A Monitoring Tool for Linear-Time μHML

Luca Aceto[2,3], Antonis Achilleos[2], Duncan Paul Attard[1,2](✉), Léo Exibard[2], Adrian Francalanza[1], and Anna Ingólfsdóttir[2]

[1] University of Malta, Msida, Malta
{duncan.attard.01,afra1}@um.edu.mt
[2] Reykjavik University, Reykjavik, Iceland
{luca,antonios,duncanpa17,leoe,annai}@ru.is
[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 μ-calculus called maxHML. Our technical development is founded on previous results that give a compositional synthesis procedure for generating monitors from 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) [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
M. H. ter Beek and M. Sirjani (Eds.): COORDINATION 2022, LNCS 13271, pp. 200–219, 2022.
https://doi.org/10.1007/978-3-031-08143-9_12

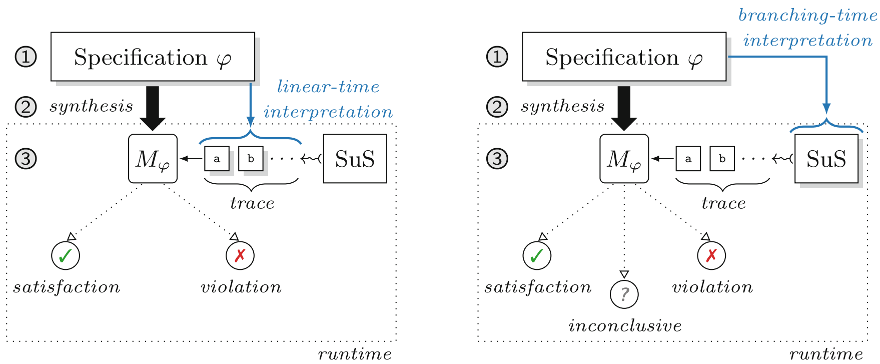
analyse its execution (expressed as a trace of events) *incrementally*, and reach *verdicts* that cannot be retracted when observing future events. Figure 1 depicts this set-up.

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 μ-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 μHML [7,51], a reformulation of the μ-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* syntactic fragments of μHML, 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 identified 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 MAXHML of μHML 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 μHML fragments in a *branching-time* setting [1,2,37,38], where the logic describes properties about the *computation graph* of programs (Fig. 1b). Much of this body



(a) Property φ of the SuS execution *trace* (b) Property φ of the SuS execution *graph*

Fig. 1. RV of linear- and branching-time property φ by analysing the SuS trace

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 $\alpha, \beta \in \textsc{Act}$, together with a distinguished internal action $\tau \notin \textsc{Act}$ 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 $t, u \in \textsc{Act}^\omega$ to represent infinite traces, and use αt to denote an infinite trace that starts with α 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,

$\textsc{maxHML}^d$ Syntax

$$\varphi,\psi \in \textsc{maxHML}^d ::= \mathsf{tt} \mid \mathsf{ff} \mid \langle \boldsymbol{x}, e\rangle\varphi \mid [\boldsymbol{x}, e]\varphi \mid \varphi\vee\psi \mid \varphi\wedge\psi \mid \mathsf{max}\,X.(\varphi) \mid X$$

$\textsc{maxHML}^d$ Semantics

$$[\![\mathsf{tt},\sigma]\!] \triangleq \textsc{Act}^\omega \qquad\qquad [\![\mathsf{ff},\sigma]\!] \triangleq \emptyset$$

$$[\![\langle \boldsymbol{x}, e\rangle\varphi,\sigma]\!] \triangleq \left\{t \mid (\exists u.\,\exists\alpha.\, t=\alpha u \text{ and } e[^\alpha/_x] \Downarrow \mathsf{true} \text{ and } u \in [\![\varphi[^\alpha/_x],\sigma]\!])\right\}$$

$$[\![[\boldsymbol{x}, e]\varphi,\sigma]\!] \triangleq \left\{t \mid (\forall u.\,\forall\alpha.\, (t=\alpha u \text{ and } e[^\alpha/_x] \Downarrow \mathsf{true}) \text{ implies } u \in [\![\varphi[^\alpha/_x],\sigma]\!])\right\}$$

$$[\![\varphi\vee\psi,\sigma]\!] \triangleq [\![\varphi,\sigma]\!] \cup [\![\psi,\sigma]\!] \qquad\qquad [\![\varphi\wedge\psi,\sigma]\!] \triangleq [\![\varphi,\sigma]\!] \cap [\![\psi,\sigma]\!]$$

$$[\![\mathsf{max}\,X.(\varphi),\sigma]\!] \triangleq \bigcup\left\{S \mid S \subseteq [\![\varphi,\sigma[X\mapsto S]]\!]\right\} \qquad\qquad [\![X,\sigma]\!] \triangleq \sigma(X)$$

Fig. 2. Syntax and linear-time semantics for the logic $\textsc{maxHML}^d$

$X, Y \in \textsc{LVar}$. 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 occurrences of X in φ. The existential and universal modalities $\langle \boldsymbol{x}, e\rangle\varphi$ and $[\boldsymbol{x}, e]\varphi$ express the dual notions of *possibility* and *necessity* respectively. We augment these two modal constructs with *symbolic actions*, $(\boldsymbol{x}, e)$, to enable the reasoning on the data carried by external actions. Symbolic actions are pairs consisting of data variables, $x, y \in \textsc{DVar}$, and *decidable* Boolean constraint expressions, $e, f \in \textsc{BExp}$. 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 φ. The set $\textsc{BExp}$, defined over $\mathbb{D}$ and $\textsc{DVar}$, consists of the usual Boolean operators $\neg$ and $\wedge$, 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 $e = \mathsf{true}$, 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, $\mathsf{max}\,X.([\boldsymbol{x}]\mathsf{ff} \wedge [\boldsymbol{y}]X)$ is guarded, as is $\mathsf{max}\,X.([\boldsymbol{x}]([\boldsymbol{y}]\mathsf{ff} \wedge X))$, while $[\boldsymbol{x}]\mathsf{max}\,X.([\boldsymbol{y}]\mathsf{ff} \wedge X)$ is not. Without loss of expressiveness [50], we assume all formulae to be guarded.

The linear-time interpretation of $\textsc{maxHML}^d$ is given by the denotational semantic function $[\![-]\!]$ that maps a formula to a *set* of traces. The function $[\![-]\!]$ uses valuations, $\sigma : \textsc{LVar} \to 2^{\textsc{Act}^\omega}$, 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 $[\![-]\!]$, modal formulae are interpreted w.r.t. symbolic actions. A symbolic action $(\boldsymbol{x}, e)$ describes a *set* of external system actions. An action α 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.*, $e[^\alpha/_x] \Downarrow \mathsf{true}$ (see Fig. 2). The possibility formula $\langle \boldsymbol{x}, e\rangle\varphi$ denotes all the traces αu where α is in the action set $(\boldsymbol{x}, e)$ *and* u satisfies the continuation $\varphi[^\alpha/_x]$. Dually, $[\boldsymbol{x}, e]\varphi$ denotes all the traces αu that, *if* prefixed by some α from the action set $(\boldsymbol{x}, e)$,

u *then* satisfies $\varphi[^{\alpha}/_{x}]$. The set of traces satisfying the greatest fixed point formula $\mathsf{max}\, X.(\varphi)$ is the union of all the post-fixed point solutions, $S \subseteq \text{ACT}^{\omega}$, of the function induced by the formula φ. Since the interpretation of *closed* formulae does not depend on the environment σ, we may use $[\![\varphi]\!]$ in lieu of $[\![\varphi, \sigma]\!]$. A trace t satisfies (the closed) formula φ when $t \in [\![\varphi]\!]$, and violates φ when $t \notin [\![\varphi]\!]$.

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.

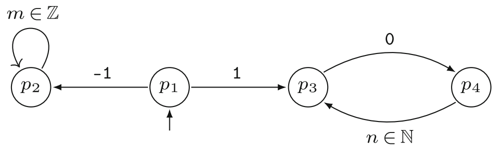


Fig. 3. Token server that issues integer identifier tokens to client programs

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 fulfils 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:

$$[\boldsymbol{x}, x = -1]\mathsf{ff} \qquad (\varphi_1)$$

The symbolic action $(\boldsymbol{x}, x = -1)$ defines the singleton set $\{-1\} \subset \mathbb{Z}$ of external system actions. Necessity modal formulae $[\boldsymbol{x}, e]\varphi$ state that, for *any* trace prefix α in the set defined by $(\boldsymbol{x}, e)$, the trace continuation u must then satisfy φ. However, *no* trace satisfies ff. This means that, in order for server traces *not* to violate formula φ_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:

$$[\boldsymbol{x}, x = -1]\mathsf{ff} \wedge \langle \boldsymbol{x}, x = 1\rangle \mathsf{tt} \qquad (\varphi_2)$$

The conjunct $[\boldsymbol{x}, x = -1]\mathsf{ff}$ guards against traces of p_1 exhibiting failure when loading; $\langle \boldsymbol{x}, x = 1\rangle\mathsf{tt}$ asserts that the trace exhibits 1 at start up, indicating a successful initialisation of the server. Formula φ_2 is satisfied exactly by server traces of the form $1 \,.\, \mathbb{N}^\omega$. Note that the binders $\boldsymbol{x}$ in $[\boldsymbol{x}, x = -1]$ and $\langle \boldsymbol{x}, x = 1\rangle$ of φ_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 φ_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 φ_3 expresses this recursive property in a general way (note that Boolean constraint expressions $e = \mathsf{true}$ are elided):

$$[\boldsymbol{x}]\mathsf{max}\, X.\big([\boldsymbol{y}]([\boldsymbol{z}, x = z]\mathsf{ff} \wedge [\boldsymbol{z}, x \neq z]X)\big) \qquad (\varphi_3)$$

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

$$[\boldsymbol{y}]\Big([\boldsymbol{z}, \alpha = z]\mathsf{ff} \wedge [\boldsymbol{z}, \alpha \neq z]\mathsf{max}\, X.\big([\boldsymbol{y}]([\boldsymbol{z}, \alpha = z]\mathsf{ff} \wedge [\boldsymbol{z}, \alpha \neq z]X)\big)\Big) \qquad ({\varphi_3}')$$

The necessity $[\boldsymbol{y}]$ maps $\boldsymbol{y}$ to the second server action β in the trace, *i.e.*, $[{}^{\beta}/_{y}]$. Applying $[{}^{\beta}/_{y}]$ to $[\boldsymbol{z}, \alpha = z]\mathsf{ff}$ and $[\boldsymbol{z}, \alpha \neq z]\mathsf{max} X\big([\boldsymbol{y}]([\boldsymbol{z}, \alpha = z]\mathsf{ff} \wedge [\boldsymbol{z}, \alpha \neq z]X)\big)$ leaves both sub-formulae unchanged, since $\boldsymbol{y}$ binds no variables. For the third server action γ, the modalities $[\boldsymbol{z}, \alpha = z]$ and $[\boldsymbol{z}, \alpha \neq z]$ map $\boldsymbol{z}$ to γ. Formula φ_3 is violated, ff, when the constraint $\alpha = z[{}^{\gamma}/_{z}]$ holds, *i.e.*, $\alpha \in \{n \in \mathbb{Z} \mid \alpha = \gamma\}$.

Crucially, a *fresh* scope for data variables is created upon each unfolding of X, such that $\boldsymbol{y}$ and $\boldsymbol{z}$ can be mapped to new values. By contrast, the value in $\boldsymbol{x}$ is substituted for *once* in ${\varphi_3}'$ and remains fixed when X is unfolded. Concretely, formula φ_3 compares actions at *every* odd position in the trace against the one at the head. Interpreting φ_3 over traces that the token sever exhibits on *successful* initialisation ensures that in particular, $1 \,.\, (0 \,.\, \{n \in \mathbb{N} \mid n \neq 1\})^* \,.\, (0 \,.\, 1) \,.\, \mathbb{N}^\omega$ are violating. We remark that this property is *not* expressible in LTL. ■

3 Monitor Synthesis

The logic $\textsc{maxHML}^d$ 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 φ [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, $(\boldsymbol{x}, e)$. 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 $\odot$ to refer to both $\oplus$ and $\otimes$ when needed. Monitor verdict states, $v \in \textsc{Vrd}$, are expressed as yes and no, respectively denoting *acceptance* and *rejection*.

Monitor Syntax

$$m,n \in \textsc{Mon} ::= v \quad | \quad (\boldsymbol{x},e).m \quad | \quad m+n \quad | \quad m \oplus n \quad | \quad m \otimes n \quad | \quad \mathsf{rec}\, X.m \quad | \quad X$$
$$v \in \textsc{Vrd} ::= \mathsf{yes} \quad | \quad \mathsf{no}$$

Monitor Small-Step Semantics

$$\textsc{mVrd}\ \frac{}{v \xrightarrow{\alpha} v} \qquad \textsc{mAct}\ \frac{e[^{\alpha}/_{x}] \Downarrow \mathsf{true}}{(\boldsymbol{x},e).m \xrightarrow{\alpha} m[^{\alpha}/_{x}]} \qquad \textsc{mChs}_{\mathrm{L}}\ \frac{m \xrightarrow{\alpha} m'}{m+n \xrightarrow{\alpha} m'}$$

$$\textsc{mTau}_{\mathrm{L}}\ \frac{m \xrightarrow{\tau} m'}{m \odot n \xrightarrow{\tau} m' \odot n} \qquad \textsc{mPar}\ \frac{m \xrightarrow{\alpha} m' \qquad n \xrightarrow{\alpha} n'}{m \odot n \xrightarrow{\alpha} m' \odot n'} \qquad \textsc{mDisY}_{\mathrm{L}}\ \frac{}{\mathsf{yes} \oplus m \xrightarrow{\tau} \mathsf{yes}}$$

$$\textsc{mDisN}_{\mathrm{L}}\ \frac{}{\mathsf{no} \oplus m \xrightarrow{\tau} m} \qquad \textsc{mConY}_{\mathrm{L}}\ \frac{}{\mathsf{yes} \otimes m \xrightarrow{\tau} m} \qquad \textsc{mConN}_{\mathrm{L}}\ \frac{}{\mathsf{no} \otimes m \xrightarrow{\tau} \mathsf{no}}$$

$$\textsc{mRec}\ \frac{}{\mathsf{rec}\, X.m \xrightarrow{\tau} m[^{\mathsf{rec}\, X.m}/_{X}]}$$

Monitor Synthesis

$$(\!|\mathsf{tt}|\!) = \mathsf{yes} \qquad\qquad (\!|\mathsf{ff}|\!) = \mathsf{no}$$
$$(\!|\langle \boldsymbol{x},e\rangle \varphi|\!) = (\boldsymbol{x},e).(\!|\varphi|\!) + (\boldsymbol{x},\neg e).\mathsf{no} \qquad\qquad (\!|[\boldsymbol{x},e]\varphi|\!) = (\boldsymbol{x},e).(\!|\varphi|\!) + (\boldsymbol{x},\neg e).\mathsf{yes}$$
$$(\!|\varphi \vee \psi|\!) = (\!|\varphi|\!) \oplus (\!|\psi|\!) \qquad\qquad (\!|\varphi \wedge \psi|\!) = (\!|\varphi|\!) \otimes (\!|\psi|\!)$$
$$(\!|\max X.(\varphi)|\!) = \mathsf{rec}\, X.(\!|\varphi|\!) \qquad\qquad (\!|X|\!) = X$$

Fig. 4. Syntax, synthesis, and small-step semantics for parallel monitors

Figure 4 outlines the behaviour of monitors, where the transitions rules mREC, mCHS$_\mathrm{L}$, and its symmetric case mCHS$_\mathrm{R}$ (omitted), are standard. Rule mACT describes the analysis monitors perform, where the binder $\boldsymbol{x}$ in the symbolic action $(\boldsymbol{x}, e)$ is mapped to an external system action α, yielding the substitution $[^{\alpha}/_{x}]$ that is applied to the Boolean constraint expression e. The monitor $(\boldsymbol{x}, e).m$ analyses α *only if* the instantiated constraint $e[^{\alpha}/_{x}]$ is satisfied, whereupon α 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 α. The rest of the rules (omitting the obvious symmetric cases) cater for the internal reconfiguration of monitors. For instance, rules mDISY$_\mathrm{L}$ and mDISN$_\mathrm{L}$ state that in disjunctive parallelism, yes supersedes the verdicts of other monitors, whilst no does not affect the verdicts of other monitors; mCONY$_\mathrm{L}$ and mCONN$_\mathrm{L}$ express the dual case for parallel conjunctions. Finally, mTAU$_\mathrm{L}$ and its symmetric analogue permit sub-monitors to execute internal reconfigurations independently.

Our adaptation $(\!|-|\!)$ of the compositional synthesis procedure for regular monitors [3,4] is also given in Fig. 4. It generates monitors for $\varphi \in$ MAXHML^d, 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 α is in the set described by the symbolic action $(\boldsymbol{x}, e)$, and the right summand, the case where α is *not* in this set. This embodies the duality of possibility and necessity: when α is not in the action set $(\boldsymbol{x}, e)$, the formula $\langle \boldsymbol{x}, e\rangle\varphi$ is violated, whereas $[\boldsymbol{x}, e]\varphi$ is trivially satisfied.

Example 4. The monitor m_2 synthesised from formula φ_2 is:

$$\begin{aligned}(\!|\varphi_2|\!) &= (\!|[\boldsymbol{x}, x = -1]\mathsf{ff} \wedge \langle \boldsymbol{x}, x = 1\rangle\mathsf{tt}|\!) = (\!|[\boldsymbol{x}, x = -1]\mathsf{ff}|\!) \otimes (\!|\langle \boldsymbol{x}, x = 1\rangle\mathsf{tt}|\!) \qquad (m_2)\\ &= \big((\boldsymbol{x}, x = -1).\mathsf{no} + (\boldsymbol{x}, x \neq -1).\mathsf{yes}\big) \otimes \big((\boldsymbol{x}, x = 1).\mathsf{yes} + (\boldsymbol{x}, x \neq 1).\mathsf{no}\big)\end{aligned}$$

When analysing the server traces $-1 \,.\, \mathbb{Z}^{\omega}$, monitor m_2 reduces to $\mathsf{no} \otimes \mathsf{no}$ via the rule mPAR. Its premises are obtained by applying the mCHS$_\mathrm{L}$ and mACT to the left sub-monitor, and mCHS$_\mathrm{R}$ and mACT to the right sub-monitor, giving:

$$(\boldsymbol{x}, x = -1).\mathsf{no} + (\boldsymbol{x}, x \neq -1).\mathsf{yes} \xrightarrow{-1} \mathsf{no} \qquad \text{and} \qquad (\boldsymbol{x}, x = 1).\mathsf{yes} + (\boldsymbol{x}, x \neq 1).\mathsf{no} \xrightarrow{-1} \mathsf{no}$$

Monitor $\mathsf{no}\otimes\mathsf{no}$ afterwards transitions internally, $\mathsf{no}\otimes\mathsf{no} \xrightarrow{\tau} \mathsf{no}$, via either rule mCONN$_\mathrm{L}$ or mCONN$_\mathrm{R}$. 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 φ_2 describes (refer to [3,4] for details). ■

Example 5. Consider the recursive monitor m_3 synthesised from formula φ_3:

$$(\boldsymbol{x}).\mathsf{rec}\,X.\,(\boldsymbol{y}).\Big(\big((\boldsymbol{z}, x = z).\mathsf{no} + (\boldsymbol{z}, x \neq z).\mathsf{yes}\big) \otimes \big((\boldsymbol{z}, x \neq z).X + (\boldsymbol{z}, x = z).\mathsf{yes}\big)\Big) \tag{m_3}$$

For the server traces $1\,.\,0\,.\,2\,.\,0\,.\,1\,.\,(0\,.\,\mathbb{N})^\omega$, m_3 instantiates $\boldsymbol{x}$ to the value 1 at the head, and applies the substitution $[{}^1\!/\!_x]$ to the residual monitor, giving:

$$\mathsf{rec}\,X.\,(\boldsymbol{y}).\Big(\big((\boldsymbol{z}, 1 = z).\mathsf{no} + (\boldsymbol{z}, 1 \neq z).\mathsf{yes}\big) \otimes \big((\boldsymbol{z}, 1 \neq z).X + (\boldsymbol{z}, 1 = z).\mathsf{yes}\big)\Big) \tag{m_3'}$$

Hereafter, m_3' unfolds continually, ensuring that no event carries the value 1 observed at the head of the trace. At every even position, $\boldsymbol{y}$ is instantiated with 0, whereas the binders $\boldsymbol{z}$ 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:

$$\begin{aligned}
m_3' &\xrightarrow{\tau} (\boldsymbol{y}).\Big(\big((\boldsymbol{z}, 1 = z).\mathsf{no} + (\boldsymbol{z}, 1 \neq z).\mathsf{yes}\big) \otimes \big((\boldsymbol{z}, 1 \neq z).m_3' + (\boldsymbol{z}, 1 = z).\mathsf{yes}\big)\Big) && (m_3'')\\
&\xrightarrow{0} \big((\boldsymbol{z}, 1 = z).\mathsf{no} + (\boldsymbol{z}, 1 \neq z).\mathsf{yes}\big) \otimes \big((\boldsymbol{z}, 1 \neq z).m_3' + (\boldsymbol{z}, 1 = z).\mathsf{yes}\big) && (m_3''')\\
&\xrightarrow{2} \mathsf{yes} \otimes m_3' \xrightarrow{\tau} m_3' \xrightarrow{\tau} m_3'' \xrightarrow{0} m_3''' \xrightarrow{1} \mathsf{no} \otimes \mathsf{yes} \xrightarrow{\tau} \mathsf{no} \xrightarrow{n} \mathsf{no} \xrightarrow{n} \ldots
\end{aligned}$$

For the non-rejecting server traces $1\,.\,(0\,.\,\{n \in \mathbb{N} \mid n \neq 1\})^\omega$, monitor m_3' visits the state $\mathsf{yes} \otimes m_3'$ indefinitely, where m_3' supersedes the uninfluential verdict yes following the rule mConY$_\mathrm{L}$. ■

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, $(l, \boldsymbol{x_1}, \ldots, \boldsymbol{x_n})$, where $x_1, x_2, \ldots$ are data variables ranging over $\mathcal{D}$. Our revised definition of symbolic actions in the modal constructs $\langle p, e\rangle\varphi$ and $[p, e]\varphi$ uses these patterns instead of variables (*cf.* Sect. 2). The binders $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ in p bind the free occurrences of $x_1, \ldots, x_n$ in the Boolean constraint e, and in the continuation φ. We define the function, $\mathsf{match}(p,\alpha)$, to handle *pattern matching*. This function returns a substitution, $\pi : \text{DVAR} \rightharpoonup \mathcal{D}$, 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 $\perp$ 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 patten match succeeds, *i.e.,* $\mathsf{match}(p, \alpha) = \pi$, and, *(ii)* the *instantiated* Boolean constraint expression $e\pi$ holds.

To instantiate our tool to Erlang, we use the action label set $\mathcal{L} = \{\rightarrow, \leftarrow, \star, !, ?\}$, that captures the lifecycle of, and interaction between 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 interaction. Table 1 details the actions related to these labels, and the data payload they carry.

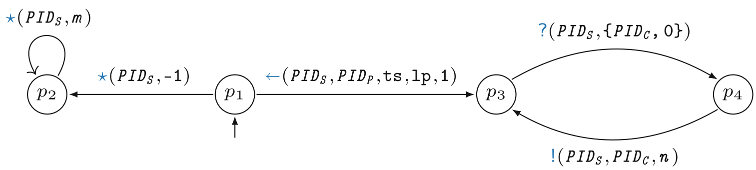


Fig. 5. Client-server interaction of the Erlang token server implementation

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 $\leftarrow(\mathit{PID}_S, \mathit{PID}_P, \texttt{ts}, \texttt{lp}, \texttt{1})$; 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 $\{\mathit{PID}_C, \texttt{0}\}$, 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 $\boldsymbol{n}$, and transitions back to p_3. This client-server interaction results in the server events $?(\mathit{PID}_S, \{\mathit{PID}_C, \texttt{0}\})$ and $!(\mathit{PID}_S, \mathit{PID}_C, \boldsymbol{n})$. When the server fails at startup, it exhibits abnormal behaviour, shown as the error events $\star(\mathit{PID}_S, \texttt{-1})$ and $\star(\mathit{PID}_S, \boldsymbol{m})$.

Example 6. Formula φ_2 w.r.t. the Erlang server of Fig. 5 is expressed as follows:

$$[\star(x_1, x_2), x_2 = \texttt{-1}]\mathsf{ff} \wedge \langle\leftarrow(x_1, x_2, x_3, x_4, x_5), x_5 = \texttt{1}\rangle\mathsf{tt} \qquad (\varphi_4)$$

Table 1. Actions capturing the behaviour exhibited by Erlang processes

Action α	Action pattern p	Variables	Description
fork *initialise*	$\rightarrow(x_1, x_2, y_1, y_2, y_3)$ $\leftarrow(x_2, x_1, y_1, y_2, y_3)$	x_1	PID of the parent process forking x_2
		x_2	PID of the child process forked by x_1
		y_1, y_2, y_3	Function signature forked by x_1
error	$\star(x_1, y_1)$	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.*

The patterns in the left and right conjuncts of φ_4 match the error and initialisation events. When p_1 exhibits an error at start up, $\mathsf{match}\big(\star(x_1, x_2), \star(\mathit{PID}_S, \texttt{-1})\big)$, yields the substitution $\pi = [{}^{\mathit{PID}_S}/_{x_1}, {}^{\texttt{-1}}/_{x_2}]$, and the instantiated Boolean constraint $(x_2 = \texttt{-1})\pi$ holds. For the same event, $\mathsf{match}\big(\leftarrow(x_1, x_2, x_3, x_4, x_5), \star(\mathit{PID}_S, \texttt{-1})\big) = \bot$ in the right conjunct, leading to a violation of formula φ_4. The reverse argument applies for when p_1 loads successfully, where φ_4 is satisfied. In φ_4, the pattern variables x_1 in $\star(x_1, x_2)$, and x_1, x_2, x_3, x_4 in $\leftarrow(x_1, x_2, x_3, x_4, x_5)$ are *redundant*.

$$[\leftarrow(\_,\_,\_,\_,x_5)]\mathsf{max}\, X\Big([\_]\big([!(\_,\_,z_3), x_5 = z_3]\mathsf{ff} \wedge [!(\_,\_,z_3), x_5 \neq z_3]X\big)\Big) \qquad (\varphi_5)$$

Formula φ_5 restates φ_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 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 tt, $\langle p, e\rangle\varphi$ and $\varphi \vee \psi$. 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

$$\llparenthesis \mathsf{tt} \rrparenthesis = \texttt{yes} \qquad\qquad \llparenthesis \varphi \vee \psi \rrparenthesis = \texttt{\{or,}\,\llparenthesis\varphi\rrparenthesis\texttt{,}\,\llparenthesis\psi\rrparenthesis\texttt{\}}$$

$$\llparenthesis \mathsf{max}\, X.(\varphi) \rrparenthesis = \texttt{\{rec, fun X()} \rightarrow \llparenthesis\varphi\rrparenthesis\ \texttt{end\}} \qquad\qquad \llparenthesis X \rrparenthesis = \texttt{\{rec, X\}}$$

$$\llparenthesis \langle p, e\rangle \varphi \rrparenthesis = \begin{cases} \texttt{\{chs,} \\ \quad \left.\begin{array}{l} \texttt{\{act,}\ \overbrace{\texttt{fun(}p\texttt{) when } e \rightarrow \texttt{true; (\_)} \rightarrow \texttt{false end}}^{\textit{predicate}}\texttt{,} \\ \texttt{fun(}p\texttt{)} \rightarrow \llparenthesis\varphi\rrparenthesis\ \texttt{end\},} \end{array}\right\} \textit{left action} \\ \quad \left.\begin{array}{l} \texttt{\{act, fun(}p\texttt{) when } e \rightarrow \texttt{false; (\_)} \rightarrow \texttt{true end,} \\ \underbrace{\texttt{fun(\_)} \rightarrow \texttt{no end}}_{\textit{monitor body}}\texttt{\}} \end{array}\right\} \textit{right action} \\ \texttt{\}} \end{cases}$$

Fig. 6. Translation from 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 $\llparenthesis - \rrparenthesis$, tt (resp. ff) is translated to the Erlang *atom* `yes` (resp. `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 `or` (resp. `and`), combining two sub-monitor descriptions. Greatest fixed point constructs, $\mathsf{max}\, X.(\varphi)$, are mapped to `rec` tuples consisting of *named* functions, $\texttt{fun X()} \rightarrow \llparenthesis\varphi\rrparenthesis\,\texttt{end}$, that can be referenced by $\llparenthesis X \rrparenthesis$. Modal constructs are synthesised as a choice with *left* and *right* actions. An action tuple, `act`, combines a *predicate* function and an associated *monitor body* that is unfolded when the predicate is `true`. The predicate function encodes the pattern matching *and* Boolean constraint evaluation as one operation, using two *clauses*. Its first clause, $\texttt{fun(}p\texttt{) when}\, e$, 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 α at runtime. The second catch-all clause `(_)` covers the remaining cases, namely when: *(i)* either the action under analysis fails to match the pattern, or, *(i)* the pattern matches *but* the Boolean constraint does *not* hold. For the left action, the predicate clause $\texttt{fun(}p\texttt{) when}\, e$ returns `true` when the pattern match and guard test succeed, and `false` otherwise, *i.e.,* `(_)`. 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, $\llparenthesis \langle p, e\rangle \varphi \rrparenthesis$ encodes the monitor body as $\texttt{fun(}p\texttt{)} \rightarrow \llparenthesis\varphi\rrparenthesis\,\texttt{end}$ 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 if ¬Holds(α, m)∧Holds(α, n)
    return DeriveAct(α, n)
   end if
  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 **case**s consists of a **return** statement that corresponds to the outcome dictated by the rule. Rules with premises (*e.g.* mCHS_L, mPAR, *etc.*) are reduced *recursively* by reapplying rules until an axiom is met, whereas axioms (*e.g.* mVRD, mDISN_L, *etc.*) reduce immediately. For example, the pattern $\{\texttt{chs}, m, n\}$ on line 7 specifies that mCHS_L and mCHS_R only apply to monitors of the form $m+n$. Selecting whether to reduce the left or right sub-monitor by analysing α is delegated to the function HOLDS. This instantiates the predicate encoded in `act` tuples with the data from α (see Fig. 6), returning the result of the predicate test. When the condition $\text{HOLDS}(\alpha, m) \wedge \neg\text{HOLDS}(\alpha, n)$ is `true`, $m + n$ is reduced to m, equivalent to the application of mCHS_L; the argument for mCHS_R 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 mDISY_L, mDISN_L, mCONY_L, mCONN_L, 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 π, and, *(ii)* the name of the rule used to derive the step. Maintaining the variable-value mapping in π 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, φ_{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'*.

$$\mathsf{max}\, X. \begin{pmatrix} [!(\boldsymbol{rprc}, \_, \{\boldsymbol{tag}, \boldsymbol{code}, \ldots\}), tag = \mathtt{resp} \wedge code = \mathsf{200}]X \wedge \\ [!(\boldsymbol{rprc}, \_, \{\boldsymbol{tag}, \boldsymbol{code}, \ldots\}), tag = \mathtt{resp} \wedge code = \mathsf{500}]\mathsf{ff} \wedge \\ [\star(\boldsymbol{rprc}, \boldsymbol{stat}), stat = \mathtt{crash}]\mathsf{ff} \end{pmatrix} \qquad (\varphi_{\text{rp}})$$

In φ_{rp}, the binders ***tag*** and ***code*** become instantiated with the atom `resp` designating 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.,* $tag = \mathtt{resp} \wedge code = \mathsf{500}$, formula φ_{rp} also asserts that these processes do not `crash`, *i.e.,* $stat = \mathtt{crash}$. The binder ***rprc***, referring to the request process PID, is included in φ_{rp} for clarity.

While the monitor synthesised from φ_{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 μ-calculus, called MAXHML, augmented with predicates over data. Our work builds on previous theoretical results for the regular setting [3,4] that give a compositional synthesis procedure which generates monitors from MAXHML formulae. We extend the logic, synthesis, and monitor operational semantics of [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 μ-calculus, called SHML [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 [37,38] as the tool detectEr [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 [15]. Ongoing work on detectEr [12] indicates that a process-free approach could lead to more efficient runtime monitoring that scales considerably better [5,6].

There are other ways to monitoring systems with events that carry data besides the ones cited in Sect. 1 (see *e.g.*, [18,20,22,41–43,61]). One work that shares characteristics with ours is Parametric Trace Slicing (PTS) [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.*, [9,28,31,47,53]), notably MarQ [16,56] for Java, and Elarva [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

1. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A.: A framework for parameterized monitorability. In: Baier, C., Dal Lago, U. (eds.) FoSSaCS 2018. LNCS, vol. 10803, pp. 203–220. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89366-2_11
2. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Kjartansson, S.Ö.: Determinizing monitors for HML with recursion. JLAMP **111** (2020)
3. 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)
4. 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)
5. Aceto, L., Attard, D.P., Francalanza, A., Ingólfsdóttir, A.: A Choreographed outline instrumentation algorithm for asynchronous components. CoRR **abs/2104.09433** (2021)
6. Aceto, L., Attard, D.P., Francalanza, A., Ingólfsdóttir, A.: On benchmarking for concurrent runtime verification. In: FASE 2021. LNCS, vol. 12649, pp. 3–23. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-71500-7_1
7. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Reactive Systems: Modelling, Specification and Verification. Cambridge University Press, Cambridge (2007)
8. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. JFP **7**(1), 1–72 (1997)
9. Allan, C., et al.: Adding trace matching with free variables to AspectJ. In: OOPSLA, pp. 345–364. ACM (2005)
10. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. **21**(4), 181–185 (1985)
11. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)
12. Attard, D.P., Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: Better late than never or: verifying asynchronous components at runtime. In: Peters, K., Willemse, T.A.C. (eds.) FORTE 2021. LNCS, vol. 12719, pp. 207–225. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78089-0_14
13. Attard, D.P., Cassar, I., Francalanza, A., Aceto, L., Ingólfsdóttir, A.: Introduction to Runtime Verification. In: Behavioural Types: From Theory to Tools, pp. 49–76. Automation, Control and Robotics, River (2017)
14. Attard, D.P., Francalanza, A.: A monitoring tool for a branching-time logic. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 473–481. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_31
15. Attard, D.P., Francalanza, A.: Trace partitioning and local monitoring for asynchronous components. In: Cimatti, A., Sirjani, M. (eds.) SEFM 2017. LNCS, vol. 10469, pp. 219–235. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66197-1_14

16. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified event automata: towards expressive and efficient runtime monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 68–84. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_9
17. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 1–33. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_1
18. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015)
19. Basin, D.A., Klaedtke, F., Zalinescu, E.: Failure-aware runtime verification of distributed systems. In: FSTTCS. LIPIcs, vol. 45, pp. 590–603. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015)
20. Basin, D., Klaedtke, F., Zălinescu, E.: Runtime verification of temporal properties over out-of-order data streams. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 356–376. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_18
21. Bauer, A., Falcone, Y.: Decentralised LTL monitoring. FMSD **48**(1–2), 46–93 (2016)
22. Bauer, A., Küster, J., Vegliach, G.: The ins and outs of first-order runtime verification. Formal Methods Syst. Des. **46**(3), 286–316 (2015)
23. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. Comput. **20**(3), 651–674 (2010)
24. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14:1–14:64 (2011)
25. Bonakdarpour, B., Fraigniaud, P., Rajsbaum, S., Rosenblueth, D.A., Travers, C.: Decentralized asynchronous crash-resilient runtime verification. In: CONCUR. LIPIcs, vol. 59, pp. 16:1–16:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)
26. 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)
27. Cesarini, F., Thompson, S.: Erlang Programming: A Concurrent Approach to Software Development. O'Reilly Media (2009)
28. Chen, F., Rosu, G.: MOP: an efficient and generic runtime verification framework. In: OOPSLA, pp. 569–588 (2007)
29. Chen, F., Roşu, G.: Parametric trace slicing and monitoring. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 246–261. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_23
30. Colombo, C., Francalanza, A., Gatt, R.: Elarva: a monitoring tool for Erlang. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 370–374. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_29
31. Decker, N., Harder, J., Scheffel, T., Schmitz, M., Thoma, D.: Runtime monitoring with union-find structures. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 868–884. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_54
32. Erlingsson, Ú.: The inlined reference monitor approach to security policy enforcement. Ph.D. thesis, Cornell University (2004)
33. Erlingsson, Ú., Schneider, F.B.: SASI enforcement of security policies: a retrospective. In: NSPW, pp. 87–95 (1999)

34. Falcone, Y., Krstić, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 241–262. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_14
35. Francalanza, A.: A theory of monitors. Inf. Comput. **281**, 104704 (2021)
36. Francalanza, A., et al.: A foundation for runtime monitoring. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 8–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_2
37. Francalanza, A., Aceto, L., Ingolfsdottir, A.: On verifying Hennessy-Milner logic 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)
39. Francalanza, A., Cini, C.: Computer says no: verdict explainability for runtime monitors using a local proof system. J. Log. Algebraic Methods Program. **119**, 100636 (2021)
40. Francalanza, A., Seychell, A.: Synthesising correct concurrent runtime monitors. FMSD **46**(3), 226–261 (2015)
41. Havelund, K., Peled, D.: Runtime verification: from propositional to first-order temporal logic. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 90–112. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_7
42. Havelund, K., Peled, D.: BDDs for representing data in runtime verification. In: Deshmukh, J., Ničković, D. (eds.) RV 2020. LNCS, vol. 12399, pp. 107–128. 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 artificial intelligence. In: IJCAI, pp. 235–245. William Kaufmann (1973)
45. Hoguin, L.: Cowboy (2020). https://ninenines.eu
46. Hoguin, L.: Ranch (2020). https://ninenines.eu
47. Jin, D., Meredith, P.O., Lee, C., Rosu, G.: JavaMOP: efficient parametric runtime monitoring framework. In: ICSE, pp. 1427–1430 (2012)
48. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
49. Kozen, D.: Results on the propositional μ-calculus. In: Nielsen, M., Schmidt, E.M. (eds.) ICALP 1982. LNCS, vol. 140, pp. 348–359. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0012782
50. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. J. ACM **47**(2), 312–360 (2000)
51. Larsen, K.G.: Proof systems for satisfiability in Hennessy-Milner logic with recursion. 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 runtime verification framework. STTT **14**(3), 249–289 (2012)
54. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
55. Papoulis, A.: Probability, Random Variables, and Stochastic Processes. McGraw Hill (1991)

56. Reger, G., Cruz, H.C., Rydeheard, D.: MarQ: monitoring at runtime with QEA. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 596–610. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_55
57. Reger, G., Rydeheard, D.: From first-order temporal logic to parametric trace slicing. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 216–232. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23820-3_14
58. Scheffel, T., Schmitz, M.: Three-valued asynchronous distributed runtime verification. In: MEMOCODE, pp. 52–61 (2014)
59. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: ICSE, pp. 418–427 (2004)
60. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Decentralized runtime analysis of multithreaded applications. In: IPDPS. IEEE (2006)
61. Stolz, V.: Temporal assertions with parametrized propositions. J. Log. Comput. **20**(3), 743–757 (2010)
62. Wolper, P.: Temporal logic can be more expressive. Inf. Control. **56**(1/2), 72–99 (1983)