A Monitoring Tool for Linear-Time
μHML

Luca Aceto\textsuperscript{2,3}, Antonis Achilleos\textsuperscript{2}, Duncan Paul Attard\textsuperscript{1,2(\textsuperscript{\textcopyright})}, Léo Exibard\textsuperscript{2}, Adrian Francalanza\textsuperscript{1}, and Anna Ingólfsdóttir\textsuperscript{2}

\textsuperscript{1} University of Malta, Msida, Malta
\{duncan.attard.01,afra1\}@um.edu.mt
\textsuperscript{2} Reykjavik University, Reykjavik, Iceland
\{luca,antonios,duncanpa17,leoe,annai\}@ru.is
\textsuperscript{3} Gran Sasso Science Institute, L’Aquila, Italy
luca.aceto@gssi.it

Abstract. We present the implementation of a prototype tool that runtime checks specifications written in a maximally-expressive safety fragment of the linear-time modal \(\mu\)-calculus called \textsc{maxHML}. Our technical development is founded on previous results that give a compositional synthesis procedure for generating monitors from \textsc{maxHML} formulae. This paper instantiates this synthesis to a first-order setting, where systems produce executions containing events that carry data. We augment the logic with predicates over data, and extend the synthesis procedure to generate executable monitors for Erlang, a general-purpose programming language. These monitors are instrumented via inlining to induce minimal runtime overhead. Our monitoring algorithm also maintains information, which it uses to explain how verdicts are reached.

Keywords: Runtime verification · Linear-time specifications · Monitor synthesis

1 Introduction

Runtime Verification (RV) \cite{13,17,36,52} is a lightweight verification technique that dynamically checks the current execution to determine whether a System under Scrutiny (SuS) satisfies or violates some correctness stipulation. These stipulations are generally expressed using a specification logic to formally describe the behaviour the SuS should observe. RV synthesises specifications into monitors: computational entities that are instrumented with the SuS to

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

© IFIP International Federation for Information Processing 2022
https://doi.org/10.1007/978-3-031-08143-9_12
analyse its execution (expressed as a trace of events) incrementally, and reach 
verdicts that cannot be retracted when observing future events. Figure 1 depicts 
this set-up.

The vast majority of existing work and tooling efforts on RV focus on checking 
specifications that describe properties of system executions (Fig. 1a). Most of 
these studies are conducted in the context of temporal logics that are based on 

LTL (e.g., [19,21,23–25,41,58–60]). Despite its widespread use, LTL has limited 
expressiveness. For instance, it cannot express properties such as ‘every even 
position in the execution satisfies some proposition p ’ [4,62].

The modal $\mu$-calculus with a linear-time interpretation [49] has been shown to 
embed several other standard logics, including LTL, making it suitable to express 
a wider range of properties. Recent work [3,4] studies monitors for $\mu$HML [7,51], 
a reformulation of the $\mu$-calculus. One aspect that sets that work apart from the 
ones cited above is the modular approach the authors adopt in their technical 
development. Rather than redefining the semantics of the logic to assimilate the 
notion of monitoring verdicts, their study identifies runtime monitorable syntac-
tic fragments of $\mu$HML, delineating between its semantics on the one hand, and 
the operational semantics of monitors on the other. The authors define a synthesis 
procedure that generates correct monitors from these fragments. They also 
establish a correspondence between monitor acceptance (resp. rejection) verdicts 
and satisfactions (resp. violations) in the logic, and show that the fragments iden-
tified are maximally-expressive, i.e., characterise all monitorable properties. This 
separation of concerns provides a principled approach to RV tool construction.

This paper presents the implementation of a prototype tool that builds on 
the theoretical foundations of [3,4]. It adopts the fragment \textsc{maxHML} of $\mu$HML 
that is used to specify safety properties on the current system execution. The 
study in [3,4] considers regular properties, which arguably limits its applicability 
to a broader setting where executions contain events that carry data [17,35]. 
We, therefore, lift the results of that study to a first-order setting, and extend 
the logic and synthesis procedure with predicates over data. Our adaptation of 
the monitor synthesis closely follows the one of [3,4], giving us high assurances 
that the corresponding monitors are correct. A facet that is often overlooked in 
RV is verdict explainability, where tools justify how monitoring judgements are 
reached [39]. Instantiations of this concept are commonplace in related fields. 
For example, model checking tools [48] produce diagnostic traces that explain 
why a model fails to satisfy some specification; in the same spirit, programming 
language frameworks capture runtime information and present it in the form of 
stack traces or core dumps. We take a first step in this direction, and engineer 
our monitoring algorithm to derive an explanation that is constructed using 
the monitor operational semantics of [3,4]. Since our prototype tool does not yet 
perform space optimisations for the purposes of explainable verdicts, this feature 
is intended for debugging or offline use.

In a parallel research direction, we study other monitorable $\mu$HML frag-
ments in a branching-time setting [1,2,37,38], where the logic describes prop-
erties about the computation graph of programs (Fig. 1b). Much of this body
of work is concretised as detectEr [12–14,26], a runtime monitoring tool for concurrent Erlang programs. The material we propose in this paper complements the one in detectEr, contributing towards one tool that can runtime check specifications under their linear- and branching-time interpretations. Our contributions are:

(i) We extend the logic maxHML with data support, and show how this is used to specify properties on the current system execution, Sect. 2;
(ii) adapt the monitor synthesis and operational semantics of [3,4] to enable monitors to reach verdicts based on the data carried by trace events, Sect. 3;
(iii) discuss the challenges encountered when instantiating the synthesis in Sect. 3 to a general-purpose programming language, and overview the technique we use to instrument monitors that induce minimal runtime overhead, Sect. 4.

2 The Logic

We overview our chosen logic, maxHML, a maximally-expressive syntactic fragment of μHML used to describe safety properties of system executions [3]. It assumes a set of external actions $\alpha, \beta \in \text{Act}$, together with a distinguished internal action $\tau \notin \text{Act}$ that represents one internal step of computation. External actions range over values taken from some (potentially infinite) data domain, $\mathbb{D}$. Executions, also referred to as traces, are infinite sequences of external system actions that abstractly represent complete system runs. We reserve the metavariables $t, u \in \text{Act}^\omega$ to represent infinite traces, and use $\alpha t$ to denote an infinite trace that starts with $\alpha$ and continues with $t$.

Figure 2 shows our extension of maxHML, called maxHML$^d$, with predicates over data. Its syntax assumes a denumerable set of logical variables,
MAXHML$^d$ Syntax

$\varphi, \psi \in \text{MAXHML}^d := \text{tt} | \text{ff} | \langle x, e \rangle \varphi | \left[ x, e \right] \varphi | \varphi \lor \psi | \varphi \land \psi | \max X.(\varphi) | X$

MAXHML$^d$ Semantics

\[
\begin{align*}
\llbracket \text{tt}, \sigma \rrbracket & \triangleq \text{Act}^\omega \\
\llbracket \text{ff}, \sigma \rrbracket & \triangleq \emptyset \\
\llbracket \langle x, e \rangle \varphi, \sigma \rrbracket & \triangleq \{ t | (\exists u. \exists \alpha. t = \alpha u \text{ and } e[\alpha/x] \downarrow \text{true and } u \in \llbracket \varphi[\alpha/x], \sigma \rrbracket) \} \\
\llbracket \left[ x, e \right] \varphi, \sigma \rrbracket & \triangleq \{ t | (\forall u. \forall \alpha. (t = \alpha u \text{ and } e[\alpha/x] \downarrow \text{true}) \text{ implies } u \in \llbracket \varphi[\alpha/x], \sigma \rrbracket) \} \\
\llbracket \varphi \lor \psi, \sigma \rrbracket & \triangleq \llbracket \varphi, \sigma \rrbracket \cup \llbracket \psi, \sigma \rrbracket \\
\llbracket \varphi \land \psi, \sigma \rrbracket & \triangleq \llbracket \varphi, \sigma \rrbracket \cap \llbracket \psi, \sigma \rrbracket \\
\llbracket \max X.(\varphi), \sigma \rrbracket & \triangleq \bigcup \{ S | S \subseteq \llbracket \varphi[\sigma[X \mapsto S]], \sigma \rrbracket \} \\
\llbracket X, \sigma \rrbracket & \triangleq \sigma(X)
\end{align*}
\]

Fig. 2. Syntax and linear-time semantics for the logic MAXHML$^d$

In addition to the standard Boolean constructs, the logic can express recursive properties as greatest fixed point formulae, $\max X.(\varphi)$, that bind the free occurrences of $X$ in $\varphi$. The existential and universal modalities $\langle x, e \rangle \varphi$ and $\left[ x, e \right] \varphi$ express the dual notions of possibility and necessity respectively. We augment these two modal constructs with symbolic actions, $(x, e)$, to enable the reasoning on the data carried by external actions. Symbolic actions are pairs consisting of data variables, $x, y \in \text{DVAR}$, and decidable Boolean constraint expressions, $e, f \in \text{BEXP}$. Data variables range over the domain $\mathbb{D}$ of data values, and bind the free occurrences of $x$ in the expression $e$ of the modality and in the continuation formula $\varphi$. The set BEXP, defined over $\mathbb{D}$ and DVAR, consists of the usual Boolean operators $\neg$ and $\land$, together with a set of relational operators that depends on $\mathbb{D}$, which we leave unspecified. For clarity, we omit writing the Boolean constraint expression $e$ in modalities when $e = \text{true}$, and use bold lettering to identify binders in symbolic actions. In the sequel, the standard notions of open and closed expressions, and formula equality up to alpha-conversion are used. A formula is said to be guarded if every fixed point variable $X$ appears within the scope of a modality that is itself in the scope of $X$. For example, $\max X.(\left[ x \right] \text{ff} \land \left[ y \right] X)$ is guarded, as is $\max X.(\langle x \rangle (\left[ y \right] \text{ff} \land X))$, while $\left[ x \right] \max X.(\left[ y \right] \text{ff} \land X)$ is not. Without loss of expressiveness [50], we assume all formulae to be guarded.

The linear-time interpretation of MAXHML$^d$ is given by the denotational semantic function $\llbracket - \rrbracket$ that maps a formula to a set of traces. The function $\llbracket - \rrbracket$ uses valuations, $\sigma : \text{LVAR} \rightarrow 2^{\text{Act}^\omega}$, to define the semantics inductively on the structure of formulae. The value $\sigma(X)$ is the set of traces that are assumed to satisfy $X$. In $\llbracket - \rrbracket$, modal formulae are interpreted w.r.t. symbolic actions. A symbolic action $(x, e)$ describes a set of external system actions. An action $\alpha$ is in this set when the data value it carries satisfies the Boolean constraint expression $e$ that is instantiated with the applied substitution $[\alpha/x]$, i.e., $e[\alpha/x] \downarrow \text{true}$ (see Fig. 2). The possibility formula $(x, e)\varphi$ denotes all the traces $\alpha u$ where $\alpha$ is in the action set $(x, e)$ and $u$ satisfies the continuation $\varphi[\alpha/x]$. Dually, $[x, e]\varphi$ denotes all the traces $\alpha u$ that, if prefixed by some $\alpha$ from the action set $(x, e)$,
then satisfies $\varphi[\alpha/x]$. The set of traces satisfying the greatest fixed point formula $\max X.(\varphi)$ is the union of all the post-fixed point solutions, $S \subseteq \text{Act}^\omega$, of the function induced by the formula $\varphi$. Since the interpretation of closed formulae does not depend on the environment $\sigma$, we may use $\llbracket \varphi \rrbracket$ in lieu of $\llbracket \varphi, \sigma \rrbracket$. A trace $t$ satisfies (the closed) formula $\varphi$ when $t \in \llbracket \varphi \rrbracket$, and violates $\varphi$ when $t \notin \llbracket \varphi \rrbracket$.

To facilitate our exposition in this section and Sect. 3, we let $\mathbb{D} = \mathbb{Z}$, and fix the set of operators used in BExp to $\neg$, $\wedge$ and $=$. Sect. 4 considers the general case where the data carried by external actions can consist of composite data types. Henceforth, the terms action and event are used synonymously.

Fig. 3. Token server that issues integer identifier tokens to client programs

2.1 Trace Properties

Consider the model of a reactive token server, $p_1$ in Fig. 3, that issues client programs with identifier tokens that they use as an alias to write logs to a remote logging service. Clients request an identifier by sending the command 0, which the server then fulfills by replying with a new token, $n \in \mathbb{N}$. Since the server is itself a program that also uses the remote logging service, it is launched with its (reserved) identifier token 1. Figure 3 shows that from its initial state $p_1$, the token server either: (i) starts up with the token 1 and transitions to $p_3$, where it waits for incoming client requests, or, (ii) fails to start and transitions with a status of $-1$ to the sink $p_2$, thereafter exhibiting undefined behaviour. There are a number of properties we want executions of this token server to observe.

Example 1. One rudimentary property the current execution of server $p_1$ should uphold is that ‘no failure occurs at start up’. This safety requirement is expressed in terms of the MAXHML$^d$ formula:

$$[x, x = -1] \text{ff} \quad (\varphi_1)$$

The symbolic action $(x, x = -1)$ defines the singleton set $\{-1\} \subset \mathbb{Z}$ of external system actions. Necessity modal formulae $[x, e] \varphi$ state that, for any trace prefix $\alpha$ in the set defined by $(x, e)$, the trace continuation $u$ must then satisfy $\varphi$. However, no trace satisfies ff. This means that, in order for server traces not to violate formula $\varphi_1$, they must start with actions $\alpha \notin \{-1\}$. The set of traces $1.(0.\mathbb{N})^\omega$ exhibited by $p_1$ satisfies this property, whereas $-1.\mathbb{Z}^\omega$ does not. ■
Example 2. Further to the stipulation of Example 1, we require that ‘the server is initialised with the identifier token 1’, expressed as:

\[ [x, x = -1] \text{ff} \land (x, x = 1) \text{tt} \quad (\varphi_2) \]

The conjunct \([x, x = -1] \text{ff}\) guards against traces of \(p_1\) exhibiting failure when loading; \((x, x = 1) \text{tt}\) asserts that the trace exhibits 1 at start up, indicating a successful initialisation of the server. Formula \(\varphi_2\) is satisfied exactly by server traces of the form \(1. \omega\). Note that the binders \(x\) in \([x, x = -1]\) and \((x, x = 1)\) of \(\varphi_2\) bind the variables \(x\) in different scopes.

The symbolic actions of Examples 1 and 2 define sets of external actions w.r.t. literal values (e.g., \(-1, 1\)). More generally, action sets can be defined using constraint expressions that refer to other data variables within the same scope.

Example 3. Amongst the executions satisfying \(\varphi_2\) are those where the server accidentally returns its identifier token 1 in reply to client requests. We therefore demand that ‘the server private token 1 is not leaked in client replies’. Formula \(\varphi_3\) expresses this recursive property in a general way (note that Boolean constraint expressions \(e = \text{true}\) are elided):

\[ [x] \text{max } X.(([y]([z, x = z] \text{ff} \land [z, x \neq z] X)) \quad (\varphi_3) \]

The symbolic action \((x, \text{true})\) in the first necessity defines the set of actions \(Z\). Its binder, \(x\), binds the variable \(x\) in \(\text{max } X. ([y]([z, x = z] \text{ff} \land [z, x \neq z] X))\). For some initial server action \(\alpha \in Z\), applying the substitution \([\alpha/x]\) to this continuation and unfolding the recursion variable once, gives the residual formula:

\[ [y] ([z, \alpha = z] \text{ff} \land [z, \alpha \neq z] \text{max } X. ([y]([z, \alpha = z] \text{ff} \land [z, \alpha \neq z] X))) \quad (\varphi_3') \]

The necessity \([y]\) maps \(y\) to the second server action \(\beta\) in the trace, i.e., \([\beta/y]\). Applying \([\beta/y]\) to \([z, \alpha = z] \text{ff}\) and \([z, \alpha \neq z] \text{max } X. ([y]([z, \alpha = z] \text{ff} \land [z, \alpha \neq z] X))\) leaves both sub-formulae unchanged, since \(y\) binds no variables. For the third server action \(\gamma\), the modalities \([z, \alpha = z]\) and \([z, \alpha \neq z]\) map \(z\) to \(\gamma\). Formula \(\varphi_3\) is violated, \(\text{ff}\), when the constraint \(\alpha = z'\) holds, i.e., \(\alpha \in \{n \in Z \mid \alpha = \gamma\}\). Crucially, a fresh scope for data variables is created upon each unfolding of \(X\), such that \(y\) and \(z\) can be mapped to new values. By contrast, the value in \(x\) is substituted for once in \(\varphi_3'\) and remains fixed when \(X\) is unfolded. Concretely, formula \(\varphi_3\) compares actions at every odd position in the trace against the one at the head. Interpreting \(\varphi_3\) over traces that the token sever exhibits on successful initialisation ensures that in particular, \(1. (0. \{n \in N \mid n \neq 1\})^* . (0.1). \omega\) are violating. We remark that this property is not expressible in LTL.
3 Monitor Synthesis

The logic \text{maxHML}^d is interpreted over infinite traces that represent complete system runs (refer to Sect. 2). In (online) RV where the SuS is reactive, obtaining complete runs is typically not possible [17,52], since the current trace corresponds to a prefix that is incrementally extended as the system execution unfolds. The notions of good and bad prefixes for monitorable properties provide sufficient evidence to determine acceptance or rejection. Informally, a good (resp. bad) prefix is a finite trace for which every extension satisfies (resp. violates) a property \( \varphi \) [10,24]. Monitors capture this principle through irrevocable verdicts that, once reached, cannot be retracted when observing future events.

Monitors may be viewed as processes via the syntax given in Fig. 4. This syntax differs from its regular counterpart of [3,4] in that it augments the prefixing construct with symbolic actions, \((x,e)\). Besides the prefixing, external choice, and recursion constructs of CCS [54], the syntax of Fig. 4 includes disjunctive, \(\oplus\), and conjunctive, \(\otimes\), parallel composition. We use the symbol \(\circ\) to refer to both \(\oplus\) and \(\otimes\) when needed. Monitor verdict states, \(v \in \text{VRD}\), are expressed as yes and no, respectively denoting acceptance and rejection.

**Monitor Syntax**

\[
m, n \in \text{Mon} ::= v \mid (x,e).m \mid m+n \mid m \oplus n \mid m \otimes n \mid \text{rec}X.m \mid X
\]

\[
v \in \text{VRD} ::= \text{yes} \mid \text{no}
\]

**Monitor Small-Step Semantics**

\[
\begin{align*}
\text{MVRD} & \quad v \xrightarrow{\alpha} v \\
\text{MACT} & \quad e[x/\alpha] \Downarrow \text{true} \\
\text{MCHSL} & \quad m \xrightarrow{\alpha} m' \quad m+n \xrightarrow{\alpha} m'
\end{align*}
\]

\[
\begin{align*}
\text{MTauL} & \quad m \circ n \xrightarrow{\alpha} m' \circ n \\
\text{MPAR} & \quad m \xrightarrow{\alpha} m' \quad n \xrightarrow{\alpha} n' \\
\text{MDisYL} & \quad \text{yes} \circ m \xrightarrow{\alpha} \text{yes}
\end{align*}
\]

\[
\begin{align*}
\text{MDisNL} & \quad \text{no} \circ m \xrightarrow{\alpha} m \\
\text{MConYL} & \quad \text{yes} \otimes m \xrightarrow{\alpha} m \\
\text{MConNL} & \quad \text{no} \otimes m \xrightarrow{\alpha} \text{no}
\end{align*}
\]

\[
\text{MRec} \quad \text{rec}X.m \xrightarrow{\alpha} m[\text{rec}X.m/x]
\]

**Monitor Synthesis**

\[
\begin{align*}
\langle \text{ttt} \rangle & = \text{yes} & \langle \text{ff} \rangle & = \text{no} \\
\langle (x,e) \varphi \rangle & = (x,e).\langle \varphi \rangle + (x,-e) \cdot \text{no} & \langle [x,e] \varphi \rangle & = (x,e).\langle \varphi \rangle + (x,-e) \cdot \text{yes} \\
\langle \varphi \lor \psi \rangle & = \langle \varphi \rangle \oplus \langle \psi \rangle & \langle \varphi \land \psi \rangle & = \langle \varphi \rangle \otimes \langle \psi \rangle \\
\langle \text{max}X.\langle \varphi \rangle \rangle & = \text{rec}X.\langle \varphi \rangle & \langle X \rangle & = X
\end{align*}
\]

**Fig. 4.** Syntax, synthesis, and small-step semantics for parallel monitors
A Monitoring Tool for Linear-Time $\mu$HML

Figure 4 outlines the behaviour of monitors, where the transitions rules mREC, mCHSL, and its symmetric case mCHSR (omitted), are standard. Rule MACT describes the analysis monitors perform, where the binder $x$ in the symbolic action ($x,e$) is mapped to an external system action $\alpha$, yielding the substitution $[\alpha/x]$ that is applied to the Boolean constraint expression $e$. The monitor $(x,e).m$ analyses $\alpha$ only if the instantiated constraint $e[\alpha/x]$ is satisfied, whereupon $\alpha$ is substituted for the free variable $x$ in the body $m$. Verdict irrevocability is modelled by MVRD, where once in a verdict state $v$, any action can be analysed by monitors without altering $v$. Rule MPAR enables parallel sub-monitors to transition in lock-step when they analyse the same action $\alpha$. The rest of the rules (omitting the obvious symmetric cases) cater for the internal reconfiguration of monitors. For instance, rules MDISYL and MDISNL state that in disjunctive parallelism, yes supersedes the verdicts of other monitors, whilst no does not affect the verdicts of other monitors; MCONYL and MCONN_L express the dual case for parallel conjunctions. Finally, MTAUL and its symmetric analogue permit sub-monitors to execute internal reconfigurations independently.

Our adaptation ($\neg\cdot$) of the compositional synthesis procedure for regular monitors [3,4] is also given in Fig. 4. It generates monitors for $\phi \in \maxhml$, following the inductive structure of formulae. The translation for truth and falsehood, and the greatest fixed point and recursion variable constructs is direct; disjunction and conjunction are transformed to their parallel counterparts. Modal constructs are mapped to deterministic external choices, where the left summand handles the case where a system action $\alpha$ is in the set described by the symbolic action $(x,e)$, and the right summand, the case where $\alpha$ is not in this set. This embodies the duality of possibility and necessity: when $\alpha$ is not in the action set $(x,e)$, the formula $(x,e)\phi$ is violated, whereas $(x,e)\phi$ is trivially satisfied.

Example 4. The monitor $m_2$ synthesised from formula $\varphi_2$ is:

\[
(|\varphi_2| = ((|x,x = -1| \text{ff} \land (x,x = 1)\text{tt}) = (|(x,x = -1)\text{ff}| \otimes (x,x = 1)\text{tt}) \quad (m_2)
\]

\[
= ((x,x = -1)\text{no} + (x,x \neq -1)\text{yes}) \otimes ((x,x = 1)\text{yes} + (x,x \neq 1)\text{no})
\]

When analysing the server traces $1..2^\omega$, monitor $m_2$ reduces to $\text{no} \otimes \text{no}$ via the rule MPAR. Its premises are obtained by applying the MCHSL and MACT to the left sub-monitor, and MCHSR and MACT to the right sub-monitor, giving:

\[
(x,x = -1)\text{no} + (x,x \neq -1)\text{yes} \overset{1}{\rightarrow} \text{no} \quad \text{and} \quad (x,x = 1)\text{yes} + (x,x \neq 1)\text{no} \overset{1}{\rightarrow} \text{no}
\]

Monitor $\text{no} \otimes \text{no}$ afterwards transitions internally, $\text{no} \otimes \text{no} \overset{\tau}{\rightarrow} \text{no}$, via either rule MCONNL or MCONN_R. Analogously, $m_2$ reaches the verdict yes when analysing the server traces $1..\mathbb{N}^\omega$. Recall that when in a verdict state, the monitor can always analyse future events by virtue of MVRD, flagging the same outcome. The behaviour of monitor $m_2$ corresponds to the property that $\varphi_2$ describes (refer to [3,4] for details).
Example 5. Consider the recursive monitor $m_3$ synthesised from formula $\varphi_3$:

$$(x) . \text{rec } X . (y) . \left( ((z, x = z) . \text{no} + (z, x \neq z) . \text{yes}) \otimes \left( ((z, x \neq z) . X + (z, x = z) . \text{yes}) \right) \right)$$

$(m_3)$

For the server traces $1.0.2.0.1.(0.N)\omega$, $m_3$ instantiates $x$ to the value $1$ at the head, and applies the substitution $[1/x]$ to the residual monitor, giving:

$$(x) . \text{rec } X . (y) . \left( ((z, 1 = z) . \text{no} + (z, 1 \neq z) . \text{yes}) \otimes \left( ((z, 1 \neq z) . X + (z, 1 = z) . \text{yes}) \right) \right)$$

$(m'_3)$

Hereafter, $m'_3$ unfolds continually, ensuring that no event carries the value $1$ observed at the head of the trace. At every even position, $y$ is instantiated with $0$, whereas the binders $z$ in each of the sub-monitors composed in parallel compare the value carried by events occurring at odd trace positions against $1$. Monitor $m'_3$ reaches the verdict no via these reductions:

$$m'_3 \xrightarrow{\tau} (y) . \left( ((z, 1 = z) . \text{no} + (z, 1 \neq z) . \text{yes}) \otimes \left( ((z, 1 \neq z) . m'_3 + (z, 1 = z) . \text{yes}) \right) \right)$$

$(m''_3)$

$$m''_3 \xrightarrow{0} ((z, 1 = z) . \text{no} + (z, 1 \neq z) . \text{yes}) \otimes ((z, 1 \neq z) . m''_3 + (z, 1 = z) . \text{yes})$$

$(m'''_3)$

$$m'''_3 \xrightarrow{2} \text{yes} \otimes m'_3 \xrightarrow{\tau} m'_3 \xrightarrow{\tau} m''_3 \xrightarrow{0} m''_3 \xrightarrow{1} \text{no} \otimes \text{yes} \xrightarrow{\tau} \text{no} \xrightarrow{n} \text{no} \xrightarrow{n} \ldots$$

For the non-rejecting server traces $1.(0, \{n \in N \mid n \neq 1\})\omega$, monitor $m'_3$ visits the state yes $\otimes m'_3$ indefinitely, where $m'_3$ supersedes the uninfluential verdict yes following the rule $\text{MCONY}_L$. ■

4 Implementation

We implement our tool in Erlang [11,27], a general-purpose programming language that adopts the actor model of concurrency [8,44]. In this model, processes communicate exclusively by addressing asynchronous messages to one another via their uniquely-assigned Process ID (PID). Besides sending and receiving messages, processes can also fork other processes. This concurrency paradigm is tailored to reactive systems, making it ideal for our study.

4.1 Refining the Model

We refine the logic of Sect. 2 and monitor model of Sect. 3 to fit the Erlang use-case, where data can consist of composite types, such as tuples and lists. Accordingly, we generalise our definition of external actions as follows. Let $l \in \mathcal{L}$ be a finite set of action labels, and $d_1, d_2, \ldots$ be data values taken from a set of data domains $\mathcal{D} = \bigcup_{i \in \mathbb{N}} \mathbb{D}_i$ (e.g. integers, PIDs, tuples, etc.). An external action
α is a tuple, \((l, d_1, \ldots, d_n)\), where the first element is the label, and \(d_1, \ldots, d_n\) is the data payload carried by \(α\). We use the notation \(l(d_1, \ldots, d_n)\) to write \(α\).

Patterns, \(p \in \text{PAT}\), are counterparts to external system actions. These are defined as tuples, \((l, x_1, \ldots, x_n)\), where \(x_1, x_2, \ldots\) are data variables ranging over \(D\). Our revised definition of symbolic actions in the modal constructs \(⟨p, e⟩\varphi\) and \([p, e]\varphi\) uses these patterns instead of variables (cf. Sect. 2). The binders \(x_1, \ldots, x_n\) in \(p\) bind the free occurrences of \(x_1, \ldots, x_n\) in the Boolean constraint \(e\), and in the continuation \(\varphi\). We define the function, \(\text{match}(p, α)\), to handle pattern matching. This function returns a substitution, \(π: \text{DVAR} \rightarrow \mathcal{D}\), that maps the variables in \(p\) to the corresponding data values in the payload carried by \(α\), when the shape of the pattern matches that of the action, or \(⊥\) if the match is unsuccessful. Analogous to the symbolic actions of Sect. 2, \((p, e)\) describes a set of actions: an action \(α\) is in this set if (i) the pattern match succeeds, i.e., \(\text{match}(p, α) = π\), and, (ii) the instantiated Boolean constraint expression \(eπ\) holds.

To instantiate our tool to Erlang, we use the action label set \(\mathcal{L} = \{-→, ←, ⋆, !, ?\}\), that captures the lifecycle of, and interaction between processes. The fork action, \(-→\), is exhibited by a process when it creates a child; its dual, \(←\), is exhibited by the child process upon initialisation. An error action, \(⋆\), signals abnormal process behaviour; send and receive, respectively \(!\) and \(?\), denote interaction. Table 1 details the actions related to these labels, and the data payload they carry.

![Diagram of client-server interaction of the Erlang token server implementation](image)

**Fig. 5.** Client-server interaction of the Erlang token server implementation

Our token server of Fig. 3 is readily translatable to Erlang, as shown in Fig. 5. The server starts when its main function, \(lp\), in the Erlang module \(ts\) is invoked, state \(p_1\). From \(p_1\), it transitions to \(p_3\), exhibiting the initialisation event \(←(PID_S, PID_P, ts, lp, 1)\); 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 \(PID_C\) denotes the PID of the client, and \(0\) is the command requesting a new token. From state \(p_4\), the server replies with \(n\), and transitions back to \(p_3\). This client-server interaction results in the server events \(?(PID_S, \{PID_C, 0\})\) and \(!(PID_S, PID_C, n)\). When the server fails at startup, it exhibits abnormal behaviour, shown as the error events \(⋆(PID_S, -1)\) and \(⋆(PID_S, m)\).

**Example 6.** Formula \(\varphi_2\) w.r.t. the Erlang server of Fig. 5 is expressed as follows:

\[⋆(x_1, x_2), x_2 = -1]\, ff \land \langle ←(x_1, x_2, x_3, x_4, x_5), x_5 = 1 \rangle tt \quad (\varphi_4)\]
Table 1. Actions capturing the behaviour exhibited by Erlang processes

<table>
<thead>
<tr>
<th>Action</th>
<th>Action pattern</th>
<th>Variables</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td>(\rightarrow(x_1, x_2, y_1, y_2, y_3))</td>
<td>(x_1)</td>
<td>PID of the parent process forking (x_2)</td>
</tr>
<tr>
<td>initialise</td>
<td>(\leftarrow(x_2, x_1, y_1, y_2, y_3))</td>
<td>(x_2)</td>
<td>PID of the child process forked by (x_1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(y_1, y_2, y_3)</td>
<td>Function signature forked by (x_1)</td>
</tr>
<tr>
<td>error</td>
<td>(\star(x_1, y_1))</td>
<td>(x_1)</td>
<td>PID of the erroneous process</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(y_1)</td>
<td>Error datum, e.g. error reason, etc.</td>
</tr>
<tr>
<td>send</td>
<td>(!(x_1, x_2, y_1))</td>
<td>(x_1)</td>
<td>PID of the process sending the message</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(x_2)</td>
<td>PID of the recipient process</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(y_1)</td>
<td>Message datum, e.g. integer, tuple, etc.</td>
</tr>
<tr>
<td>receive</td>
<td>(?(x_2, y_1))</td>
<td>(x_2)</td>
<td>PID of the recipient process</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(y_1)</td>
<td>Message datum, e.g. integer, tuple, etc.</td>
</tr>
</tbody>
</table>

The patterns in the left and right conjuncts of \(\varphi_4\) match the error and initialisation events. When \(p_1\) exhibits an error at start up, \(\text{match}(\star(x_1, x_2), \star(PID_3, -1))\), yields the substitution \(\pi = [\text{PIDS}/x_1, -1/x_2]\), and the instantiated Boolean constraint \((x_2 = -1)\pi\) holds. For the same event, \(\text{match}(\leftarrow(x_1, x_2, x_3, x_4, x_5), \star(PID_3, -1)) = \bot\) in the right conjunct, leading to a violation of formula \(\varphi_4\). The reverse argument applies for when \(p_1\) loads successfully, where \(\varphi_4\) is satisfied. In \(\varphi_4\), the pattern variables \(x_1\) in \(\star(x_1, x_2)\), and \(x_1, x_2, x_3, x_4, x_5\) in \(\leftarrow(x_1, x_2, x_3, x_4, x_5)\) are redundant.

\[
\begin{align*}
\text{max}_{x_5}\{\max_{x_5}\left(\lfloor\lfloor[!(-, -, z_3), x_5 = z_3]\rfloor\land [!(-, -, z_3), x_5 \neq z_3]X\right)\} & \quad (\varphi_5)
\end{align*}
\]

Formula \(\varphi_5\) restates \(\varphi_3\) with pattern matching. It uses the ‘don’t care’ pattern \(-\), that matches arbitrary values, eliding redundant patterns and variables.

4.2 The Monitor Synthesis

Our synthesis from \textsc{maxHML}^d specifications to executable Erlang monitors follows that of Fig. 4. Figure 6 omits the cases for the falsity, necessity and conjunction constructs, as these are analogous to the ones for \(\text{tt}, \langle p, e\rangle\varphi\) and \(\varphi \lor \psi\). The translation from specifications to monitors is executed in three stages. First, a formula is parsed into its equivalent Abstract Syntax Tree (AST). This is then
\[
\begin{align*}
\langle \text{tt} \rangle &= \text{yes} & \langle \varphi \lor \psi \rangle &= \{ \text{or}, \langle \varphi \rangle, \langle \psi \rangle \} \\
\langle \max X. (\varphi) \rangle &= \{ \text{rec}, \text{fun} X() \to \langle \varphi \rangle \text{ end} \} & \langle X \rangle &= \{ \text{rec}, X \}
\end{align*}
\]

Fig. 6. Translation from \(\text{maxHML}^d\) formulae to Erlang code (excerpt)

passed to the code generator that visits each of its nodes, mapping it to a monitor description as per the rules of Fig. 6. The monitor description is encoded as an Erlang AST to simplify its handling. In the final stage, this AST is processed by the Erlang compiler to emit the monitor source code or a BEAM [27] executable.

In this definition of \(\llbracket - \rrbracket\), \texttt{tt} (resp. \texttt{ff}) is translated to the Erlang atom \texttt{yes} (resp. \texttt{no}) that indicates acceptance (resp. rejection). The remaining cases generate Erlang tuples whose first element, called the tag, is an atom that identifies the kind of monitor. Disjunctions (resp. conjunctions) are translated to the tuple tagged with \texttt{or} (resp. \texttt{and}), combining two sub-monitor descriptions. Greatest fixed point constructs, \(\max X. (\varphi)\), are mapped to \texttt{rec} tuples consisting of named functions, \(\text{fun} X() \to \langle \varphi \rangle \text{ end} \), that can be referenced by \(\langle X \rangle \). Modal constructs are synthesised as a choice with left and right actions. An action tuple, \texttt{act}, combines a predicate function and an associated monitor body that is unfolded when the predicate is \texttt{true}. The predicate function encodes the pattern matching and Boolean constraint evaluation as one operation, using two clauses. Its first clause, \(\text{fun}(p) \text{ when } e \to \text{true}; (\_ \to \text{false end})\), tests the constraint \(e\) w.r.t. the variables in the pattern \(p\) that become dynamically instantiated with the data values carried by an action \(\alpha\) at runtime. The second catch-all clause \((\_ \to \text{false})\) covers the remaining cases, namely when: (i) either the action under analysis fails to match the pattern, or, (ii) the pattern matches \textit{but} the Boolean constraint does \textit{not} hold. For the left action, the predicate clause \(\text{fun}(p) \text{ when } e \to \text{true when the pattern match and guard test succeed, and false otherwise, i.e., (\_ \to \text{false})\). This condition is inverted for the right action, modelling cases (i) and (ii) just described. Our encoding of the aforementioned predicate in terms of Erlang function clauses spares us from implementing the pattern matching and constraint evaluation mechanism. It also enables monitors to support most of the Erlang data types and its full range of Boolean constraint expression syntax [11]. For similar reasons, \(\langle (p,e)\varphi \rangle\) encodes the monitor body as \(\text{fun}(p) \to \langle \varphi \rangle \text{ end} \) to delegate scoping to the Erlang language. This facilitates our synthesis and optimises the memory management of monitors by offloading this aspect onto the language runtime.
def DERIVEACT(α, mon)
match mon do
  case yes ∨ no
  print ’Verdict reached’
  case {act, Pred, m}
  return m(α) # Apply m to event α
  case {chs, m, n}
if HOLDS(α, m) ∧ ¬HOLDS(α, n)
  return DERIVEACT(α, m)
else
  ¬HOLDS(α, m) ∧ HOLDS(α, n)
  return DERIVEACT(α, n)
case {Op, m, n} ∧ Op ∈ {or, and}
m' = DERIVEACT(α, m)
n' = DERIVEACT(α, n)
  return {Op, m', n'}
end def

Expect: Monitor must be in ready state

def ANALYSEACT(α, m)
m' = DERIVEACT(α, m)
return REDUCETAU(m')
end def

def DERIVETAU(mon)
match mon do
  case {or, yes, m} return yes
  case {or, no, m} return m
  case {and, yes, m} return m
  case {and, no, m} return no
  case {rec, m} return m() # Unfold
  case {Op, m, n} ∧ Op ∈ {or, and}
if m' = DERIVETAU(m) ∧ m' ≠ ⊥
  return m'
else
  return DERIVETAU(n)
end if
  case Otherwise return ⊥
end def

def REDUCETAU(m)
if m' = DERIVETAU(m) ∧ m' ≠ ⊥
  return REDUCETAU(m')
else
  return m # No more τ reductions
end if
end def

Alg. 1. Algorithm that reduces monitors following the small-step rules of Fig. 4

4.3 The Monitoring Algorithm

The synthesis procedure of Fig. 4 generates monitors that can runtime check formulae in parallel against the same position in the trace via disjunctive and conjunctive parallel composition. Our tool is however engineered to emulate parallel monitors, rather than forking processes and delegate their execution to the Erlang runtime. While the latter method tends to simplify the synthesis and runtime monitoring, we adopt the former approach for two reasons:

(i) Previous empirical evidence suggests that parallelising via processes may induce high overhead when the RV set-up is considerably scaled [15]. A process-free design may render this overhead more manageable [5,6].

(ii) Emulating parallel monitors requires us to tease apart the synthesised monitor description from its operational semantics. By separating these two aspects, our monitoring algorithm can track the operational rules it applies to reduce the monitor state, and use these to justify how verdicts are reached.

Our monitoring algorithm (Algorithm 1) takes a monitor description m generated by ⟨−⟩, and performs successive reductions by applying m to events from the trace until a verdict is reached. Simultaneously, the algorithm maintains all the possible active states of the monitor as this is evolved from one state to the
next. Algorithm 1 encodes this reduction strategy using a series of case statements (lines 2–16 and 23–35), following the operational semantics of Fig. 4. Each case maps the first part of a rule conclusion to a pattern, enabling the monitoring algorithm to unambiguously match the rule to apply. The body of cases consists of a return statement that corresponds to the outcome dictated by the rule. Rules with premises (e.g. mChsL, mPar, etc.) are reduced recursively by reapplying rules until an axiom is met, whereas axioms (e.g. mVrd, mDisNL, etc.) reduce immediately. For example, the pattern \{chs, m, n\} on line 7 specifies that mChsL and mChsR only apply to monitors of the form \(m + n\). Selecting whether to reduce the left or right sub-monitor by analysing \(\alpha\) is delegated to the function Holds. This instantiates the predicate encoded in act tuples with the data from \(\alpha\) (see Fig. 6), returning the result of the predicate test. When the condition \(\text{Holds}(\alpha, m) \land \neg \text{Holds}(\alpha, n)\) is true, \(m + n\) is reduced to \(m\), equivalent to the application of mChsL; the argument for mChsR is symmetric.

The function AnalyseAct of Algorithm 1 conducts the runtime analysis. It ensures that once an action is analysed, the monitor is left in a state where it is ready to analyse the next action. We implement this logic by organising the application of the operational rules of Fig. 4 into two functions, DeriveAct and DeriveTau, according to the kind of action used to reduce the monitor. DeriveAct on line 19 reduces the monitor once by applying it to the action under analysis, yielding \(m'\). Subsequently, ReduceTau reapplies the function DeriveTau until all the internal transitions of the monitor are exhausted (lines 38–42). The cases on lines 24–27, corresponding to the axioms mDisYL, mDisNL, mConYL, mConNL, terminate redundant monitor states, and may be seen as a form of garbage collection. Due to space constraints, DeriveTau omits the cases symmetric to those of lines 24–27.

Every single application of DeriveAct and DeriveTau results in a derivation that shows how a monitor evolves from one state to the next according to the operational rules of Fig. 4. The function AnalyseAct keeps a complete history of these derivations internally. Derivations are represented as trees, rooted at the conclusion and terminating at the axiom nodes, that for each step: (i) maintain the monitor state consisting of the substitution \(\pi\), and, (ii) the name of the rule used to derive the step. Maintaining the variable-value mapping in \(\pi\) across derivation steps demands that we track the changes in these variables for every active monitor state. This includes accounting for different binding scopes, variable shadowing, and the creation of fresh scopes when recursion variables, \(X\), are unfolded. Our Erlang implementation of the functions listed in Algorithm 1 in the tool incorporate this described logic. We are aware that storing the complete history ultimately impacts the performance of the runtime analysis. Our tool compromises by offering two operating modes, normal and debug, where in the latter, the full derivation history is stored in memory for the purpose of explainability.
4.4 Monitor Instrumentation

Our tool leverages the existing inlining [34] mechanism implemented in detectEr to instrument the SuS with monitors. While this approach assumes access to the source code of the SuS, it has been shown to induce lower overhead, by contrast to its outline counterpart [5,32,33]. The tool provides the meta keyword with, to identify the SuS components against which maxHML\(^d\) specifications are runtime checked. Readers are referred to [12] for more details.

5 Case Study

We show the usability of our tool by applying it to an off-the-shelf Erlang webserver called Cowboy [45]. Cowboy delegates its socket management to Ranch (a socket acceptor pool for TCP protocols [46]), but forwards incoming HTTP client requests to protocol handlers that are forked dynamically by the webserver to service requests independently. We use our tool to runtime check maxHML\(^d\) specifications describing fragments of the interaction protocol between the Cowboy and Ranch components. Our aim is to: (i) demonstrate the expressiveness of our logic by capturing properties of real-world software, (ii) validate the applicability of our monitoring and instrumentation techniques to third-party code, namely to applications built on top of the Erlang OTP middleware libraries, and, (iii) explore the utility of explainable verdicts for diagnosing software issues.

We redesign the token server of Fig. 5 as a REST web service deployed on Cowboy. The server generates identifier tokens using one of two formats, UUIDs, or short alphanumeric strings. Clients request tokens by issuing a GET request with parameter, type=uuid or type=short, specifying the token format required. The web service offers a standard interface: (i) it returns HTTP 400 when the type parameter is omitted from the request, and, (ii) HTTP 500 when an unsupported type is used. We also simulate intermittent faults in the Cowboy components by injecting process crashes based on a fair Bernoulli trial [55].

For our case study, we consider a selection of properties describing the Cowboy-Ranch interaction protocol. One such property, \(\varphi_{rp}\), concerns Cowboy request processes that service client requests. It states that in its (current) execution, ‘a request process does not issue HTTP responses with code 500, nor does it crash’.

\[
\text{max } X. \left( \left( \neg \{rprc,-,\{\text{tag}, \text{code}, \ldots \}\}, \text{tag} = \text{resp} \land \text{code} = 200 \right) X \land \left( \neg \{rprc,-,\{\text{tag}, \text{code}, \ldots \}\}, \text{tag} = \text{resp} \land \text{code} = 500 \right) \text{ff} \land \left( \{rprc, \text{stat}\}, \text{stat} = \text{crash} \right) \text{ff} \right) \quad (\varphi_{rp})
\]

In \(\varphi_{rp}\), the binders tag and code become instantiated with the atom respdesignating a response message, and the HTTP code of the response returned to requesting clients. Besides ensuring that response messages sent by request processes do not contain the code 500, i.e., \(\text{tag} = \text{resp} \land \text{code} = 500\), formula \(\varphi_{rp}\) also asserts that these processes do not crash, i.e., \(\text{stat} = \text{crash}\). The binder rprc, referring to the request process PID, is included in \(\varphi_{rp}\) for clarity.
While the monitor synthesised from $\varphi_{rp}$ flags the corresponding rejection, the verdict (alone) does not indicate the source of the error. This may suffice for verifying small-scale systems where errors are manually-trackable, but becomes impractical in realistic settings such as this case study. Our algorithm addresses this shortcoming by giving a justification showing how a monitor reaches its verdict.

6 Conclusion

This paper presents the implementation of a RV tool that runtime checks specifications written in a safety-fragment of the linear-time modal $\mu$-calculus, called maxHML, augmented with predicates over data. Our work builds on previous theoretical results for the regular setting \cite{3,4} that give a compositional synthesis procedure which generates monitors from maxHML formulae. We extend the logic, synthesis, and monitor operational semantics of \cite{3,4} to enable the tool to handle events that carry data. We discuss the implementability of this synthesis procedure, and overview the approach our monitoring algorithm takes towards providing justifiable monitoring verdicts. Our tool is validated via a realistic case study that uses an off-the-shelf, third-party Erlang webserver. We show how our augmented logic flexibly expresses properties involving data, and argue for the utility of explainable verdicts for diagnosing software issues.

Related Work. The synthesis procedure in this paper contrasts with another by a line of work that investigates the monitorable safety fragment of the branching-time modal $\mu$-calculus, called sHML \cite{37,38}. The latter synthesis generates monitors with non-deterministic behaviour that, while sufficient for the theoretical results required in op. cit., may lead to missed detections in practice. An early materialisation of \cite{37,38} as the tool detectEr \cite{13,14,26,40} addresses this shortcoming by parallelising monitors using processes, enabling them to reach verdicts along all possible paths. While effective, this approach scales poorly \cite{15}. Ongoing work on detectEr \cite{12} indicates that a process-free approach could lead to more efficient runtime monitoring that scales considerably better \cite{5,6}.

There are other ways to monitoring systems with events that carry data besides the ones cited in Sect. 1 (see e.g., \cite{18,20,22,41–43,61}). One work that shares characteristics with ours is Parametric Trace Slicing (PTS) \cite{29,57}, where the global trace is projected into local sub-traces called slices, based on parametric specifications. These are properties specified in terms of symbolic events whose parameters are instantiated to values from events in the global trace. We use similar means to identify the SuS components to be instrumented, thus filtering out events and obtain trace slices (see Sect. 4.4). PTS is adopted by a number of RV tools that handle data (see e.g., \cite{9,28,31,47,53}), notably MarQ \cite{16,56} for Java, and Elarva \cite{30} for Erlang. Elarva takes a naïve strategy to PTS, relying on a central process to collect trace events that are demultiplexed between monitors to obtain slices. This makes it susceptible to single point of failures,
and does not scale in practice. The design we use is more robust, as it directly instruments components of the SuS, giving us a modicum of fault containment when monitoring independently-executing components that can fail in isolation.

References


34. Falcone, Y., Krsti´c, S., Reger, G., Traytel, D.: A taxonomy for classifying run-
03769-7_14
doi.org/10.1007/978-3-319-67531-2_2
with recursion at runtime. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS,
vol. 9333, pp. 71–86. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-
23820-3_5
38. Francalanza, A., Aceto, L., Ingólfsdóttir, A.: Monitorability for the Hennessy-
Milner logic with recursion. FMSD 51(1), 87–116 (2017)
monitors using a local proof system. J. Log. Algebraic Methods Program. 119,
100636 (2021)
41. Havelund, K., Peled, D.: Runtime verification: from propositional to first-order
42. Havelund, K., Peled, D.: BDDs for representing data in runtime verification.
Springer, Cham (2020). https://doi.org/10.1007/978-3-030-60508-7_6
43. Havelund, K., Reger, G., Thoma, D., Zălinescu, E.: Monitoring events that carry
data. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS,
vol. 10457, pp. 61–102. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-
75632-5_3
44. Hewitt, C., Bishop, P.B., Steiger, R.: A universal modular ACTOR formalism for
47. Jin, D., Meredith, P.O., Lee, C., Rosu, G.: JavaMOP: efficient parametric runtime
48. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cam-
bridge (1999)
49. Kozen, D.: Results on the propositional $\mu$-calculus. In: Nielsen, M., Schmidt, E.M.
https://doi.org/10.1007/BFb0012782
51. Larsen, K.G.: Proof systems for satisfiability in Hennessy-Milner logic with recur-
sion. TCS 72(2&3), 265–288 (1990)
52. Leucker, M., Schallhart, C.: A brief account of runtime verification. JLAP 78(5),
293–303 (2009)
53. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP
Hill (1991)


