Science of Computer Programming 232 (2024) 103031

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Original software publication

A monitoring tool for linear-time μHML

Luca Aceto [b,c], Antonis Achilleos [b], Duncan Paul Attard [a,b,*], Léo Exibard [b], Adrian Francalanza [a], Anna Ingólfsdóttir [b]

[a] University of Malta, Msida, Malta
[b] Reykjavik University, Reykjavik, Iceland
[c] Gran Sasso Science Institute, L'Aquila, Italy

ARTICLE INFO

Keywords:
Runtime verification
Linear-time properties
Monitor synthesis

ABSTRACT

We present detectEr, a monitoring tool that targets software applications written for Erlang/OTP. The tool runtime checks specifications expressed in a safety fragment of the linear-time modal μ-calculus called MAXHML^D, used to describe properties about the current system execution. Our technical development is founded on previous theoretical results that are lifted to a first-order setting, where systems produce executions containing events that carry data. We overview the main features of detectEr, showing how properties can be flexibly written and synthesised as executable Erlang monitors that can be instrumented with the running system.

Code metadata

Code metadata description	
Current code version	0.9
Permanent link to code/repository used for this code version	https://github.com/ScienceofComputerProgramming/SCICO-D-22-00294
Permanent link to Reproducible Capsule	https://zenodo.org/record/6418234#.ZEvVZy8Ro04
Legal Code License	GPL-3.0 license
Code versioning system used	git
Software code languages, tools, and services used	Erlang
Compilation requirements, operating environments and dependencies	Linux, macOS, Windows, and Erlang
If available, link to developer documentation/manual	https://duncanatt.github.io/detecter
Support email for questions	duncanatt@gmail.com

1. Motivation and significance

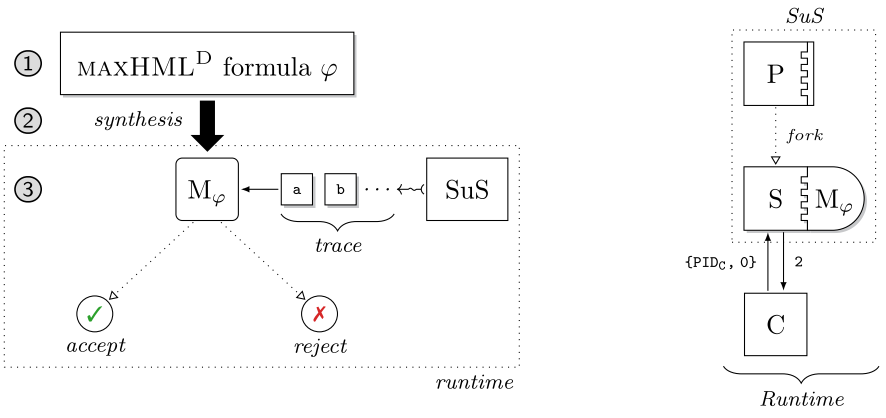
This paper presents detectEr, a runtime verification (RV) tool that targets software applications written for the Erlang/OTP [1]. detectEr builds on the theoretical foundations of [2,3], which are lifted to a first-order setting [4] to handle systems that operate on data. Our tool adopts the logic fragment MAXHML^D to specify properties about the *current* system execution [5]. Fig. 1a illustrates

The code (and data) in this article has been certified as Reproducible by Code Ocean: https://codeocean.com/. More information on the Reproducibility Badge Initiative is available at https://www.elsevier.com/physical-sciences-and-engineering/computer-science/journals.

* Corresponding author.

E-mail addresses: luca@ru.is, luca.aceto@gssi.it (L. Aceto), antonios@ru.is (A. Achilleos), duncan.attard.01@um.edu.mt, duncanpa17@ru.is (D.P. Attard), leoe@ru.is (L. Exibard), afra1@um.edu.mt (A. Francalanza), annai@ru.is (A. Ingólfsdóttir).

https://doi.org/10.1016/j.scico.2023.103031
Received 14 November 2022; Received in revised form 21 September 2023; Accepted 21 September 2023
Available online 27 September 2023



(a) Property φ of the *current* execution of the SuS (b) Monitor inlining via weaving

Fig. 1. detectEr runtime monitoring set-up.

the detectEr set-up, where $\textsc{maxHML}^{\mathrm{D}}$ specifications φ (step ①) are synthesised as Erlang monitors, M_{φ} (step ②), and instrumented with the system under scrutiny (SuS). Monitors incrementally analyse the current execution of the SuS (step ③) to reach one of two *irrevocable* judgements: *accept* verdicts (✓) that correspond to satisfactions of $\textsc{maxHML}^{\mathrm{D}}$ formulae φ, and *reject* verdicts (✗) that correspond to violations of φ. detectEr instruments monitors via *inlining* by weaving them with the SuS to induce minimal runtime overhead. Fig. 1b depicts an instrumented system where the monitor M_{φ} of Fig. 1a executes together with process S.

Most RV tools use temporal logics based on LTL, *e.g.* [6–14]. Despite its widespread adoption, LTL has limited expressiveness [15, 3]. For instance, it cannot describe properties such as *'every* odd *position in the execution satisfies some proposition p'* (formula φ_2 in sec. 2.2 expresses this requirement). We show how such commonly-occurring properties can be flexibly expressed and monitored for with detectEr. To the best of our knowledge, the only other tool that runtime checks linear-time properties of Erlang systems is the proof-of-concept software Elarva that uses automata-based specifications [16]. However, Elarva suffers from severe scalability issues due to its centralised architecture. By contrast, detectEr can scale under high loads while inducing feasible runtime overhead; see [17] for details.

2. Overview

Properties in detectEr are specified using a *maximally-expressive* runtime monitorable fragment [2] of the linear-time modal μ-calculus [5] called $\textsc{maxHML}^{\mathrm{D}}$. This logic describes properties over *infinite* executions, u, that abstractly represent complete system runs.

$$
\begin{array}{rlll}
\varphi,\psi \in \textsc{maxHML}^{\mathrm{D}} ::= & \mathsf{tt} \quad \text{(truth)} & \mid \mathsf{ff} & \text{(falsehood)} \\
\mid & \varphi \vee \psi \quad \text{(disjunction)} & \mid \varphi \wedge \psi & \text{(conjunction)} \\
\mid & \langle p,e\rangle\varphi \quad \text{(possibility)} & \mid [p,e]\varphi & \text{(necessity)} \\
\mid & \mathsf{max}\, X.(\varphi) \quad \text{(greatest fp.)} & \mid X & \text{(rec. variable)}
\end{array}
$$

Syntax The $\textsc{maxHML}^{\mathrm{D}}$ syntax assumes a denumerable set of propositional variables, $X,Y \in \textsc{PVar}$. In addition to the standard Boolean constructs, the logic can express recursive properties as greatest fixed point formulae, $\mathsf{max}\, X.(\varphi)$, that *bind* the free occurrence of X in φ. The existential and universal modalities, $\langle p,e\rangle\varphi$ and $[p,e]\varphi$, respectively express the dual notions of *possibility* and *necessity*.

Semantics Modal constructs are interpreted w.r.t. *symbolic actions* that enable the reasoning over the data carried by system actions (*i.e.*, events), $\alpha \in \textsc{Act}$. Symbolic actions, written as (p,e), consist of a pattern, p, and a *decidable* Boolean constraint expression, e. Patterns are tuples, $\langle \ell, x_1, \ldots, x_n\rangle$, comprised of the *action label*, ℓ, and pairwise-distinct binders, $x_1,\ldots,x_n$, that range over Erlang data types, *e.g.* tuples, lists, process identifiers (PIDs), *etc.* Binders bind the free occurrences of $x_1,\ldots,x_n$ in e, along with any other free variables in constraints of the formula continuation φ. These bindings induce the usual notion of *scoping*, whereby variables in different sub-formulae can refer to others, making it possible to define relationships between events along the trace. A symbolic action (p,e) describes a set of system actions, called the *action set*. An event is in this set if: (i) the pattern p *matches* the shape of the action α, returning a *substitution* π that maps every variable in p to corresponding data values in α, and (ii) the *instantiated* Boolean constraint expression $e\pi$ holds. The existential modal formula $\langle p,e\rangle\varphi$ describes all the executions αu where α is in the action set (p,e) *and* u satisfies the continuation $\varphi\pi$. Dually, $[p,e]\varphi$ describes all the traces αu that, *if* prefixed by any α from the action set (p,e), the continuation formula u *then* satisfies $\varphi\pi$. This means that, in particular, the universal modal formula $[p,e]\varphi$ is trivially satisfied whenever α is *not* in the action set (p,e). Elaborate formulae are built from the basic constituents $\langle p,e\rangle\mathsf{tt}$ and $[p,e]\mathsf{ff}$. Intuitively,

Table 1
Actions capturing the behaviour exhibited by Erlang processes.

Action α	Action pattern p	Variables	Description
fork initialise	$x_1 \rightarrow x_2, y_1 : y_2(y_3)$	x_1	PID of the parent process forking x_2
	$x_2 \leftarrow x_1, y_1 : y_2(y_3)$	x_2	PID of the child process forked by x_1
		y_1, y_2, y_3	Function signature forked by x_1
error	$x_1 \star y_1$	x_1	PID of the erroneous process
		y_1	Error datum, *e.g.* error reason, *etc.*
send	$x_1 : x_2 ! y_1$	x_1	PID of the process sending the message
		x_2	PID of the recipient process
		y_1	Message datum, *e.g.* integer, tuple, *etc.*
receive	$x_2 ? y_1$	x_2	PID of the recipient process
		y_1	Message datum, *e.g.* integer, tuple, *etc.*

$\langle p, e\rangle$tt stipulates that a system *can* exhibit any action α in the action set described by (p, e), whereas $[p, e]$ff asserts that *no* action in (p, e) is exhibited. Stated formally, $\langle p, e\rangle$tt is satisfied exactly by all the executions that start with an action α in the action set (p, e); dually, $[p, e]$ff is satisfied exactly by all the executions that do not start with α in (p, e). The meaning of the existential and universal modal constructs $\langle p, e\rangle\varphi$ and $[p, e]\varphi$ is, therefore, the standard one in modal logic; see, for instance, [18]. Finally, the set of traces satisfying the greatest fixed point formula $\max X.(\varphi)$ is the union of all the post-fixed point solutions of the function induced by the formula φ.

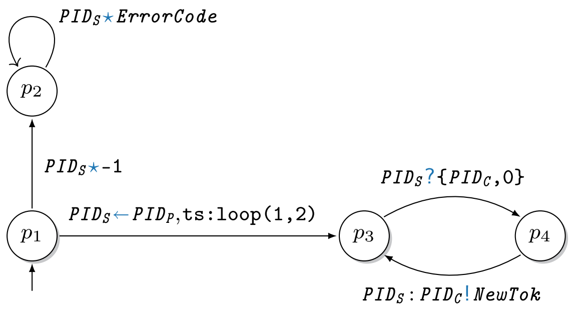
2.1. Adapting the logic to Erlang

We fix the action label set $\ell \in \{\rightarrow, \leftarrow, \star, !, ?\}$ to model the lifecycle of, and interaction between Erlang processes. The *fork* action, $\rightarrow$, is exhibited by a process when it creates a child; its dual, $\leftarrow$, is exhibited by the child process upon *initialisation*. An *error* action, $\star$, signals abnormal process behaviour; *send* and *receive*, respectively $!$ and $?$, denote process communication. Table 1 details the patterns related to these labelled actions, along with the data payload they carry.

The syntax of the specification logic that detectEr uses deviates slightly from the one given above for MAXHML^{D} to make it more readable. Concretely, we drop the comma symbol that delimits patterns p and Boolean constraint expressions e in symbolic actions in favour of the keyword `when`, writing (p, e) as p `when` e. A Boolean constraint expression e can be elided when $e = \texttt{true}$, as can redundant pattern variables by using the 'don't care' pattern '_'. detectEr follows the Erlang syntactic conventions, using capitalised names for variables. Symbolic actions in modal constructs are enclosed in braces $\{\dots\}$. Our tool also borrows the Erlang guard syntax to specify Boolean constraint expressions. For instance, the operators `or` and `and` are used in lieu of the connectives $\vee$ and $\wedge$, the relational operator == instead of =, =/= instead of $\neq$, *etc.* See [1] for details.

2.2. Specifying properties in detectEr

Consider the Erlang implementation of a reactive token server p_1, modelled in Fig. 2, which issues client programs with numeric identification tokens that they use as an alias to write to a remote logging service. Clients request tokens by sending the command `0`, which the server fulfils by replying with a new token, *NewTok*. The token server itself also uses the remote logging service and is, thus, launched with its *reserved* identification token `1`. Our server starts when its main function, `loop`, declared in the Erlang module `ts` is invoked (state p_1, line 2 in Fig. 2b). From p_1, it transitions to p_3 (line 4), exhibiting the initialisation event $PID_S \leftarrow PID_P$,`ts:loop(1,2)`; the placeholders PID_S and PID_P respectively denote the PID values of the token server process and of the parent process forking the server. At p_3, the server accepts client requests, consisting of the tuple $\{PID_C, 0\}$, where PID_C is the PID of the client, and `0`, the command requesting a new identification token, line 5. From state p_4, the server replies with the



(a) Erlang token server model

```
1  start(Tok) →
2   spawn(ts, loop, [Tok, Tok + 1]).

3  loop(Tok, NewTok) when Tok = 1 →
4   receive
5   {Clt, 0} →
6    Clt ! NewTok,
7    io:format("Token ~p.",
8     [NewTok]),
9    loop(Tok, NewTok + 1)
10  end.
```

(b) Erlang code in `ts` module

Fig. 2. Erlang implementation of the token server.

new token, *NewTok* on line 6, and transitions back to p_3. This client-server interaction emits the server events $\mathit{PID}_S ? \{\mathit{PID}_C, 0\}$ and $\mathit{PID}_S : \mathit{PID}_C ! \mathit{NewTok}$. When the server fails to load, it transitions with a status of `-1` to the sink p_2, thereafter exhibiting *undefined behaviour*, shown as the error events $\mathit{PID}_S \star \texttt{-1}$ and $\mathit{PID}_S \star \mathit{ErrorCode}$ in Fig. 2a.

Example 1 There are various properties we want the current execution of our token server of Fig. 2 to observe. For instance, we require that *'the server is initialised correctly* and *with the identifier token `1`'*, expressed as:

```
 1  with                                                        (φ1)
 2    ts:loop(_, _)
 3  check
 4      [{Srv ⋆ Code when Code == -1}] ff
 5    and
 6      <{Prnt ← Srv, ts:loop(Tok, NewTok) when Tok == 1}> tt
```

The symbolic action $\{\mathit{Srv} \star \mathit{Code}\ \texttt{when}\ \mathit{Code}\ \texttt{==}\ \texttt{-1}\}$ in the left conjunct of φ_1 on line 4, defines the set $\{\mathit{Srv} \star \texttt{-1}\}$ of external system actions. Necessity modal formulae $[p\ \texttt{when}\ e]\varphi$ state that for any trace prefix α in the action set defined by (p, e), the trace continuation u must satisfy φ. However, *no* trace satisfies `ff`. This means that for server traces *not* to violate φ_1, they must start with actions $\alpha \notin \{\mathit{Srv} \star \texttt{-1}\}$. In other words, either (i) the action α must not match the pattern p, or (ii) if α matches p, the instantiated Boolean constraint $e\pi$ must not hold. When the server p_1 exhibits an error at start up, the pattern $\mathit{Srv} \star \mathit{Code}$ yields the substitution $\pi = [^{\mathit{PID}_S}/_{\mathit{Srv}}, ^{\texttt{-1}}/_{\mathit{Code}}]$ and the instantiated Boolean constraint $(\mathit{Code}\ \texttt{==}\ \texttt{-1})\pi$ holds, leading to a violation of φ_1. The dual argument can be made for the second conjunct of the formula on line 6: if an initialisation event that carries the token value `1` matches the pattern $\mathit{Prnt} \leftarrow \mathit{Srv}, \texttt{ts:loop}(\mathit{Tok}, \mathit{NewTok})$ the instantiated constraint $\mathit{Tok}\ \texttt{==}\ \texttt{1}$ holds, and the existential modal sub-formula is satisfied. Simultaneously, `[{Srv ⋆ Code when Code == -1}]` is *trivially* satisfied since the pattern $\mathit{Srv} \star \mathit{Code}$ does not match the initialisation event. ■

The above example uses the '`with` *MFA* `check` φ' syntax provided by detectEr to pick out processes of the SuS whose execution is to be runtime checked against the formula φ. *MFA* designates the Erlang Module exposing the Function that is launched with the specified Arguments. In this instance, `with` matches the pattern of the main function `loop` from our token server module `ts` that accepts two arguments. These arguments have been replaced by the 'don't care' pattern '_', and are shown as `(_,_)` in φ_1 on line 2. The redundant variables *Srv*, *Prnt*, and *NewTok* included in φ_1 to elucidate the various placeholders in patterns can be likewise elided.

Example 2 Amongst the executions satisfying φ_1 are those where the server accidentally returns its identification token `1` in reply to client requests. We therefore demand that *'the server private token `1` is not leaked in client replies'*. Formula φ_2 expresses this recursive property in a general way, *i.e.*, it does not hardcode the token `1`. The Boolean constraints $e = \mathrm{tt}$ are elided.

```
 1  with                                                        (φ2)
 2    ts:loop(_, _)
 3  check
 4    [{_ ← _, ts:loop(Tok, _)}] max X. (
 5      [{_ ? _}] (
 6          [{_ : _ ! NewTok when Tok == NewTok}] ff
 7        and
 8          [{_ : _ ! NewTok when Tok =/= NewTok}] X
 9      )
10    )
```

The symbolic action $\{\_ \leftarrow \_, \texttt{ts:loop}(\mathit{Tok}, \_)\}$ in the first necessity on line 4 matches initialisation events. It instantiates the variable *Tok* with the token value `1` that the server is launched with, and substitutes every occurrence of *Tok* in the formula continuation on lines 5–8. Conceptually, this makes this residual sub-formula equivalent to:

```
11  max X. (                                                    (φ'2)
12    [{_ ? _}] (
13        [{_ : _ ! NewTok when 1 == NewTok}] ff
14      and
15        [{_ : _ ! NewTok when 1 =/= NewTok}] X
16    )
17  )
```

The greatest fixed point sub-formula describes the client-server interaction loop. It states that this request-response behaviour holds *invariantly* throughout the system execution. The symbolic action `{_ ? _}` in the universal modality on line 12 matches server receive events resulting from incoming client requests. Lines 13 and 15 check the value of the token issued by the server. The first conjunct, `[_:_ ! NewTok when 1 == NewTok] ff`, asserts that the server token stored in *NewTok* can be anything other than the value `1`, for if it were `1` (*i.e.*, `[_:_ ! 1 when 1 == 1] ff`), φ_2 would be violated. The second conjunct on line 15 specifies the case where the token value stored in *NewTok* does not match the value of `1`, prompting the recursive sub-formula to unfold anew. This unfolding yields back the formula φ_2'. Note that a *fresh* scope for the variable *NewTok* is created upon every recursive unfolding of φ_2', enabling *NewTok* to be instantiated to a new token value. ■

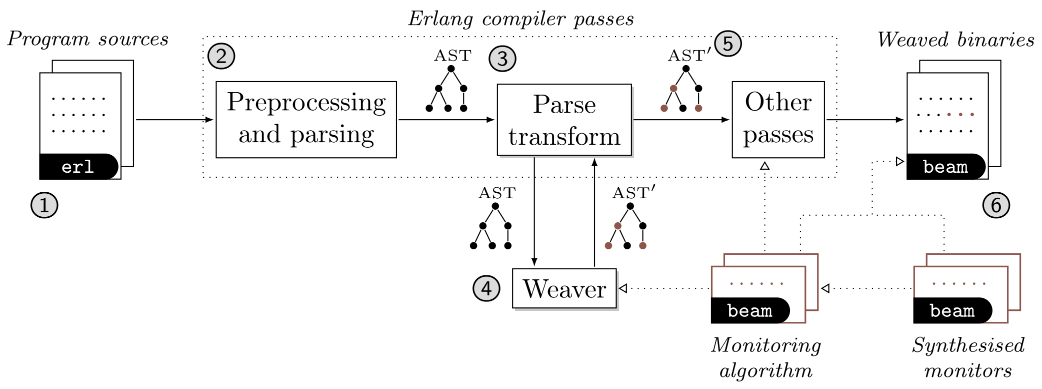


Fig. 3. Instrumentation pipeline for inlined monitors using Erlang source-level weaving.

Formula φ_2 compares actions at *every* odd position in the trace against the one at the head (see line 4). Such formulae cannot be expressed in LTL, as remarked in sec. 1.

2.3. Monitor synthesis

The synthesis procedure detectEr uses generates executable Erlang monitors from maxHML^D formulae. Our translation from formulae to monitors is performed in three stages. First, a formula is parsed into its equivalent abstract syntax tree (AST). This is then passed to the code generator that visits each of its nodes, mapping every logical construct to a corresponding Erlang description. The resulting monitor is encoded as an AST to simplify its handling. In the final stage, this AST is forwarded to the Erlang compiler that generates the monitor source code or BEAM [1] executable. Our synthesis encodes symbolic actions in modal constructs as Erlang function clauses with *guards*. This carries two benefits. On the one hand, it enables us to streamline the synthesis and support most of the Erlang data types, along with its full range of Boolean constraint expression syntax. On the other, organising symbolic actions as functions leverages the lexical *scoping* of Erlang, which facilitates our management of pattern variables and their use in Boolean constraints. detectEr synthesises monitors as *templates* whose variables become dynamically instantiated at runtime. These templates are interpreted by our monitoring algorithm: it progressively reduces them based on the trace events analysed until a verdict is reached. Further technical details regarding the synthesis procedure and monitoring algorithm detectEr uses can be found in our main paper [4].

2.4. Monitor instrumentation

The inline instrumentation performed by detectEr assumes access to the source code of the SuS. It instruments invocations to our monitoring algorithm via code injection by manipulating the program AST. We leverage the Erlang compilation pipeline, which includes a *parse transformation* phase [19] that offers an optional hook through which the AST can be processed externally, prior to code generation. This program code modification procedure is outlined in Fig. 3. In step ①, the Erlang program source code is preprocessed and parsed into the corresponding AST, step ②. Subsequently, the AST is passed to the parse transformer in step ③: this invokes our custom-built weaver, step ④, that produces the modified AST′ in step ⑤. The decorated AST is compiled by the Erlang compiler into the program binary in the final stage, step ⑥. Note that this compilation phase, as well as the executing SuS, require two dependencies, namely, the detectEr core modules that include the monitoring algorithm, and the synthesised Erlang monitors.

3. Using detectEr

maxHML^D formulae such as φ_1 and φ_2 are written in plain-text script files with a `.hml` extension. Scripts are compiled to generate monitors that the SuS can be instrumented with, following the workflow described next.

3.1. Compiling monitor scripts

A maxHML^d script file is compiled into a monitor using the detectEr function `maxhml_eval:compile/2` (by convention, `mod_name:fun_name/arity` identifies Erlang functions [19]). This function accepts two arguments:

1. the path pointing to the maxHML^D `.hml` script file, and
2. a list of options that control how the monitor is generated.

detectEr script files contain at least one specification that must be terminated with a full-stop; multiple specifications can be placed in the same file as long as these are separated by commas.

Suppose the formula φ_2 is specified in the script file `prop_no_leak.hml`. The scripted formula can be synthesised into its corresponding monitor by launching the Erlang shell and invoking `compile`.

```
1  user@local:detecter/examples/erlang$ erl -pa ../../detecter/ebin ebin
2  Erlang/OTP 23 [erts-11.2.1] [source] [64-bit] [smp:4:4] [ds:4:4:10]
3
4  Eshell V11.2.1 (abort with ^G)
5  1> maxhml_eval:compile("prop_no_leak.hml", [{outdir, "ebin"}]).
6  ok
```

The command generates the Erlang monitor file `prop_no_leak.beam` in the indicated output directory, `ebin`.

3.2. Inlining

Our server implementation of Fig. 2b is given in the Erlang module `ts.erl`. We show how this can be instrumented with the monitor synthesised earlier in sec. 3.1. detectEr offers the functions `lin_weaver:weave_file/3` and `lin_weaver:weave/3` for this purpose, which inject the SuS with monitors and additional instructions that extract trace events. The first function, `lin_weaver:weave_file/3` instruments a single file; `lin_weaver:weave/3` instruments a directory of files. Both variants of `weave` accept:

1. the path where the Erlang source file (or directory) to be weaved resides,
2. the function `mfa_spec/1` of the monitor to be weaved, and,
3. a list of options that controls how the instrumented system is generated.

The *hard-coded* function `mfa_spec/1` is generated automatically by detectEr as the entry point that launches the monitor runtime analysis.

Here we use `lin_weaver:weave_file/3` to weave the module `ts.erl`. We specify the arguments to `lin_weaver:weave_file/3`, namely, (i) the relative path of the file `ts.erl`, (ii) entry function of the monitor `prop_no_leak`, and (iii) the output directory where the generated monitor is to be written, `ebin`.

```
7   2> lin_weaver:weave_file("ts.erl", fun prop_no_leak:mfa_spec/1, ↩
8        [{outdir, "ebin"}]).
9   {ok, ts, []}
10  3> _
```

The weaving step produces the compiled token server binary, `ts.beam`, and loads it into the code path of the Erlang shell.

3.3. Launching the system

Once the token server is instrumented, it can be executed normally by launching it from the Erlang shell. We recall that the detectEr and synthesised monitor binaries must be loaded in the code path of the Erlang environment, otherwise the instrumented system fails to load; see sec. 2.4.

```
10  3> Pid = ts:start(1).
11  <0.94.0>
12  4> _
```

If the token server is implemented correctly, a new token request by a client instructs the server to issue a *valid* token (*i.e.*, any value other than `1`). This should not trigger the monitor. A new token is requested by sending (`!`) the command `0` to the token server with `Pid`. The server returns the token `2`.

```
12  4> Pid ! {self(), 0}.
13  {<0.82.0>, 0}
14  Token 2.
15  5> _
```

Our token server implementation may also be *incorrect*, in which case the server private token `1` is leaked in client replies. This results in the monitor flagging a reject verdict that corresponds to a violation of formula φ_2.

```
15  5> Pid ! {self(), 0}.
16  {<0.82.0>, 0}
17  Violation: After analysing event {send, <0.94.0>, <0.82.0>, 1}.
18  Token 1.
19  6> Pid ! {self(), 0}.
20  {<0.82.0>, 0}
21  Violation: After analysing event {send, <0.94.0>, <0.82.0>, 3}.
22  Token 3.
```

Observe that further requests to the server trigger the *same* verdict, even if *valid* tokens are returned from this point onward (*e.g.* the token value `3` is issued by the server on line 22, but a violation is flagged regardless). Persisting the monitoring verdict reflects its irrevocability, where once announced, cannot be changed even when analysing future events; see sec. 1.

3.4. Case studies

Our tool has been empirically evaluated in [17] using synthetic benchmarks to quantify the overhead induced by monitors. In the same work, detectEr is used to monitor an off-the-shelf third-party webserver called Cowboy [20]. Cowboy delegates its socket management to Ranch (a socket acceptor pool for TCP protocols [21]). In the companion version of this paper, we also validate the expressiveness of the MAXHML^{D} by runtime checking fragments of the Cowboy-Ranch interaction protocol. Further details can be found in [4]. detectEr has also been used to verify parts of the RAFT [22] consensus algorithm written in Elixir [23].

4. Conclusion

This paper showcases detectEr, a monitoring tool that targets software applications developed for Erlang/OTP. Our tool runtime checks linear-time specifications that describe properties about the current system execution. The examples considered show how the logic can express recursive properties, and how symbolic actions enable the reasoning on data carried by trace events. We outline how detectEr synthesises executable monitors that are instrumented via inlining to minimise runtime overhead. Our case studies [17,4,23] demonstrate that the logic is sufficiently expressive to describe properties of real-world software. More information about the tool can be found on the detectEr website [24].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

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

References

[1] J. Armstrong, Programming Erlang: Software for a Concurrent World, Pragmatic Bookshelf, 2007.
[2] L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, K. Lehtinen, Adventures in monitorability: from branching to linear time and back again, Proc. ACM Program. Lang. 3 (POPL) (2019) 52.
[3] L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, K. Lehtinen, An operational guide to monitorability with applications to regular properties, Softw. Syst. Model. 20 (2) (2021) 335–361.
[4] L. Aceto, A. Achilleos, D.P. Attard, L. Exibard, A. Francalanza, A. Ingólfsdóttir, A monitoring tool for linear-time μHML, in: COORDINATION, in: LNCS, vol. 13271, Springer, 2022, pp. 200–219.
[5] D. Kozen, Results on the propositional μ-calculus, in: ICALP, in: LNCS, vol. 140, 1982, pp. 348–359.
[6] A. Bauer, M. Leucker, C. Schallhart, Runtime verification for LTL and TLTL, ACM Trans. Softw. Eng. Methodol. 20 (4) (2011) 14.
[7] A. Bauer, M. Leucker, C. Schallhart, Comparing LTL semantics for runtime verification, J. Log. Comput. 20 (3) (2010) 651–674.
[8] A. Bauer, Y. Falcone, Decentralised LTL monitoring, Form. Methods Syst. Des. 48 (1–2) (2016) 46–93.
[9] B. Bonakdarpour, P. Fraigniaud, S. Rajsbaum, D.A. Rosenblueth, C. Travers, Decentralized asynchronous crash-resilient runtime verification, in: CONCUR, in: LIPIcs, vol. 59, Schloss Dagstuhl - Leibniz-Zentrum Für Informatik, 2016, pp. 16:1–16:15.
[10] D.A. Basin, F. Klaedtke, E. Zalinescu, Failure-aware runtime verification of distributed systems, in: FSTTCS, in: LIPIcs, vol. 45, Schloss Dagstuhl - Leibniz-Zentrum Für Informatik, 2015, pp. 590–603.
[11] K. Havelund, D. Peled, Runtime verification: from propositional to first-order temporal logic, in: RV, in: LNCS, vol. 11237, Springer, 2018, pp. 90–112.
[12] K. Sen, A. Vardhan, G. Agha, G. Rosu, Efficient decentralized monitoring of safety in distributed systems, in: ICSE, 2004, pp. 418–427.
[13] K. Sen, A. Vardhan, G. Agha, G. Rosu, Decentralized runtime analysis of multithreaded applications, in: IPDPS, IEEE, 2006.
[14] T. Scheffel, M. Schmitz, Three-valued asynchronous distributed runtime verification, in: MEMOCODE, 2014, pp. 52–61.
[15] P. Wolper, Temporal logic can be more expressive, Inf. Control 56 (1/2) (1983) 72–99.
[16] C. Colombo, A. Francalanza, R. Gatt, Elarva: a monitoring tool for Erlang, in: RV, in: LNCS, vol. 7186, Springer, 2011, pp. 370–374.
[17] L. Aceto, D.P. Attard, A. Francalanza, A. Ingólfsdóttir, On benchmarking for concurrent runtime verification, in: FASE, in: LNCS, vol. 12649, 2021, pp. 3–23.
[18] P. Blackburn, M. de Rijke, Y. Venema, Modal Logic, Cambridge Tracts in Theoretical Computer Science, vol. 53, Cambridge University Press, 2001.
[19] F. Cesarini, S. Thompson, Erlang Programming: A Concurrent Approach to Software Development, O'Reilly Media, 2009.
[20] L. Hoguin, Cowboy, https://ninenines.eu, 2020.
[21] L. Hoguin, Ranch, https://ninenines.eu, 2020.
[22] D. Ongaro, J.K. Ousterhout, In search of an understandable consensus algorithm, in: USENIX Annual Technical Conference, 2014, pp. 305–319.
[23] M.A. Le Brun, D.P. Attard, A. Francalanza, Graft: general purpose RAFT consensus in elixir, in: Erlang Workshop, 2021, pp. 2–14.
[24] D.P. Attard, Detecter, https://duncanatt.github.io/detecter/detecter-linear-time/setting-up-detecter.html, 2022.