Francalanza, A., Ingólfsdóttir, A.: On Benchmarking for Concurrent Runtime Verification. In: FASE. LNCS, vol. 12649, pp. 3–23 (2021)
10. Aceto, L., Cassar, I., Francalanza, A., Ingólfsdóttir, A.: On Runtime Enforcement via Suppressions. In: CONCUR. LIPIcs, vol. 118, pp. 34:1–34:17 (2018)
11. Aceto, L., Ingólfsdóttir, A.: Testing Hennessy-Milner Logic with Recursion. In: FoSSaCS. LNCS, vol. 1578, pp. 41–55 (1999)
12. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A Foundation for Actor Computation. JFP **7**(1), 1–72 (1997)
13. Alpern, B., Schneider, F.B.: Recognizing Safety and Liveness. Distributed Comput. **2**(3), 117–126 (1987)
14. Andersen, J.R., Andersen, N., Enevoldsen, S., Hansen, M.M., Larsen, K.G., Olesen, S.R., Srba, J., Wortmann, J.K.: CAAL: Concurrency Workbench, Aalborg Edition. In: ICTAC. LNCS, vol. 9399, pp. 573–582 (2015)
15. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)
16. ASF: Log4J 2 (2021), https://logging.apache.org/log4j/2.x
17. Attard, D.P., Francalanza, A.: A Monitoring Tool for a Branching-Time Logic. In: RV. LNCS, vol. 10012, pp. 473–481 (2016)
18. Attard, D.P., Francalanza, A.: Trace Partitioning and Local Monitoring for Asynchronous Components. In: SEFM. LNCS, vol. 10469, pp. 219–235 (2017)
19. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
20. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to Runtime Verification. In: Lectures on RV, LNCS, vol. 10457, pp. 1–33. Springer (2018)
21. Basho: Lager (2021), https://github.com/basho/lager
22. Bunte, O., Groote, J.F., Keiren, J.J.A., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, W., Wijs, A., Willemse, T.A.C.: The mCRL2 Toolset for Analysing Concurrent Systems - Improvements in Expressivity and Usability. In: TACAS (2). LNCS, vol. 11428, pp. 21–39 (2019)

23. Cantrill, B.: Hidden in Plain Sight. ACM Queue **4**(1), 26–36 (2006)
24. Cassar, I., Francalanza, A., Attard, D.P., Aceto, L., Ingólfsdóttir, A.: A Suite of Monitoring Tools for Erlang. In: RV-CuBES. Kalpa Publications in Computing, vol. 3, pp. 41–47 (2017)
25. Cassar, I., Francalanza, A., Said, S.: Improving Runtime Overheads for detectEr. In: FESCA. EPTCS, vol. 178, pp. 1–8 (2015)
26. Cesarini, F., Thompson, S.: Erlang Programming: A Concurrent Approach to Software Development. O'Reilly Media (2009)
27. Clarke, E.M., Klieber, W., Novácek, M., Zuliani, P.: Model Checking and the State Explosion Problem. In: LASER Summer School. LNCS, vol. 7682, pp. 1–30 (2011)
28. Desnoyers, M., Dagenais, M.R.: The LTTng Tracer : A Low Impact Performance and Behavior Monitor for GNU/Linux. Tech. rep., École Polytechnique de Montréal (2006)
29. Dijkstra, E.W.: Chapter I: Notes on Structured Programming, p. 1–82. Academic Press Ltd. (1972)
30. Eclipse/IBM: Openj9 (2021), https://www.eclipse.org/openj9
31. Erlingsson, Ú.: The Inlined Reference Monitor Approach to Security Policy Enforcement. Ph.D. thesis, Cornell University (2004)
32. Erlingsson, Ú., Schneider, F.B.: SASI Enforcement of Security Policies: A Retrospective. In: NSPW. pp. 87–95 (1999)
33. Facorro, J.: Clojerl language (2021), http://clojerl.org
34. Falcone, Y., Krstic, S., Reger, G., Traytel, D.: A Taxonomy for Classifying Runtime Verification Tools. In: RV. LNCS, vol. 11237, pp. 241–262 (2018)
35. Francalanza, A.: Consistently-Detecting Monitors. In: CONCUR. LIPIcs, vol. 85, pp. 8:1–8:19 (2017)
36. Francalanza, A.: A Theory of Monitors. Information and Computation p. 104704 (2021). https://doi.org/https://doi.org/10.1016/j.ic.2021.104704
37. Francalanza, A., Aceto, L., Achilleos, A., Attard, D.P., Cassar, I., Della Monica, D., Ingólfsdóttir, A.: A Foundation for Runtime Monitoring. In: RV. LNCS, vol. 10548, pp. 8–29 (2017)
38. Francalanza, A., Aceto, L., Ingólfsdóttir, A.: On Verifying Hennessy-Milner Logic with Recursion at Runtime. In: RV. LNCS, vol. 9333, pp. 71–86 (2015)
39. Francalanza, A., Aceto, L., Ingólfsdóttir, A.: Monitorability for the Hennessy-Milner Logic with Recursion. FMSD **51**(1), 87–116 (2017)
40. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. Int. J. Softw. Tools Technol. Transf. **15**(2), 89–107 (2013)
41. Jr., E.M.C., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
42. Jurić, S.: Elixir in Action. Manning (2019)
43. Kozen, D.: Results on the Propositional μ-Calculus. In: ICALP. LNCS, vol. 140, pp. 348–359 (1982)
44. Larsen, K.G.: Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion. TCS **72**(2&3), 265–288 (1990)
45. Leucker, M., Schallhart, C.: A Brief Account of Runtime Verification. JLAP **78**(5), 293–303 (2009)
46. Loy, M., Niemeyer, P., Leuck, D.: Learning Java: An Introduction to Real-World Programming with Java. O'Reilly Media (2020)
47. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley (2011)
48. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
49. Python: Logging Facility for Python (2021), https://docs.python.org/3/library/logging.html