A Synthesis Tool for Optimal Monitors in a Branching-Time Setting⋆

Antonis Achilleos[1], Léo Exibard[1], Adrian Francalanza[2], Karoliina Lehtinen[3], and Jasmine Xuereb[1,2]

[1] Reykjavik University, Iceland
[2] University of Malta, Malta
[3] CNRS, Aix-Marseille University and University of Toulon, LIS, France

Abstract. Monitorability is a characteristic that delineates between the properties that can be runtime verified by a monitor and those that cannot. Existing notions of monitorability for branching-time specifications are quite restrictive, limiting the set of monitorable properties to a small logical fragment. A recent study has enlarged the set of monitorable branching-time properties by weakening the requirements expected of the monitors effecting the verification: it defines a novel notion of optimal monitor that carries out the maximum number of detections that can be effected for any property, thereby turning a branching-time property into a monitorable one. The study also outlines a method for obtaining a unique optimal monitor from any branching-time property but falls short of providing an automation for this procedure. In this paper, we present a prototype tool that generates monitorable properties for branching-time properties expressed in a variant of the modal μ-calculus, based on this procedure. We also assess the performance of the prototype tool by evaluating its performance against several specifications.

Keywords: Runtime Verification · Monitor Synthesis · Branching-time specifications

1 Introduction

Runtime Verification (RV) is a lightweight verification technique that checks whether a system satisfies some correctness property [11]. This is achieved using monitors [18], which are computational entities that run alongside the system to incrementally observe its behaviour, flagging acceptance or rejection verdicts whenever they detect property satisfactions or violations. When compared to other verification techniques, RV is constrained by the fact that monitors base their analysis on the current system execution being observed. This complicates the verification of correctness properties describing aspects such as infinite executions or alternative execution paths, the evidence of which is hard to represent in a (finite) execution trace.

⋆ This research was supported by the project 'Mode(l)s of Verification and Monitorability (MoVeMnt)' (no. 217987-051) of the Icelandic Research Fund.

These monitorability limits were extensively studied in [19] for branching-time properties expressed in recHML [5], a variant of the modal μ−calculus. There, the authors describe what should be demanded of monitors to adequately runtime verify properties. The first monitor requirement is *soundness*, meaning that whenever a monitor flags an acceptance or a rejection verdict, the system must respectively satisfy or violate the property. Since monitors that may flag both acceptance and rejection verdicts (called *multi-verdict* monitors) are generally inconsistent in the branching-time setting [19], the second monitor requirement is a weaker form of the dual of soundness, termed *partial completeness.* This means that monitors must be able to either reach a verdict for *all* property satisfactions or *all* property violations.

In this study, we focus on monitors that can only flag rejections. The properties for which such monitors can reject all violations are called *rejection-monitorable* (hereafter simply called *monitorable*), and they are precisely the known class of *safety properties* [6] as all of their violations can be detected within a finite sequence of events. This set of monitorable properties is characterised by a maximally expressive syntactic fragment of recHML, termed sHML [19]. In other words, a formula is semantically equivalent to another one in this fragment *if and only if* it can be runtime verified.

Example 1. A system operating a coffee machine produces three events; insert money (m), output coffee (c), and grind more coffee beans (g). Suppose that this system is expected to satisfy the following specification:

"The coffee machine cannot produce event g *(grind) immediately after event* c *(coffee)."* (S_1)

Specification S_1 can only be violated if the system can exhibit event c followed by event g. Such a violation can *always* be witnessed by a finite execution, which, in turn, implies that S_1 is monitorable.

"In all executions, the coffee machine eventually produces event c*, but not before* m *(money)."* (S_2)

Consider specification S_2 above. It can be violated either *(i)* if the system never generates event c or *(ii)* it generates c before m. Suppose the monitor observes the sequence of events mgmgmg $\cdots$. Although event c will *never* be generated, a monitor runtime verifying this property *cannot* flag a violation because despite not having observed c yet, it cannot tell whether it might observe it in the future. Put otherwise, observing a finite execution does *not* provide the monitor with enough information to detect the violation in *(i)*. This implies that S_2 is not monitorable since no monitor can detect *all* violating systems. ■

As indicated by the limited syntax of sHML, restricting RV to monitorable properties severely limits its applicability. Indeed, many properties (such as S_2 above) still fall outside of this scope as *some* violations cannot be determined from finite system executions. Two possible approaches to extend these monitorability limits are weakening the correctness requirements expected of the monitors

effecting the verification or increasing the monitors' observational power. We focus on the first approach, which was studied in [4]. In that work, the authors define a novel notion of *optimal monitors*, which flag all possible violations that can be determined with a finite sequence of events. More concretely, these monitors runtime verify the part of the property that is monitorable, termed the *strongest monitorable consequence.*

Example 2. Although specification S_2 from Example 1 is not monitorable, it turns out that *some* of its violations can be detected. In particular, a violation can be flagged from a finite execution whenever a sequence of events with prefix $\mathsf{g} \cdots \mathsf{gc}$ is observed since c occurs before m. Rather than ruling out S_2 as not monitorable and disregarding its runtime verification altogether, we extract its strongest monitorable consequence, informally described by specification S_3.

"In all executions, the coffee machine never produces c *before* m*."* (S_3)

Clearly, specification S_3 is weaker than S_2, but it gives the best monitorable approximation of the original specification. ■

The work in [4] outlines a two-step procedure to effectively construct an optimal monitor for branching-time recHML properties, expressed in disjunctive form [30]. This procedure first extracts the strongest monitorable consequence, which is formulated in sHML, then synthesises a sound and complete monitor to effect the runtime verification. However, the work in [4] falls short of providing an automation for this procedure.

In this study, we investigate whether the procedure in [4] is amenable to mechanisation. To date, there are several automated monitor synthesis procedures that generate sound and complete monitors: one such tool is detectEr [7]. However, like all known synthesis procedures [7,8,19], this tool is only defined for sHML, and thus fails to generate monitors for properties that are either unmonitorable or not expressed in this fragment. To this end, our final aim is to build a toolchain that takes an arbitrary recHML formula φ, generates its disjunctive form φ', and after extracting its strongest monitorable consequence φ'', synthesises an optimal monitor m, as outlined in Figure 1. We leave the first phase for future work and focus on the second one, represented by the component labelled SMC. The third phase will be handled by the detectEr tool. Our contributions are two-fold:

1. In Section 3, we present a prototype tool that generates the strongest monitorable consequence for arbitrary branching-time properties expressed in disjunctive recHML, based on the procedure proposed in [4].
2. In Section 5, we assess the performance of the implemented prototype tool against several specifications.

Fig. 1: Toolchain

2 Preliminaries

We assume a finite set of *actions* $a, b, \ldots \in \textsc{Act}$ and *processes* $p, q \ldots \in \textsc{Prc}$. The triple $\langle \textsc{Prc}, \textsc{Act}, \longrightarrow \rangle$ forms a Labelled Transition System (LTS), where $\longrightarrow \subseteq (\textsc{Prc}, \textsc{Act}, \textsc{Prc})$ is a transition relation and $(p, a, q) \in \longrightarrow$ is denoted by the suggestive notation $p \xrightarrow{a} q$. Traces are finite or infinite sequences of actions, $t, u \in \textsc{Act}^* \cup \textsc{Act}^w$, and we say that process p *produces trace* t if there exists a sequence of transitions $p \xrightarrow{a} q \xrightarrow{b} \cdots$ for $t = ab \cdots$. *Specifications* (or *properties*) are defined as sets of processes, $P, Q, R \in \mathcal{P}(\textsc{Prc})$, where P is a *consequence* of Q when $Q \subseteq P$.

2.1 The Specification Logic

In this RV set-up, we presuppose a specification logic to unambiguously describe the behaviour expected from the system under scrutiny, *i.e.,* the properties of states in the respective LTS. Properties are formulated in recHML, allowing a good level of generality of the obtained results. This set of formulae assumes a countably infinite supply of recursion variables $X, Y, \ldots \in \textsc{LVar}$ and is built using the actions in $\textsc{Act}$, as described in Figure 2. On the other hand, the semantics of recHML is given by the set of processes that satisfy each formula. We write $[\![\varphi, \rho]\!]$ to denote the set of processes that satisfy φ given an interpretation ρ of the free variables of the formula φ where $\rho : \textsc{LVar} \rightharpoonup \mathcal{P}(\textsc{Prc})$. The notation $\rho[X \mapsto P]$ represents an interpretation ρ' such that $\rho'(X) = P$ and $\rho'(Y) = \rho(Y)$ for all $Y \neq X$. A process satisfies the universal modality $[a]\varphi$ if *all* the states that it *can* reach after performing an a-labelled transition satisfy φ. Conversely, the existential modality $\langle a \rangle \varphi$ is satisfied by the processes that can perform *at least one* a-labelled transition and reach a state that satisfies φ. The fixed point in $\mathsf{min}\, X.\varphi$ and $\mathsf{max}\, X.\varphi$ binds all the free occurrences of X in φ, and we assume that for each recursion variable, there is only one such formula binding it. We call the subformula φ the *binding formula* of X and denote it as φ_X. Intuitively, these least and greatest fixed points allow for recursion, whereby they can be respectively interpreted as *reachability* and *invariance.*

Example 3. Specification S_2 from Example 1 for $\textsc{Act} = \{\mathsf{c}, \mathsf{g}, \mathsf{m}\}$ is formalised as formula φ_2 below.

$$\varphi_2 = \mathsf{min}\, Y.[c]\mathsf{ff} \wedge [g]Y \wedge [m]\varphi_1$$
$$\text{where } \varphi_1 = \mathsf{min}\, X.([m]X \wedge [g]X) \vee \langle c \rangle \mathsf{tt}$$

While the inner least fixed point in φ_2 (*i.e.,* formula φ_1) ensures that the system eventually produces a c event, the outermost least fixed point prohibits it from happening before the first occurrence of m. ■

2.2 Monitorability in recHML

A translation procedure known as *monitor synthesis* generates computational entities, termed *monitors*, from correctness specifications. These monitors are

recHML Syntax

$$
\begin{array}{llllll}
\varphi,\psi \in \text{recHML} ::= & \mathsf{tt} & \text{(truth)} & \mid \mathsf{ff} & \text{(falsehood)} \\
& \mid \varphi \vee \psi & \text{(disjunction)} & \mid \varphi \wedge \psi & \text{(conjunction)} \\
& \mid \langle a \rangle \varphi & \text{(existential modality)} & \mid [a]\varphi & \text{(universal modality)} \\
& \mid \mathsf{min}\, X.\varphi & \text{(least fixed point)} & \mid \mathsf{max}\, X.\varphi & \text{(greatest fixed point)} \\
& \mid X & \text{(recursion variable)} & &
\end{array}
$$

Branching-Time Semantics

$$
\begin{array}{ll}
[\![\mathsf{tt},\rho]\!] \stackrel{\textbf{def}}{=} \text{Prc} & [\![\mathsf{ff},\rho]\!] \stackrel{\textbf{def}}{=} \emptyset \\
[\![[a]\varphi,\rho]\!] \stackrel{\textbf{def}}{=} \{p \mid \forall q \cdot p \xrightarrow{a} q \text{ implies } q \in [\![\varphi,\rho]\!]\} & [\![\varphi \vee \psi,\rho]\!] \stackrel{\textbf{def}}{=} [\![\varphi,\rho]\!] \cup [\![\psi,\rho]\!] \\
[\![\langle a \rangle\varphi,\rho]\!] \stackrel{\textbf{def}}{=} \{p \mid \exists q \cdot p \xrightarrow{a} q \text{ and } q \in [\![\varphi,\rho]\!]\} & [\![\varphi \wedge \psi,\rho]\!] \stackrel{\textbf{def}}{=} [\![\varphi,\rho]\!] \cap [\![\psi,\rho]\!] \\
[\![\mathsf{min}\, X.\varphi,\rho]\!] \stackrel{\textbf{def}}{=} \bigcap \{P \mid [\![\varphi,\rho[X \mapsto P]]\!] \subseteq P\} & [\![X,\rho]\!] \stackrel{\textbf{def}}{=} \rho(X) \\
[\![\mathsf{max}\, X.\varphi,\rho]\!] \stackrel{\textbf{def}}{=} \bigcup \{P \mid P \subseteq [\![\varphi,\rho[X \mapsto P]]\!]\} &
\end{array}
$$

Fig. 2: The syntax and semantics of recHML in the branching-time setting.

then instrumented to run alongside the system and flag a verdict once they have observed sufficient runtime behaviour: an *acceptance* if the system satisfies the specification and a *rejection* if it violates it. These monitoring outcomes are assumed to be definite and irrevocable. In the branching-time setting, multi-verdict monitors (*i.e.*, monitors that may output both acceptance and rejection verdicts) are inconsistent [19, Theorem 2]. Therefore, we restrict our study to single-verdict monitors. To simplify the exposition, we focus on rejection monitors, *i.e.*, monitors that can only flag rejections. Monitors $m, n \in \text{Mon}$ can be described as suffix-closed sets of traces $m, n \subseteq \text{Act}^*$ that witness property violations [3,15], where Mon is the set of all possible monitors. More concretely, monitor m rejects process p, denoted as $\mathbf{rej}(m,p)$, if p produces a trace in m.

Definition 1 (Monitorability [19]). *A specification P is monitorable if there exists some $m \in \text{Mon}$ that is:*

1. sound *for specification P, i.e., for all $p \in \text{Prc}$, $\boldsymbol{rej}(m,p)$ implies $p \notin P$;*
2. complete *for specification P, i.e., for all $p \in \text{Prc}$, $p \notin P$ implies $\boldsymbol{rej}(m,p)$.*

Since monitors can only observe finite prefixes of a trace, several logical formulae from Figure 2, such as property φ_2 in Example 3, are not monitorable. Indeed, the work in [19] shows that the subset of formulae in recHML that is monitorable is characterised by the syntactic fragment sHML.

Theorem 1 (Safety Fragment [19]). *Formula $\varphi \in \text{recHML}$ is monitorable iff it is equivalent to a formula in the syntactic fragment sHML defined as below:*

$$\varphi, \psi \in \text{sHML} ::= \mathit{tt} \quad \mid \mathit{ff} \quad \mid [a]\varphi \quad \mid \varphi \wedge \psi \quad \mid \mathit{max}\, X.\varphi \quad \mid X \qquad \square$$

2.3 Extending the Limits of Monitorability

The syntax of sHML is restricted and, indeed, many recHML formulae are not monitorable [19]. However, this should not deter our efforts, since, in general, a part of those properties could be amenable to monitoring. For instance, although specification S_2 from Example 2 is not monitorable, specification S_3 (which is its consequence) is monitorable. Following a best-effort strategy, the work in [4] defines the notion of an *optimal monitor* that aims at capturing the best monitor among all possible sound monitors for a given property. There, the authors also show that such a monitor actually runtime verifies the *strongest monitorable consequence* of that property. In the rest of this section, we give an overview of those results as they form the theoretical foundation of our prototype tool.

Definition 2 (Optimal Monitor [4]). *Monitor m is* optimal *for property P whenever:*

1. *it is sound for P;*
2. *for all $n \in \text{Mon}$, if n is sound for P then $n \subseteq m$.*

Optimal monitors can be characterised in terms of the strongest monitorable consequence of the specification for which they are monitoring. In turn, this allows us to establish a correspondence between the two.

Definition 3 (Strongest Monitorable Consequence [4]). *The* strongest monitorable consequence *of specification P is a property Q that is monitorable such that:*

1. *it is a consequence of P,* i.e., *$P \subseteq Q$;*
2. *for any R that is monitorable, if $P \subseteq R$ then $Q \subseteq R$.*

Example 4. Properties φ_1 and φ_2 from Example 3 are not monitorable, and thus cannot be expressed in sHML. However, their strongest monitorable consequences can be respectively formalised as $\varphi_3 = \mathsf{tt}$ and $\varphi_4 = \mathsf{max}\, Y.[c]\mathsf{ff} \wedge [g]Y$. In such cases where the strongest monitorable consequence of a property is tt, then it is impossible to detect any violations from a finite prefix. ■

Theorem 2 ([4]). *A monitor $m \in M$ that is sound for P is optimal for P iff it is sound and complete for the strongest monitorable consequence of P.* □

From Theorem 2, we elaborate a two-step procedure to construct the optimal monitor for a property that first extracts its strongest monitorable consequence and then synthesises a sound and complete monitor for it. In this study, we focus on the former as the latter will be handled by the detectEr tool.

3 Design and Implementation

In this section, we give a detailed overview of the algorithm that constructs the strongest monitorable consequence of arbitrary recHML formulae, following

closely the procedure laid out in [4]. This construction consists of three steps: eliminating existential modalities, eliminating least fixed points, and eliminating disjunctions. Since these constructs are sources of non-monitorability, removing them from a recHML formula yields a formula which can be shown to be the strongest monitorable consequence. This procedure relies on two crucial assumptions, namely that formulae are in *disjunctive form* and all subformulae are *satisfiable*, with the exception of ff. We thus proceed to give all the necessary technical developments before delving into the implementation details.

3.1 Disjunctive Form

For a finite set of formulae Γ, we use the standard notation $\bigwedge \Gamma$ to denote the conjunction of all the formulae in Γ. Similarly, $\bigvee \Gamma$ denotes the disjunction of all the formulae in Γ. As usual, $\bigwedge \varnothing$ denotes tt and $\bigvee \varnothing$ denotes ff.

Definition 4 (Disjunctive Form [30]). *The set of* recHML *formulae in disjunctive form is given by the following grammar:*

$$\varphi, \psi \in \textsc{disHML} ::= \mathsf{tt} \quad | \; \mathsf{ff} \quad | \; \varphi \vee \psi \quad | \; \bigwedge_{a \in A} \left((\bigwedge_{\varphi \in \mathcal{B}_a} \langle a \rangle \varphi) \wedge [a] \bigvee_{\varphi \in \mathcal{B}_a} \varphi \right)$$
$$| \; \mathsf{min}\, X.\varphi \quad | \; \mathsf{max}\, X.\varphi \quad | \; X$$

where $A \subseteq$ Act and $\mathcal{B}_a \subseteq$ disHML is a finite set of formulae, where $a \in A$.

The conjunctions in disjunctive form denote that for each action $a \in A$, all formulae in $\mathcal{B}_a$ are satisfied by some a-successor, and all a-successors satisfy a formula in $\mathcal{B}_a$. The intuition behind this representation is to push conjunctions as far as possible towards the modalities to explicitly describe the interaction between conjuncts. As will be demonstrated in Section 3.2, this is crucial for constructing the strongest monitorable consequence.

Example 5. Consider formula $\varphi_5 = [c][g]\mathsf{ff} \wedge [c](\langle g \rangle \mathsf{tt} \vee [c]\mathsf{ff})$, whereby the subformula $\langle g \rangle \mathsf{tt} \vee [c]\mathsf{ff}$ represents the implication $[g]\mathsf{ff} \implies [c]\mathsf{ff}$. The conjuncts in φ_5 respectively describe the specifications *"g cannot occur immediately after c"* and *"if after c, g cannot occur, then c cannot occur either."* This property is not in disjunctive form, but it is equivalent to the the disjunctive formula φ_6 below, which describes the local behaviour *"after c, neither c nor g can occur."*

$$\varphi_6 = [c]\mathsf{ff} \vee \big(\langle c \rangle([g]\mathsf{ff} \wedge [c]\mathsf{ff}) \wedge [c]([g]\mathsf{ff} \wedge [c]\mathsf{ff})\big) \qquad \blacksquare$$

Walukiewicz [30] presents a procedure for constructing an equivalent disHML formula from any recHML one. However, in this paper, we focus on the computation of the strongest monitorable consequence and leave the conversion to disjunctive form for future work. The work in [30] also shows that satisfiability checking is linear; in our tool, all unsatisfiable subformulae are reduced to ff in a single pass.

Example 6. For the rest of this section, we use the following running example. Assume that $\textsc{Act} = \{c, m\}$ and consider $\varphi_7 = (\mathsf{max}\, X.[c]X \wedge [m]\mathsf{ff}) \wedge (\langle c \rangle \mathsf{tt} \vee [m]\mathsf{ff})$. This formula describes the property *"m never occurs, and if c cannot occur, then m cannot occur either."* Its equivalent disjunctive form is given by φ_8 below.

$$\varphi_8 = ([c]\mathsf{ff} \wedge [m]\mathsf{ff}) \vee ([m]\mathsf{ff} \wedge \langle c \rangle \varphi_8' \wedge [c]\varphi_8')$$
$$\text{where } \varphi_8' = \mathsf{max}\, X.([c]\mathsf{ff} \wedge [m]\mathsf{ff}) \vee (\langle c \rangle X \wedge [c]X \wedge [m]\mathsf{ff}) \qquad \blacksquare$$

3.2 Step 1: Eliminating Existential Modalities

In the first step, all occurrences of the existential modalities in the disjunctive formula are eliminated by replacing them with tt. Intuitively, this step is necessary since the non-existence of an a-successor, which would violate formulae of the form $\langle a \rangle \varphi$, cannot be identified by observing a single execution.

Remark 1. Disjunctive form is crucial for this step. Applying this transformation to φ_5 from Example 5 yields $[c]\big([g]\mathsf{ff} \wedge (\mathsf{tt} \vee [c]\mathsf{ff})\big)$, which can be simplified to $[c][g]\mathsf{ff}$. However, this is *not* the best approximation as the strongest monitorable consequence obtained from its disjunctive form, φ_6, is $[c][c]\mathsf{ff} \wedge [c][g]\mathsf{ff}$. $\blacksquare$

Example 7. Given formula φ_8 from Example 6, the algorithm automating this step returns the formula φ_9 below.

$$\varphi_9 = \big([c]\mathsf{ff} \wedge [m]\mathsf{ff}\big) \vee \big([m]\mathsf{ff} \wedge \mathsf{tt} \wedge [c]\mathsf{max}\, X.([c]\mathsf{ff} \wedge [m]\mathsf{ff}) \vee (\mathsf{tt} \wedge [c]X \wedge [m]\mathsf{ff})\big)$$

It is not hard to see that this induces several redundant terms. However, we ignore them for now as they will be handled in the ensuing step. $\blacksquare$

3.3 Step 2: Eliminating Least Fixed Points

The second step consists of transforming all least fixed points into greatest fixed points. Indeed, the only way to detect a violation of a least fixed point at runtime is to find a violation with a finite sequence of events, which is equivalent to detecting a violation of a greatest fixed point. We directly automate this by replacing all subformulae of the form $\mathsf{min}\, X.\varphi$ with $\mathsf{max}\, X.\varphi$.

This step, together with the previous one, induces several redundant subformulae, which, in turn, introduce a significant amount of unnecessary computation in the ensuing step. To this end, our algorithm recursively simplifies the resulting formula based on the axioms below, where $A \subseteq \textsc{Act}$, in the following order of precedence: $(A1)$, $(A2)$, ..., $(A6)$.

$$\text{(A1)}\quad \varphi \vee \mathsf{tt} \Rrightarrow \mathsf{tt} \qquad \text{(A2)}\quad \varphi \wedge \mathsf{tt} \Rrightarrow \varphi \qquad \text{(A3)}\quad [a]\mathsf{tt} \Rrightarrow \mathsf{tt}$$
$$\text{(A4)}\quad \mathsf{max}\, X.\mathsf{tt} \Rrightarrow \mathsf{tt} \qquad \text{(A5)}\quad \mathsf{max}\, X.X \Rrightarrow \mathsf{tt} \qquad \text{(A6)}\quad \mathsf{max}\, X. \bigwedge_{a \in A} [a]X \Rrightarrow \mathsf{tt}$$

$$\frac{\Gamma \cup \{\psi \vee \varphi\}}{\Gamma \cup \{\psi, \varphi\}}\,(\vee) \quad \frac{\Gamma \cup \{\psi \wedge \varphi\}}{\Gamma \cup \{\psi\} \qquad \Gamma \cup \{\varphi\}}\,(\wedge) \quad \frac{\Gamma \cup \{\mathsf{max}\, X.\varphi\}}{\Gamma \cup \{\varphi\}}\,(\mathsf{max}) \quad \frac{\Gamma \cup \{X\}}{\Gamma \cup \{\varphi_X\}}\,(X)$$

$$\frac{\Gamma}{\{\psi \mid [a]\psi \in \Gamma\}}\,([a]) \quad \frac{\Gamma \cup \{\mathsf{ff}\}}{\Gamma}\,(\mathsf{ff}) \quad \frac{\Gamma \cup \{\mathsf{tt}\}}{\{\mathsf{tt}\}}\,(\mathsf{tt}) \quad \frac{\Gamma \cup \{[a]\psi, [b]\varphi\}, a \neq b}{\{\mathsf{tt}\}}\,([a,b])$$

Fig. 3: Tableau rules where Γ is a formula set.

Example 8. Formula φ_9 from Example 7 does not have any least fixed points. Therefore, during the first pass, the algorithm automating this transformation leaves the formula unchanged. The second pass then returns the simplified formula with respect to the axioms ($A1$) to ($A6$), resulting in φ_{10} below.

$$\varphi_{10} = \big([c]\mathsf{ff} \wedge [m]\mathsf{ff}\big) \vee \big([m]\mathsf{ff} \wedge [c]\mathsf{max}\, X.([c]\mathsf{ff} \wedge [m]\mathsf{ff}) \vee ([c]X \wedge [m]\mathsf{ff})\big) \qquad \blacksquare$$

3.4 Step 3: Eliminating Disjunctions

The final and most challenging step is to obtain a disjunction-free formula. This can be decomposed into two parts: apply the tableau rules in Figure 3 to obtain a tree with back edges (*i.e.,* edges from leaves to inner nodes), and relabel the nodes of the tree. These are respectively automated by Algorithms 1 and 2.

Definition 5 (Tableau for Disjunction Elimination [4]). *Given a formula φ, its tableau is a pair $\langle T, L\rangle$, where T is a tree with back edges and L is a labelling function such that:*

1. *the root of T is labelled as $\{\varphi\}$, and*
2. *each internal node and its children are labelled according to a rule in Figure 3. Internal nodes are labelled with the premise, while their children are labelled with the conclusion. Additionally, rule $[a]$ is applied only when $L(n)$ matches the premise of no other rule and it contains at least one $[a]\varphi'$ for some φ'.*

The interpretation of the tableau in Definition 5 is that the formulae in Γ are disjuncted, whereas the branches are conjuncted. The rules are read top-down to form a tree, with the topmost premise being the root and the conclusions being the branches. Since formulae might not have unique tableaux, rule ($[a]$) must be left for last to synchronises the different tableaux of the same formula. This means that irrespective of the order of rule application, all children derived using rule ($[a]$) are identical. Additionally, although all the rules except for regenerations, *i.e.,* rule (X), reduce the formula size, the order of application directly influences the size of the tableau, which, in turn, affects the tool's performance.

Example 9. Consider the set of formulae $\{[a]\varphi_1 \wedge [a]\varphi_2, \mathsf{tt}\}$ for some φ_1, φ_2. This set pattern matches with the premise of two rules, namely (tt) and ($\wedge$). Applying the former, the formula set is immediately reduced to $\{\mathsf{tt}\}$. However, if the latter rule is applied, the tree branches into $\{[a]\varphi_1, \mathsf{tt}\}$ and $\{[a]\varphi_2, \mathsf{tt}\}$, both of which induce further proof obligations. $\blacksquare$

```
1  def CreateTableau(F, V, count)
2    if F = {tt} then
3      return Leaf(F, −1)
4    else if F = {ff} then
5      return Leaf(F, −1)
6    else
7      ⟨children, rule⟩ ← ApplyRule(F)
8      if children = [c] and ∃⟨c, x⟩ ∈ V for some x then
9        return Leaf(F, x)
10     else
11       V ← V ++ [⟨F, count⟩]
12       c_trees ← CreateTableau(c, V, count+1) for each c in children
13       return Node(count, F, c_trees, rule, false)
14 def SetBackedgeTargets(t)
15   targets ← [ ]
16   for each Leaf l in t where l.backedge_target ≠ −1 do
17     targets ← targets ++ [l.backedge_target]
18   for each Node n in t where n.node_id ∈ targets do
19     n.backedge ← true
20   return t
```

Alg. 1: Pseudocode for Building the Tableau

Our implementation circumvents the unnecessary computation steps induced by the application order of the rules by assuming the following order of priorities: (tt), ($[a, b]$), (ff), (max), ($\vee$), ($\wedge$), (X), ($[a]$). We chose this ordering based on the fact that the first two rules simplify the formula set, rule ($\vee$) increases the size of the formula set and thus the chance of applying (tt) or ($[a, b]$), whereas rules ($\wedge$) and (X) respectively increase the width and depth of the tree.

We implement the tableau in Definition 5 as a polymorphic tree: this allows us to use the same tree structure albeit with different implementations. Nodes are composed of *(i)* a node_id, *(ii)* a node_label, *(iii)* a list of children, where each child is a tree, *(iv)* a rule of type string, and *(v)* a boolean value, backedge, indicating whether that node is the target of some back edge. Leaves have two elements: *(i)* a leaf_label, and *(ii)* an integer value, backedge_target, to store the node_id of the back edge target; when there is no back edge, this is set to -1. For convenience, we use the suggestive dot notation (.) to access specific elements. *E.g.*, the rule applied at node n is accessed via the field n.rule.

The algorithm automating the tableau construction is described in Algorithm 1. The procedure starts from the root of the tableau, which is the singleton element $\{\varphi\}$, applies the rule with a matching premise, and then repeats this procedure for each resulting child. Since each node is a set of formulae, our algorithm implements the tableau as a tree of formula sets. The algorithm uses a set $\mathcal{F}$ and a list $\mathcal{V}$. The former is initialised to $\{\varphi\}$ and stores the formula set waiting to be analysed. The latter, initialised to empty, stores pairs $\langle \mathcal{S}, x\rangle$, where $\mathcal{S}$ is a formula set that has been already generated and x is its identifier. When each set $\mathcal{S}$ in $\langle \mathcal{S}, x\rangle \in \mathcal{V}$ can reach some other set $\mathcal{S}'$ in $\langle \mathcal{S}', x'\rangle \in \mathcal{V}$ via some rule, the algorithm terminates as no new leaves or nodes can be created.

Function CreateTableau( ) is the main function. If $\mathcal{F}$ is the singleton element $\{\mathsf{tt}\}$ or $\{\mathsf{ff}\}$, a leaf with no back edges is created on lines 3 and 5. Otherwise, the algorithm checks whether it needs to create a new node or add a back edge to some previous node in the tree by calling ApplyRule( ) on line 7. This function returns a pair containing a list of formula sets, which represent the current node's children, and the rule applied to derive them. If there is only one child c and $\langle c, x\rangle$ is in $\mathcal{V}$, then some node n with label c has already been generated. Thus, a leaf with a back edge to n is created on line 9 by setting backedge_target to x. Otherwise, the tree for each resulting child is constructed on line 12. Once the entire tree is constructed, SetBackedgeTargets( ) on line 14 performs two passes: it first retrieves the list of identifiers of the back edge targets from the leaves (lines 16,17), then traverses the tree again to update n.backedge to *true* for all nodes n that are target of some back edges (lines 18,19).

Example 10. Recall property φ_{10} in Example 8. Since tableaux tend to grow relatively in size, we focus on subformula $\varphi_{11} = \mathsf{max}\, X.([c]\mathsf{ff} \wedge [m]\mathsf{ff}) \vee ([c]X \wedge [m]\mathsf{ff})$, whose tableau is depicted by the left tree in Figure 4 and forms a subgraph of the tableau for φ_{10}. We omit the outer curly brackets denoting that the formulae form a set and only include specific elements for better readability.

Starting with the initial formula, Algorithm 1 creates a node with identifier 0 and a child with label $\{([c]\mathsf{ff} \wedge [m]\mathsf{ff}) \vee ([c]X \wedge [m]\mathsf{ff})\}$, obtained via rule max. Since the latter formula set has not been generated yet, its tree is created via a recursive call to CreateTableau( ) on line 12. This tree generation continues until the fifth recursive call, where a node with label $\{\mathsf{ff}, X\}$ and identifier 5 is created, whose child is the tree for formula set $\{X\}$. The latter has not been generated yet, but its child, obtained via rule X, has label $\{\varphi_X\}$ where φ_X is given by $([c]\mathsf{ff} \wedge [m]\mathsf{ff}) \vee ([c]X \wedge [m]\mathsf{ff})$: this is precisely the same as that of the node with identifier 1. Therefore, a leaf with backedge_target 1 is created (line 9). Once the entire tree is generated, SetBackedgeTargets( ) then updates the field backedge of the node with identifier 1 to *true*. ■

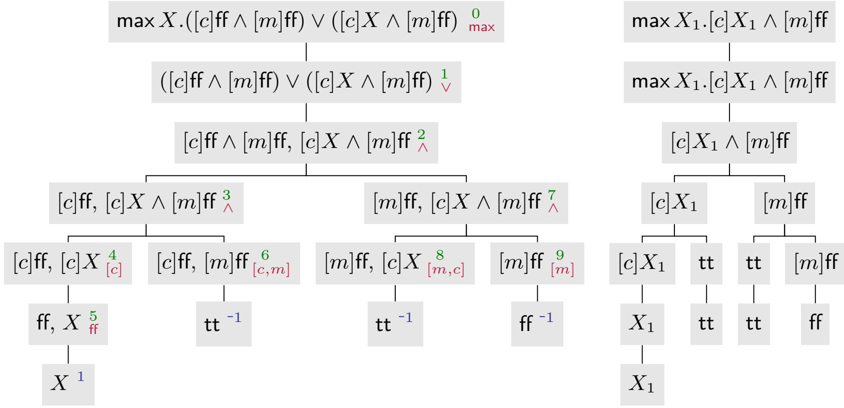

Fig. 4: The tableau for formula φ_{11} before and after relabelling.

```
1  def RelabelTableau(t)
2    if t = Leaf l then
3      m ← l.backedge_target
4      if m ≠ −1 then return Leaf(X_m, m)
5      else if tt ∈ l.leaf_label then return Leaf(tt, −1)
6      else return Leaf(ff, −1)
7    else t = Node n
8      children ← RelabelTableau(c) for c in n.children
9      if n.rule = (∧) then
10       [c_1, c_2] ← children
11       f ← Label(c_1) ∧ Label(c_2)
12     else if n.rule = [a] for some a then
13       [c] ← children
14       f ← [a]Label(c)
15     else
16       [c] ← children
17       f ← Label(c)
18     if n.backedge then
19       m ← n.id
20       return Node(m, max X_m.f, children, n.rule, true)
21     else
22       return Node(m, f, children, n.rule, false)
```

Alg. 2: Pseudocode for Relabelling the Tableau

The algorithm implementing the tableau relabelling is described in Algorithm 2. Given a tree of formula sets, RelabelTableau( ) recursively constructs a new tree of the same shape, whose root label is the strongest monitorable consequence. Each leaf of the inputted tree is relabelled to either X_m (where m is the identifier of the back edge target), tt or ff (lines 4,5,6). Each node is relabelled in two steps: the algorithm first relabels its children via a recursive call to RelabelTableau( ) on line 8, then it relabels the node according to which rule was applied to derive its children. For rule ($\wedge$), the new label is the conjunction of its two children c_1 and c_2, whereas for rule ($[a]$), it is that of its child prefixed by $[a]$ (lines 11,14). Otherwise, the label is identical to that of its child (line 17). Before creating a node with these values, line 18 checks whether it is the target of a back edge: if it is, the label is turned into a greatest fixed point by prefixing it with $\max X_m$ where m is the node's identifier. The function Label( ) retrieves the value of node_label or leaf_label, depending on the node type.

Remark 2. Implementing line 11 naively results in formulae with several redundant terms, where one of the conjuncts (or both) is tt. Our implementation sidesteps this by conjuncting c_1 and c_2 only if both are different from tt, and sets the label to c_1 when c_2 is equivalent to tt and vice versa. Also, all labels in the relabelled tableau consist a single formula. Arguably, such a tableau still can be implemented as a tree of formula sets, but it would impinge on the tool's performance as the algorithm would repeatedly have to retrieve the formula from the set. To this end, we implement the relabelled tableau as a tree of formulae, justifying why we opted to define the tree structure as a polymorphic type. □

Example 11. The relabelled tableau for formula φ_{11} from Example 10 is depicted by the right tree in Figure 4. Since its internal structure is analogous to that of the left tree, we omit the node identifiers and rules used to derive the children.

Starting from the root of the left tree, Algorithm 2 creates a new node with label $\mathsf{max}\, X_1.[c]X_1 \wedge [m]\mathsf{ff}$ on line 17. This is obtained by inheriting the label of its child, generated via a recursive call to RelabelTableau( ) on line 8. Since the node with identifier 1 is the target of a back edge, a new node is created on line 18 where its label is obtained in two steps. First, it retrieves the label of its child on line 17, and then transforms it into a greatest fixed point label by prefixing it with $\mathsf{max}\ X_1$. This tree generation continues until the leaves of the tree are reached: for the leaf with label $\{X\}$ and a back edge to node 1, a new leaf with label X_1 is created, whereas all the other leaves are left untouched. ■

Once Algorithm 2 returns the relabelled tableau, our prototype tool outputs its root label, which describes the strongest monitorable consequence of the initial formula.

4 The Tool

The previous section presented a thorough overview of the algorithms for constructing the strongest monitorable consequence. In this section, we give more details on the tool's internal architecture and how it can be used in practice.

4.1 Internal Architecture

The algorithms in Section 3 are implemented in OCaml (version 4.08.0) in a straightforward fashion, resulting in a prototype tool that takes as input a formula and outputs its strongest monitorable consequence. This is achieved by first building a parse tree for the inputted formula, using the *Menhir* parser generator, and then successively calling the functions automating the three steps in Section 3. The OCaml code is organised into several modules in the src/ directory, which can be further decomposed into three folders; definitions, parsing and utils. The first directory stores two modules, strongestMonCons.ml and SMCTableauRules.ml. The former implements the first two steps in Sections 3.2 and 3.3, while the latter implements Algorithms 1 and 2 and the tableau rules as the functions create_tableau(), relabel_tableau(), and apply_rules() respectively, preserving the naming conventions from Section 3.4. The second directory then contains the modules that handle the parsing, whereas the last stores all modules containing user-defined types and helper functions.

4.2 Usage

The tool can be invoked from the terminal, where the formula is inputted either by passing it as a command line argument or by providing the path of the file containing the formula when prompted. For instance, in the first approach,

the strongest monitorable consequence of φ_8 from Example 6 is generated by executing the command below, where the logical or's and and's are respectively substituted by the | and & operators.

```
./main.native "([c]ff & [m]ff) | ([m]ff & <c>(max X.([c]ff & [m]ff) |
    (<c>X & [c]X & [m]ff)) & [c](max X.([c]ff & [m]ff) | (<c>X & [c]X
    & [m]ff)))"
```

Conversely, the second approach involves omitting the formula altogether: this is especially appealing when formulae are more complex, while facilitating integration with the first component in the toolchain of Figure 1 once it is realized. The output returned by both methods can be decomposed into four parts; *(i)* the parse tree of the inputted formula, *(ii)* elimination of existential modalities, *(iii)* elimination of least fixed points, and *(iv)* elimination of disjunctions. Inputting formula φ_8 from Example 6, our tool returns the following, where the formulae outputted in *(ii)* and *(iii)* respectively correspond to φ_9 and φ_{10} from Examples 7 and 8.

```
================================ STEP 1 ================================

The formula after eliminating existential modalities is:
[c]ff & [m]ff | [m]ff & tt & [c](max X.[c]ff & [m]ff | tt & [c]X & [m]ff)

================================ STEP 2 ================================

The formula after eliminating minimal fixed points is:
[c]ff & [m]ff | [m]ff & tt & [c](max X.[c]ff & [m]ff | tt & [c]X & [m]ff)

The simplified formula is:
[c]ff & [m]ff | [m]ff & [c](max X.[c]ff & [m]ff | [c]X & [m]ff)
```

The fourth part of the output consists of two trees, representing the tableau of the formula returned in *(iii)* before and after relabelling. In our example, the output below shows a subtree of the tableau for φ_{10} before the relabelling, which corresponds to the tableau for φ_{11} from Example 10, depicted in Figure 4. We omit the output after the relabelling as it possesses a similar format.

```
 | └ (max)6 —— max X.[c]ff & [m]ff | [c]X & [m]ff;
 |   └ (or)7 —— [c]ff & [m]ff | [c]X & [m]ff; back edge target
 |     └ (and)8 —— [c]ff & [m]ff; [c]X & [m]ff;
 |       ├ (and)9 —— [c]X & [m]ff; [c]ff;
 |       | ├ ([c])10 —— [c]ff; [c]X;
 |       | | └ (ff)11 —— ff; X;
 |       | |   └ (X)12 —— X;
 |       | |     └ [c]ff & [m]ff | [c]X & [m]ff; back edge to 7
 |       | └ ([a,b])13 —— [c]ff; [m]ff;
 |       |   └ tt;
 |       └ (and)14 —— [c]X & [m]ff; [m]ff;
```

```
│           ├─ ([a,b])15 ─── [c]X; [m]ff;
│           │  └─ tt;
│           └─ ([m])16 ─── [m]ff;
│              └─ (ff)17 ─── ff;
```

Our tool can also export the computed strongest monitorable consequence to the format expected by detectEr tool and write it to file. This functionality can be triggered using the keyword `save`, as shown below. In turn, this allow us to input the file to detectEr, which will then synthesise the optimal monitor.

```
./main.native "([c]ff & [m]ff) | ([m]ff & <c>(max X.([c]ff & [m]ff) |
    (<c>X & [c]X & [m]ff)) & [c](max X.([c]ff & [m]ff) | (<c>X & [c]X
    & [m]ff)))" save
```

5 Evaluation

Sections 3 and 4 demonstrate that the procedure in [4] for computing the strongest monitorable consequence of recHML formulae can be automated, albeit with an exponential worst-case complexity upper bound. However, it remains unclear whether this is fully-representative of the implemented prototype. In this section, we evaluate the scalability our tool; in the absence of standard benchmarks, we devise two strategies for our empirical evaluation. All experiments were carried out on a Quad-Core Intel Core i5 64-bit machine with 16 GB memory, running OCaml version 4.08.0 on OSX Catalina. They can be *reproduced* using the sources provided at https://github.com/jasmine97xuereb/optimal-monitor.

Parametrisable Formulae. Since eliminating the existential modalities and least fixed points is linear, the disjunction elimination step is responsible for the overall complexity of the algorithm. We thus construct a family of disHML formulae aimed at maximizing the width and depth of the tableau that is constructed in Algorithm 1. More concretely, $P_1(k)$ below defines a family of formulae with a high branching-factor, resulting in a high level of branching in the tableau. The formulae generated by the skeleton $P_2(k)$ consist of several disjunctions and modalities over the same action, which blow-up the size of the formula sets. Additionally, several of these sets are composed of recursion variables $X_1, \ldots, X_n$, inducing further iterations in Algorithms 1 and 2. We contend that these skeletons adequately stress test our tool since the tableau construction of the generated formulae heavily relies on the application of rules $(\wedge)$, $(\vee)$, and (X), which induce the highest increase in tableaux size and complexity.

$$P_1(k) = \mathsf{max}\ X. \bigwedge_{i \in k} \Big(\bigwedge_{\varphi \in \mathcal{B}_{a_i}} \langle a \rangle \varphi \wedge [a] \bigvee_{\varphi \in \mathcal{B}_{a_i}} \varphi \Big) \text{ where } \mathcal{B}_a = \{[a]\mathsf{ff}, X\}$$

$$P_2(k) = \bigvee_{i \in k} \mathsf{max}\ X_i. \Big(\bigwedge_{j \in k} \langle a_j \rangle X_i \wedge [a_j] X_i \Big) \wedge [b_i]\mathsf{ff}$$

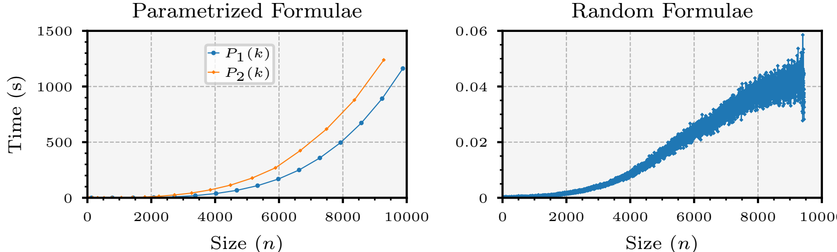

Fig. 5: Performance of the tool against different formulae

We evaluate the mean running time (over 5 repeated runs) for these property instances over an increasing parameter k. The results, reported in the left graph of Figure 5, show that for this set of properties, our implementation runs in quadratic time. We remark that, at this point, it is open to investigation whether the exponential worst-case complexity can be reached.

Random Formulae. Since the parametrised instances only target specific features of the algorithm, we also evaluate it against formulae that are randomly generated following a uniform distribution on the grammar of DISHML for better coverage. The plotted results in the right graph of Figure 5, which show the mean running time (over 500 repeated runs) of random formulae with increasing size, indicate that the average running time remains considerably lower than that for the family of formulae generated by the parameterised instances. Indeed, formulae are drastically simplified before reaching the third step, and modalities rarely interact in a way that make the tableau grow. We note that it is not clear how close the distribution adopted here is to the one obtained from uniformly chosen RECHML formulae that are *then* converted to disjunctive form.

Although there is no guarantee that the results obtained in this section carry over to the toolchain in Figure 1, they give preliminary evidence that our prototype tool scales well in the general case.

6 Conclusion

This paper investigates the implementability aspects of the procedure outlined in [4]. In particular, our prototype tool takes arbitrary branching-time properties expressed in disjunctive RECHML and constructs their best monitorable approximation according to [1,20]. This enables us to extend the known synthesis tools to generate optimal monitors for arbitrary branching-time properties. The tool and the accompanying demo video can be found at https://zenodo.org/badge/latestdoi/420058357 and https://youtu.be/XI6GoG4MaNk.

6.1 Future Work

We plan to automate the translation from RECHML formulae to their equivalent disjunctive form as presented in [30] to complete the toolchain of Figure 1. In

turn, this will allow us to investigate possible optimisations based on a more precise evaluation of our tool. Finally, we note that the detectEr tool can handle actions that carry data from an infinite domain. We plan to investigate to what extent the techniques in [4] generalise to this setting. This is a challenging endeavour as the automata-logic correspondence they rely on is far more complex in the presence of data [13,16].

6.2 Related Work

Linear- vs branching-time. In linear-time monitoring, we are interested in a property of the current execution, rather than the system as a whole. This is particularly useful for checking in deployed systems whether the output of a third-party component is safe to use in a critical component, for example. Then, whether the non-trusted component can also produce unsafe executions is largely irrelevant. Finding optimal monitors corresponds to computing the good and bad prefixes of a linear-time property, that is, the prefixes of which either all or no continuation satisfies the property, as done by Kupferman and Vardi [24] or by Havelund and Peled [20]. In contrast, when RV is used as a best-effort alternative to model-checking, we are trying to work out whether the *system*, rather than the current execution, is correct. Since monitors still only observe one execution, the proportion of monitorable properties is, unavoidably, smaller [2]. As a result, the benefit of using optimal monitors is even greater, as it expands the realm of properties that monitors can be used for. Note that the complexity of finding optimal monitors is double-exponential already in the linear-time setting [24], so the difficulty added by the branching-time setting is mostly conceptual.

Monitoring with prior knowledge. One of the use-cases for optimal monitors is the incorporation of prior knowledge (assumptions) into the monitor, which allows more violations to be identified. As argued in [4], computing the optimal monitor is also an optimal way to incorporate prior knowledge into the monitor. This problem has been studied in the linear-time setting (with linear-time assumptions) by Henzinger and Saraç [22], by Cimatti *et al.* [14], and by Leucker [25], and for hyperproperties by Stucki *et al.* [29].

Monitoring tools. Among many RV tools, let us mention MaC [23], PathExplorer [21], Eagle [9], and RuleR [10], Temporal Rover [17], and JavaMOP [26], all runtime verification tools based on various specification languages. Blech *et al.* [27], Schneider *et al.* [28] and Basin *et al.* [12] aim to generate *verified* monitors from specifications. The former uses proof assistant Coq and targets regular properties, while the latter two use Isabelle/HOL and target metric first order temporal logics; both produce executable monitors in OCaml. Typically, these tools focus on linear-time specifications, which makes them harder to adapt to properties generated primarily for model-checking rather than RV, and does not lend them to incorporating prior knowledge of the system, expressed as branching time properties, into the monitoring set-up.

References

1. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A.: Monitoring for Silent Actions. In: FSTTC. LIPIcs, vol. 93, pp. 7:1–7:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017)
2. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: Adventures in Monitorability: From Branching to Linear Time and Back Again. PACMPL **3**(POPL), 52:1–52:29 (2019)
3. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: An Operational Guide to Monitorability. In: SEFM. LNCS, vol. 11724, pp. 433–453 (2019)
4. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: The Best a Monitor Can Do. In: CSL. LIPIcs, vol. 183, pp. 7:1–7:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
5. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Reactive Systems: Modelling, Specification and Verification. Cambridge U.P. (2007)
6. Alpern, B., Schneider, F.B.: Recognizing Safety and Liveness. Distributed Comput. **2**(3), 117–126 (1987)
7. Attard, D.P., Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: Better Late Than Never or: Verifying Asynchronous Components at Runtime. In: IFIP. LNCS, vol. 12719, pp. 207–225. Springer (2021)
8. Attard, D.P., Francalanza, A.: A Monitoring Tool for a Branching-Time Logic. In: RV. LNCS, vol. 10012, pp. 473–481. Springer (2016)
9. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-Based Runtime Verification. In: VMCAI. LNCS, vol. 2937, pp. 44–57. Springer (2004)
10. Barringer, H., Rydeheard, D., Havelund, K.: Rule Systems for Run-time Monitoring: from Eagle to RuleR. Journal of Logic and Computation **20**(3), 675–706 (2008)
11. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to Runtime Verification. In: Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS, vol. 10457, pp. 1–33. Springer (2018)
12. Basin, D., Dardinier, T., Heimes, L., Krstić, S., Raszyk, M., Schneider, J., Traytel, D.: A Formally Verified, Optimized Monitor for Metric First-Order Dynamic Logic. In: Automated Reasoning. pp. 432–453. Springer (2020)
13. Björklund, H., Schwentick, T.: On notions of regularity for data languages. Theor. Comput. Sci. **411**(4-5), 702–715 (2010)
14. Cimatti, A., Tian, C., Tonetta, S.: Assumption-Based Runtime Verification with Partial Observability and Resets. In: Runtime Verification. LNCS, vol. 11757, pp. 165–184. Springer (2019)
15. d'Amorim, M., Rosu, G.: Efficient Monitoring of omega-Languages. In: CAV. LNCS, vol. 3576, pp. 364–378. Springer (2005)
16. D'Antoni, L.: In the Maze of Data Languages. CoRR **abs/1208.5980** (2012)
17. Drusinsky, D.: The Temporal Rover and the ATG Rover. In: International SPIN Workshop on Model Checking of Software. LNCS, vol. 1885, pp. 323–330. Springer (2000)
18. Francalanza, A.: A Theory of Monitors. Inf. Comput. **281**, 104704 (2021)
19. Francalanza, A., Aceto, L., Ingólfsdóttir, A.: Monitorability for the Hennessy-Milner logic with recursion. FMSD **51**(1), 87–116 (2017)
20. Havelund, K., Peled, D.: Runtime Verification: From Propositional to First-Order Temporal Logic. In: International Conference on Runtime Verification. LNCS, vol. 11237, pp. 90–112. Springer (2018)

21. Havelund, K., Roşu, G.: Monitoring Java Programs with Java PathExplorer. Electronic Notes in Theoretical Computer Science **55**(2), 200–217 (2001)
22. Henzinger, T.A., Saraç, N.E.: Monitorability Under Assumptions. In: Runtime Verification. LNCS, vol. 12399, pp. 3–18. Springer (2020)
23. Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I., Sokolsky, O.: Formally specified monitoring of temporal properties. In: ECRTS. pp. 114–122. IEEE (1999)
24. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods in System Design **19**(3), 291–314 (2001)
25. Leucker, M.: Sliding between Model Checking and Runtime Verification. In: RV. LNCS, vol. 7687, pp. 82–87 (2012)
26. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. International Journal on Software Tools for Technology Transfer **14**(3), 249–289 (2012)
27. Olaf Blech, J., Falcone, Y., Becker, K.: Towards Certified Runtime Verification. In: ICFEM. LNCS, vol. 7635, pp. 494–509. Springer (2012)
28. Schneider, J., Basin, D., Krstić, S., Traytel, D.: A Formally Verified Monitor for Metric First-Order Temporal Logic. In: RV. LNCS, vol. 11757, pp. 310–328. Springer (2019)
29. Stucki, S., Sánchez, C., Schneider, G., Bonakdarpour, B.: Gray-box monitoring of hyperproperties with an application to privacy. FMSD pp. 1–34 (2021)
30. Walukiewicz, I.: Completeness of Kozen's Axiomatisation of the Propositional mu-Calculus. In: Proceedings, 10th Annual IEEE Symposium on Logic in Computer Science. pp. 14–24. IEEE (1995)