





# Monitoring Hyperproperties with Prefix Transducers

Marek Chalupa<sup>(✉)</sup> and Thomas A. Henzinger

Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria  
marek.chalupa@ist.ac.at

**Abstract.** Hyperproperties are properties that relate multiple execution traces. Previous work on monitoring hyperproperties focused on synchronous hyperproperties, usually specified in HyperLTL. When monitoring synchronous hyperproperties, all traces are assumed to proceed at the same speed. We introduce (multi-trace) *prefix transducers* and show how to use them for monitoring synchronous as well as, for the first time, asynchronous hyperproperties. Prefix transducers map multiple input traces into one or more output traces by incrementally matching prefixes of the input traces against expressions similar to regular expressions. The prefixes of different traces which are consumed by a single matching step of the monitor may have different lengths. The deterministic and executable nature of prefix transducers makes them more suitable as an intermediate formalism for runtime verification than logical specifications, which tend to be highly non-deterministic, especially in the case of asynchronous hyperproperties. We report on a set of experiments about monitoring asynchronous version of observational determinism.

## 1 Introduction

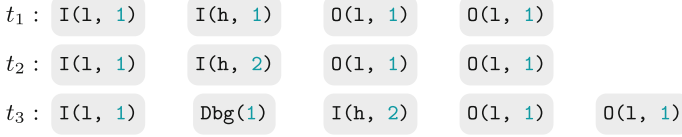
Hyperproperties [20] are properties that relate multiple execution traces of a system to each other. One of the most prominent examples of hyperproperties nowadays are the information-flow security policies [33]. Runtime monitoring [1] is a lightweight formal method for analyzing the behavior of a system by checking dynamic execution traces against a specification. For hyperproperties, a monitor must check relations between multiple traces. While many hyperproperties cannot be monitored in general [3, 13, 25], the monitoring of hyperproperties can still yield useful results, as we may detect their violations [24].

Previous work on monitoring hyperproperties focused on HyperLTL specifications [3, 13], or other synchronous hyperlogics [2]. Synchronous specifications model processes that progress at the same speed in lockstep, one event on each trace per step. The synchronous time model has been found overly restrictive for specifying hyperproperties of asynchronous processes, which may proceed at varying speeds [7, 9, 12, 27]. A more general, asynchronous time model allows multiple traces to proceed at different speeds, independently of each other, in order to wait for each other only at certain synchronization events. As far as we know, there has been no previous work on the runtime monitoring of asynchronous hyperproperties.

© The Author(s) 2023

P. Katsaros and L. Nenzi (Eds.): RV 2023, LNCS 14245, pp. 168–190, 2023.

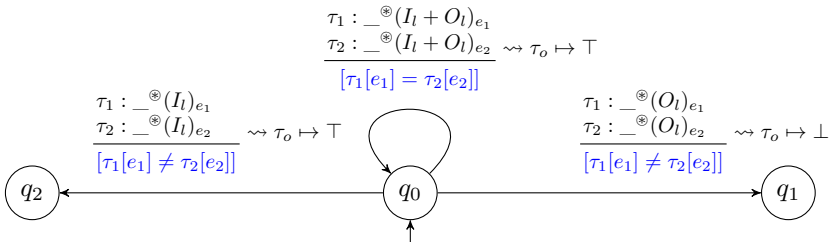
[https://doi.org/10.1007/978-3-031-44267-4\\_9](https://doi.org/10.1007/978-3-031-44267-4_9)



**Fig. 1.** Traces of abstract events. The event  $I(x, v)$  signals an input of value  $v$  into variable  $x$ ; and  $O(x, v)$  signals an output of value  $v$  from variable  $x$ . The event  $Dbg(b)$  indicates whether the debugging mode is turned on or off.

The important class of  $k$ -safety hyperproperties [20,22] can be monitored by processing  $k$ -tuples of traces [3,22]. In this work, we develop and evaluate a framework for monitoring  $k$ -safety hyperproperties under both, synchronous and asynchronous time models. For this purpose, we introduce (*multi-trace prefix transducers*), which map multiple (but fixed) input traces into one or more output traces by incrementally matching prefixes of the input traces against expressions similar to regular expressions. The prefixes of different traces which are consumed by a single matching step of the transducer may have different lengths, which allows to proceed on the traces asynchronously. By instantiating prefix transducers for different combinations of input traces, we can monitor  $k$ -safety hyperproperties instead of monitoring only a set of fixed input traces. The deterministic and executable nature of prefix transducers gives rise to natural monitors. This is in contrast with monitors synthesized from logical specifications which are often highly non-deterministic, especially in the case of asynchronous (hyper)properties.

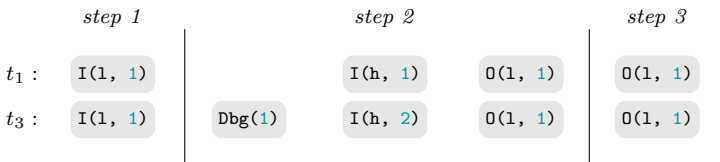
We illustrate prefix transducers on the classical example of *observational determinism* [37]. Informally, observational determinism (OD) states that whenever two execution traces agree on *low* (publicly visible) inputs, they must agree also on low outputs, thus not leaking any information about *high* (secret) inputs. Consider, for example, the traces  $t_1$  and  $t_2$  in Fig. 1. These two traces satisfy OD, because they have the same low inputs (events  $I(1, \cdot)$ ) and produce the same low outputs (events  $O(1, \cdot)$ ). All other events in the traces are irrelevant to OD. The two traces even satisfy synchronous OD, as the input and output events appear at the same positions in both traces, and thus they can be analysed by synchronous-time monitors (such as those based on HyperLTL [3,13,24]), or by the following prefix transducer:



The transducer reads two traces  $\tau_1$  and  $\tau_2$  that are instantiated with actual traces, e.g.,  $t_1$  and  $t_2$ . It starts in the initial state  $q_0$  and either repeatedly takes the self-loop transition, or goes into one of the states  $q_1$  or  $q_2$  where it gets stuck. The self-loop transition matches the shortest prefix of  $\tau_1$  that contains any events until a low input or output is found. This is represented by the *prefix expression*  $\_{}^*(I_l + O_l)$ , where we use  $I_l$  (resp.  $O_l$ ) to represent any low input (resp. output) event, and  $\_{}^*$  stands for any event that does not match the right-hand side of  $\_{}^*$ . The same pattern is independently matched by this transition also against the prefix of  $\tau_2$ . Moreover, the low input or output events found on traces  $\tau_1$  and  $\tau_2$  are labeled by  $e_1$  and  $e_2$ , resp. The self-loop transition is taken if the prefixes of  $\tau_1$  and  $\tau_2$  match the expressions and, additionally, the condition  $\tau_1[e_1] = \tau_2[e_2]$  is fulfilled. The term  $\tau[e]$  denotes the sequence of events in trace  $\tau$  on the position(s) labeled by  $e$ . Therefore, the condition  $\tau_1[e_1] = \tau_2[e_2]$  asserts that the matched input or output events must be the same (both in terms of type and values). If the transducer takes the self-loop transition, it outputs (appends) the symbol  $\top$  to the output trace  $\tau_o$  (as stated by the right-hand side of  $\rightsquigarrow$ ). Then, the matched prefixes are consumed from the input traces and the transducer continues with matching the rest of the traces. The other two edges are processed analogously.

It is not hard to see that the transducer decides if OD holds for two *synchronous* input traces. State  $q_0$  represents the situation when OD holds but may still be violated in the future. If the self-loop transition over  $q_0$  cannot be taken, then (since we now assume synchronised traces), the matched prefixes must end either with different low input or different low output events. In the first case, OD is satisfied by the two traces and the transducer goes to state  $q_2$  where it gets stuck (we could, of course, make the transducer total to avoid getting stuck, but in this example we are interested only in its output). In the second case, OD is violated and before the transducer changes the state to  $q_1$ , it appends  $\perp$  to the output trace  $\tau_o$ . OD is satisfied if  $\tau_o$  does not contain (end with)  $\perp$  after finishing reading (or getting stuck on) the input traces.

The transducer above works also for monitoring asynchronous OD, where the low input and output events are misaligned by “padding” events (but it requires that there is the same number and order of low input and output events on the input traces – they are just misaligned and possibly carry different values; the general setup where the traces can be arbitrary is discussed in Sect. 5). The transducer works for asynchronous OD because the prefix expressions for  $\tau_1$  and  $\tau_2$  are matched independently, and thus they can match prefixes of different lengths. For example, for  $\tau_1 = t_1$  and  $\tau_2 = t_3$ , the run consumes the traces in the following steps:



Hitherto, we have used the output of prefix transducers to decide OD for the given two traces, i.e., to perform the monitoring task. We can also define a prefix transducer that, instead of monitoring OD for the two traces, transforms the asynchronous traces  $\tau_1$  and  $\tau_2$  into a pair of synchronous traces  $\tau'_1$  and  $\tau'_2$  by filtering out “padding” events:

$$\tau_1 : \_{}^{\otimes}(I_l + O_l)_{e_1} \rightsquigarrow \tau'_1 \mapsto \tau_1[e_1] \quad \begin{array}{c} \downarrow \\ \textcircled{q_0} \\ \leftarrow \quad \rightarrow \end{array} \quad \tau_2 : \_{}^{\otimes}(I_l + O_l)_{e_2} \rightsquigarrow \tau'_2 \mapsto \tau_2[e_2]$$

In this example, the transducer appends every event labeled by  $e_i$  to the output trace  $\tau'_i$ , and so it filters out all events except low inputs and outputs. It reads and filters both of the input traces independently of each other. The output traces from the transducer can then be forwarded to, e.g., a HyperLTL monitor<sup>1</sup>.

**Contributions.** This paper makes the following contributions:

- We introduce multi-trace prefix expressions and transducers (Sect. 2 and Sect. 3). These are formalisms that can efficiently and incrementally process words (traces) either fully synchronously, or asynchronously with synchronization points.
- We suggest that prefix transducers are a natural formalism for specifying many synchronous and asynchronous  $k$ -safety hyperproperties, such as observational determinism.
- We design an algorithm for monitoring synchronous and asynchronous  $k$ -safety hyperproperties using prefix transducers (Sect. 4).
- We provide some experiments to show how our monitors perform (Sect. 5).

## 2 Prefix Expressions

In this section, we define *prefix expressions* – a formalism similar to regular expressions designed to deterministically and unambiguously match prefixes of words.

### 2.1 Preliminaries

We model sequences of events as *words* over finite non-empty alphabets. Given an alphabet  $\Sigma$ , the set of finite words over this alphabet is denoted as  $\Sigma^*$ . For two words  $u = u_0 \dots u_l \in \Sigma_1^*$  and  $v = v_0 \dots v_m \in \Sigma_2^*$ , their concatenation  $u \cdot v$ , also written  $uv$  if there is no confusion, is the word  $u_0 \dots u_l v_0 \dots v_m \in (\Sigma_1 \cup \Sigma_2)^*$ . If

<sup>1</sup> One more step is needed before we can use a HyperLTL monitor, namely, to transform the trace of abstract events from the example into a trace of sets of atomic propositions. This can be also done by a prefix transducer.

$w = uw$ , we say that  $u$  is a prefix of  $w$ , written  $u \leq w$ , and  $v$  is a suffix of  $w$ . If  $u \leq w$  and  $u \neq w$ , we say that  $u$  is a proper prefix of  $w$ .

For a word  $w = w_0 \dots w_{k-1} \in \Sigma^*$ , we denote  $|w| = k$  its length,  $w[i] = w_i$  for  $0 \leq i < k$  its  $i$ -th element,  $w[s..e] = w_s w_{s+1} \dots w_e$  the sub-word beginning at index  $s$  and ending at index  $e$ , and  $w[s..] = w_s w_{s+1} \dots w_{k-1}$  its suffix starting at index  $s$ .

Given a function  $f : A \rightarrow B$ , we denote  $Dom(f) = A$  its domain. Partial functions with domain  $A$  and codomain  $B$  are written as  $A \hookrightarrow B$ . Functions with a small domain are sometimes given extensionally by listing the mapping, e.g.,  $\{x \mapsto 1, y \mapsto 2\}$ . Given a function  $f$ ,  $f[x \mapsto c]$  is the function that coincides with  $f$  on all elements except on  $x$  where it is  $c$ .

## 2.2 Syntax of Prefix Expressions

Let  $L$  be a non-empty set of labels (names) and  $\Sigma$  a finite non-empty alphabet. The syntax of *prefix expressions* ( $PE_{\underline{L}}$ ) is defined by the following grammar:

$$\begin{aligned} \alpha &::= \epsilon \mid a \mid (\alpha.\alpha) \mid (\alpha + \alpha) \mid (\alpha^{\otimes}\beta) \mid (\alpha)_l \\ \beta &::= a \mid (\beta + \beta) \mid (\beta)_l \end{aligned}$$

where  $a \in \Sigma$  and  $l \in L$ . Many parenthesis can be elided if we let ‘ $\otimes$ ’ (iteration) take precedence before ‘ $\cdot$ ’ (concatenation), which takes precedence before ‘ $+$ ’ (disjunction, plus). We write  $a.b$  as  $ab$  where there is no confusion. In the rest of the paper, we assume that a set of labels  $L$  is implicitly given, and that  $L$  always has „enough” labels. We denote the set of all prefix expressions over the alphabet  $\Sigma$  (and any set of labels  $L$ ) as  $PE(\Sigma)$ , and  $PE(\Sigma, L)$  if we want to stress that the expressions use labels from  $L$ .

The semantics of PEs (defined later) is similar to the semantics of regular expressions with the difference that a PE is not matched against a whole word but it matches only its prefix, and this prefix is the shortest one possible (and non-empty – if not explicitly specified). For this reason, we do not use the classical Kleene’s iteration as it would introduce ambiguity in matching. For instance, the regular expression  $a^*$  matches all the prefixes of the word  $aaa$ . And even if we specify that we should pick the shortest one, the result would be  $\epsilon$ , which is usually not desirable, because that means no progress in reading the word. Picking the shortest non-empty prefix would be a reasonable solution in many cases, but the problem with ambiguity persists. For example, the regular expression  $(ab)^*(a+b)^*$  matches the word  $ab$  in two different ways, which introduces non-determinism in the process of associating the labels with the matched positions.

To avoid the problems with Kleene’s iteration, we use a binary iteration operator that is similar to the *until* operator in LTL in that it requires some letter to appear eventually. The expression  $\alpha^{\otimes}\beta$  could be roughly defined as  $\beta + \alpha\beta + \alpha^2\beta + \dots$  where we evaluate it from left to right and  $\beta$  must match *exactly* one letter. The restriction on  $\beta$  is important to tackle ambiguity, but it also helps efficiently evaluate the expression – it is enough to look at a single letter to decide whether to continue matching whatever follows the iteration, or

whether to match the left-hand side of the expression. Allowing  $\beta$  to match a sequence of letters would require a look-ahead or backtracking and it is a subject of future extensions. With our iteration operator, expressions like  $(ab)^{\otimes}(a+b)^{\otimes}$  and  $a^{\otimes}$  are malformed and forbidden already on the level of syntax.

Sub-expressions of a PE can be labeled and the matching procedure described later returns a list of positions in the word that were matched for each of the labels. We assume that every label is used maximally once, that is, no two sub-expressions have the same label. Labels in PEs are useful for identifying the sub-word that matched particular sub-expressions, which will be important in the next section when we use logical formulae that relate sub-words from *different* words (traces). Two examples of PEs and their informal evaluation are:

- $(a+b)_l^{\otimes}a$  - match  $a$  or  $b$ , associating them to  $l$ , until you see  $a$ . Because whenever the word contains  $a$ , the right-hand side of the iteration matches, the left part of the iteration never matches  $a$  and  $a$  is redundant in the left-hand side sub-expression. For the word  $bbbaba$ , the expression matches the prefix  $bbba$  and  $l$  is associated with the list of position ranges  $(0, 0)$ ,  $(1, 1)$ ,  $(2, 2)$  that corresponds to the positions of  $b$  that were matched by the sub-expression  $(a+b)$ .
- $(a^{\otimes}b)_{l_1}((b+c)^{\otimes}(a+d))_{l_2}$  - match  $a$  until  $b$  is met and call this part  $l_1$ ; then match  $b$  or  $c$  until  $a$  or  $d$  and call that part  $l_2$ . For the word  $aabbada$ , the expression matches the prefix  $aabbba$ . The label  $l_1$  is associated with the range of positions  $(0, 2)$  containing the sub-word  $aab$ , and  $l_2$  with the range  $(3, 5)$  containing the sub-word  $bba$ .

### 2.3 Semantics of Prefix Expressions

We first define *m-strings* before we get to the formal semantics of PEs. An m-string is a sequence of pairs of numbers or a special symbol  $\perp$ . Intuitively,  $(p, \perp)$  represents the beginning of a match at position  $p$  in the analyzed word,  $(\perp, p)$  the end of a match at position  $p$ , and  $(s, e)$  is a match on positions from  $s$  to  $e$ . The concatenation of m-strings reflects opening and closing the matches.

**Definition 1 (M-strings).** *M-strings are words over the alphabet  $M = (\mathbb{N} \cup \{\perp\}) \times (\mathbb{N} \cup \{\perp\})$  with the partial concatenation function  $\odot : M^* \times M \hookrightarrow M^*$  defined as*

$$\alpha \odot (c, d) = \begin{cases} (c, d) & \text{if } \alpha = \epsilon \wedge c \neq \perp \\ \alpha \cdot (c, d) & \text{if } \alpha = \alpha' \cdot (a, b) \wedge b \neq \perp \\ \alpha' \cdot (a, d) & \text{if } \alpha = \alpha' \cdot (a, b) \wedge b = \perp \wedge c = \perp \\ \alpha' \cdot (c, d) & \text{if } \alpha = \alpha' \cdot (a, b) \wedge b = \perp \wedge c \neq \perp \end{cases}$$

Every m-string is built up from the empty string  $\epsilon$  by repeatedly concatenating  $(p, \perp)$  or  $(\perp, p)$  or  $(p, p)$ . The concatenation  $\odot$  is only a partial function (e.g.,  $\epsilon \odot (\perp, \perp)$  is undefined), but we will use only the defined fragment of the domain. It works as standard concatenation if the last match was closed

(e.g.,  $(0, 4) \odot (7, \perp) = (0, 4)(7, \perp)$ ), but it overwrites the last match if we start a new match without closing the old one (e.g.,  $(0, \perp) \odot (2, \perp) = (2, \perp)$ ). Overwriting the last match in this situation is valid, because labels are assumed to be unique and opening a new match before the last match was closed means that the old match failed.

We extend  $\odot$  to work with m-strings on the right-hand side:

$$\alpha \odot w = \begin{cases} \alpha & \text{if } w = \epsilon \\ (\alpha \odot x) \odot w' & \text{if } w = x \cdot w' \end{cases}$$

While evaluating a PE, we keep one m-string per label used in the PE. To do so we use *m-maps*, partial mappings from labels to m-strings.

**Definition 2 (M-map).** Let  $L$  be a set of labels and  $M = (\mathbb{N} \cup \{\perp\}) \times (\mathbb{N} \cup \{\perp\})$  be the alphabet of m-strings. An m-map is a partial function  $m : L \hookrightarrow M^*$ . Given two m-maps  $m_1, m_2 : L \hookrightarrow M^*$ , we define their concatenation  $m_1 \odot m_2$  by

$$(m_1 \odot m_2)(l) = \sigma_1 \odot \sigma_2$$

for all  $l \in \text{Dom}(m_1) \cup \text{Dom}(m_2)$  where  $\sigma_i = \epsilon$  if  $m_i(l)$  is not defined and  $\sigma_i = m_i(l)$  otherwise.

We denote the set of all m-maps  $L \hookrightarrow M^*$  for the set of labels  $L$  as  $\mathbb{M}_L$ .

The evaluation of a PE over a word  $w$  is defined as an iterative application of a set of rewriting rules that are inspired by language derivatives [5, 14]. For the purposes of evaluation, we introduce two new prefix expressions:  $\perp^2$  and  $[\alpha]_l$ . PE  $\perp$  represents a failed match and  $[\alpha]_l$  is a ongoing match of a labeled sub-expressions  $(\alpha)_l$ . Further, for a PE  $\alpha$ , we define inductively that  $\epsilon \in \alpha$  iff

- $\alpha = \epsilon$ , or
- $\alpha$  is one of  $(\alpha_0 + \alpha_1)$  or  $(\alpha_0 + \alpha_1)_l$  or  $[\alpha_0 + \alpha_1]_l$  and it holds that  $(\epsilon \in \alpha_0 \vee \epsilon \in \alpha_1)$ .

For the rest of this section, let us fix a set of labels  $L$ , an alphabet  $\Sigma$ , and denote  $\mathbb{E} = PE(\Sigma, L)$  the set of all prefix expressions over this alphabet and this set of labels. The core of evaluation of PEs is the *one-step relation*  $\xrightarrow{a, p} \subseteq (\mathbb{E} \times \mathbb{M}_L) \times (\mathbb{E} \times \mathbb{M}_L)$  defined for each letter  $a$  and natural number  $p$ , whose defining rules are depicted in Fig. 2. We assume that the rules are evaluated modulo the equalities  $\epsilon \cdot \alpha = \alpha \cdot \epsilon = \alpha$ , and we say that  $\alpha'$  is a *derivation* of  $\alpha$  if  $\alpha \xrightarrow{a, p} \alpha'$  for some  $a$  and  $p$ .

The first seven rules in Fig. 2 are rather standard. Rules for disjunction are non-standard in the way that whenever an operand of a disjunction is evaluated to  $\epsilon$ , the whole disjunction is evaluated to  $\epsilon$  (rule **OR-END**) in order to obtain the shortest match. Also, after rewriting a disjunction, operands that evaluated

---

<sup>2</sup> Because PEs and m-strings never interact together, we use the symbol  $\perp$  in both, but formally they are different symbols.

$$\begin{array}{c}
\frac{}{(\epsilon, M) \xrightarrow{a,p} (\perp, \emptyset)} \text{ (EPS)} \qquad \frac{}{(\perp, M) \xrightarrow{a,p} (\perp, \emptyset)} \text{ (BOT)} \qquad \frac{}{(a, M) \xrightarrow{a,p} (\epsilon, M)} \text{ (LTR)} \\
\\
\frac{a \neq b}{(\alpha, M) \xrightarrow{b,p} (\perp, \emptyset)} \text{ (LTR-FAIL)} \qquad \frac{(\alpha, M) \xrightarrow{a,p} (\alpha', M') \quad \alpha' \neq \perp \quad \epsilon \notin \alpha'}{(\alpha\beta, M) \xrightarrow{a,p} (\alpha'\beta, M')} \text{ (CONCAT)} \\
\\
\frac{(\alpha, M) \xrightarrow{a,p} (\alpha', M') \quad \epsilon \in \alpha'}{(\alpha\beta, M) \xrightarrow{a,p} (\beta, M')} \text{ (CONCAT-}\epsilon\text{)} \qquad \frac{(\alpha, M) \xrightarrow{a,p} (\alpha', M') \quad \alpha' = \perp}{(\alpha\beta, M) \xrightarrow{a,p} (\perp, \emptyset)} \text{ (CONCAT-}\perp\text{)} \\
\\
\frac{(\alpha_0, M) \xrightarrow{a,p} (\alpha'_0, M'_0) \quad (\alpha_1, M) \xrightarrow{a,p} (\alpha'_1, M'_1) \quad \epsilon \in \alpha'_0 \vee \epsilon \in \alpha'_1}{(\alpha_0 + \alpha_1, M) \xrightarrow{a,p} (\epsilon, M'_0 \odot M'_1)} \text{ (OR-END)} \\
\\
\frac{(\alpha_0, M) \xrightarrow{a,p} (\alpha'_0, M'_0) \quad (\alpha_1, M) \xrightarrow{a,p} (\alpha'_1, M'_1) \quad \epsilon \notin \alpha'_0 \wedge \epsilon \notin \alpha'_1}{(\alpha_0 + \alpha_1, M) \xrightarrow{a,p} (\alpha'_0 + \alpha'_1, M'_0 \odot M'_1)} \quad \alpha'_0 + \alpha'_1 \text{ is reduced w.r.t } \alpha + \perp = \perp + \alpha = \alpha \text{ (OR)} \\
\\
\frac{(\beta, M) \xrightarrow{a,p} (\beta', M') \quad \epsilon \in \beta'}{(\alpha^\circledast\beta, M) \xrightarrow{a,p} (\epsilon, M')} \text{ (ITER-END)} \qquad \frac{(\beta, M) \xrightarrow{a,p} (\beta', \_ ) \quad \epsilon \notin \beta' \quad (\alpha, M) \xrightarrow{a,p} (\alpha', M')}{(\alpha^\circledast\beta, M) \xrightarrow{a,p} (\alpha'\alpha^\circledast\beta, M')} \text{ (ITER)} \\
\\
\frac{(\alpha, M) \xrightarrow{a,p} (\alpha', M') \quad \epsilon \notin \alpha'}{(\alpha)_l, M \xrightarrow{a,p} ([\alpha']_l, \{l \mapsto (p, \perp)\})} \text{ (L-START)} \qquad \frac{(\alpha, M) \xrightarrow{a,p} (\alpha', M') \quad \epsilon \notin \alpha'}{([\alpha]_l, M) \xrightarrow{a,p} ([\alpha']_l, M')} \text{ (L-CONT)} \\
\\
\frac{(\alpha, M) \xrightarrow{a,p} (\alpha', M') \quad \epsilon \in \alpha'}{([\alpha]_l, M) \xrightarrow{a,p} (\epsilon, M' \odot \{l \mapsto (p, p)\})} \text{ (L-LTR)} \qquad \frac{(\alpha, M) \xrightarrow{a,p} (\alpha', M') \quad \epsilon \in \alpha'}{([\alpha]_l, M) \xrightarrow{a,p} (\epsilon, M' \odot \{l \mapsto (\perp, p)\})} \text{ (L-END)}
\end{array}$$

**Fig. 2.** One-step relation for evaluating prefix expressions. The rules are evaluated modulo the equalities  $\epsilon \cdot \alpha = \alpha \cdot \epsilon = \alpha$ .

to  $\perp$  are dropped (unless the last one if we should drop all operands). The only non-shortening rule is **ITER** that unrolls  $\alpha$  if  $\beta$  was not matched in  $\alpha^\circledast\beta$ . Thus, evaluating  $\alpha^\circledast\beta$  can first prolong the expression and then eventually end up again in  $\alpha^\circledast\beta$  after a finite number of steps. This does not introduce any problems as the set of derivations remains bounded [16].

There are four rules for handling labellings. Rule **L-LTR** handles one-letter matches. Rule **L-START** handles the beginning of a match where the expression  $(\alpha)_l$  gets rewritten to  $[\alpha]_l$  so that we know we are currently matching label  $l$  in the next steps. Rules **L-CONT** and **L-END** continue and finish the match once the labeled expression evaluates to  $\epsilon$ . Concatenating m-maps in the rules works well because of the assumption that no two sub-expressions have the same label. Therefore, there are no collisions and we always concatenate information from processing the same sub-expression.

The one-step relation is deterministic, i.e., there is always at most one rule that can be applied and thus every PE has a unique single derivation for a fixed letter  $a$  and position  $p$ .



**Theorem 1 (Determinism of  $\xrightarrow{a,p}$ ).** *For an arbitrary PE  $\alpha$  over alphabet  $\Sigma$ , and any  $a \in \Sigma$  and  $p \in \mathbb{N}$ , there exist at most one  $\alpha'$  such that  $\alpha \xrightarrow{a,p} \alpha'$  (that is, there is at most one defining rule of  $\xrightarrow{a,p}$  that can be applied to  $\alpha$ ).*

*Proof (Sketch).* Multiple rules could be applied only if they match the same structure of  $\alpha$  (e.g., that  $\alpha$  is a disjunction). But for such rules, the premises are pairwise unsatisfiable together.

Having the deterministic one-step relation, we can define the *evaluation function* on words  $\delta : \mathbb{E} \times \Sigma^* \rightarrow \mathbb{E} \times \mathbb{M}_L$  that returns the rewritten PE and the m-map resulting from evaluating the one-step relation for each single letter of the word:  $\delta(\alpha, w) = \bar{\delta}(\alpha, w, 0, \emptyset)$  where

$$\bar{\delta}(\alpha, w, p, M) = \begin{cases} (\alpha, M) & \text{if } w = \epsilon \\ \bar{\delta}(\alpha', w', p + 1, M') & \text{if } w = aw' \wedge (\alpha, M) \xrightarrow{a,p} (\alpha', M') \end{cases}$$

We also define the *decomposition function*  $\rho : \mathbb{E} \times \Sigma^* \rightarrow (\Sigma^* \times \mathbb{M}_L \times \Sigma^*) \cup \{\perp\}$  that decomposes a word  $w \in \Sigma^*$  into the matched prefix with the resulting m-map, and the rest of  $w$ :

$$\rho(\alpha, w) = \begin{cases} (u, m, v) & \text{if } w = uv \wedge \delta(\alpha, u) = (\epsilon, m) \\ \perp & \text{otherwise} \end{cases}$$

Function  $\rho$  is well-defined as there is at most one  $u$  for which  $\delta(\alpha, u) = (\epsilon, \_)$ . This follows from the determinism of the one-step relation. Function  $\rho$  is going to be important in the next section.

Before we end this section, let us remark that thanks to the determinism of the one-step relation and the fact that there is only a finite number of derivations of any PE  $\alpha$ , the evaluation function for a fixed PE can be represented as a *finite transducer* [16]. Given the transducer for PE  $\alpha$  and  $w \in \Sigma^*$  s.t.  $(u, m, v) = \rho(\alpha, w)$ ,  $u$  is the prefix of  $w$  that leads the transducer to the accepting state and the transducer outputs a sequence  $m_0, \dots, m_k$  such that  $m_0 \odot \dots \odot m_k = m$ . As a result, we have an efficient and incremental way of evaluating PEs. Moreover, it suggests how to compose and perform other operations with PEs.

### 3 Multi-trace Prefix Expressions and Transducers

In this section, we define *multi-trace prefix expressions* and *multi-trace prefix transducers*.

#### 3.1 Multi-trace Prefix Expressions

A multi-trace prefix expression (MPE) matches prefixes of multiple words. Every MPE consists of prefix expressions associated with input words and a *condition* which is a logical formula that must be satisfied by the matched prefixes.

**Definition 3 (Multi-trace prefix expression).** *Multi-trace prefix expression (MPE) over trace variables  $V_\tau$  and alphabet  $\Sigma$  is a list of pairs together with a formula:*

$$(\tau_0, e_0), \dots, (\tau_k, e_k)[\varphi]$$

where  $\tau_i \in V_\tau$  are trace variables and  $e_i$  are PEs over the alphabet  $\Sigma$ . The formula  $\varphi$  is called the condition of MPE. We require that for all  $i \neq j$ , labels in  $e_i$  and  $e_j$  are distinct.

If the space allows, we typeset MPEs over multiple lines as can be seen in the examples of prefix transducers throughout the paper.

MPE conditions are logical formulae over m-strings and input words<sup>3</sup>. Terms of MPE conditions are  $l_1 = l_2$ ,  $\tau_1[l_1] = \tau_2[l_2]$ , and  $\tau_1[l_1] = w$  for labels or m-string constants  $l_1, l_2$ , trace variables  $\tau_1, \tau_2 \in V_\tau$ , and (constant) words  $w$  over arbitrary alphabet. Labels are evaluated to the m-strings associated with them by a given m-map,  $\tau[l]$  is the concatenation of sub-words of the word associated with  $\tau$  on positions specified by the m-string for  $l$ , and constants evaluate to themselves. For example, if label  $l$  evaluates to  $(0, 1)(3, 4)$  for a given m-map, then  $\tau[l]$  evaluates to *abba* if  $\tau$  is mapped to *abcba*. A well-formed MPE condition is a boolean formula built up from the terms.

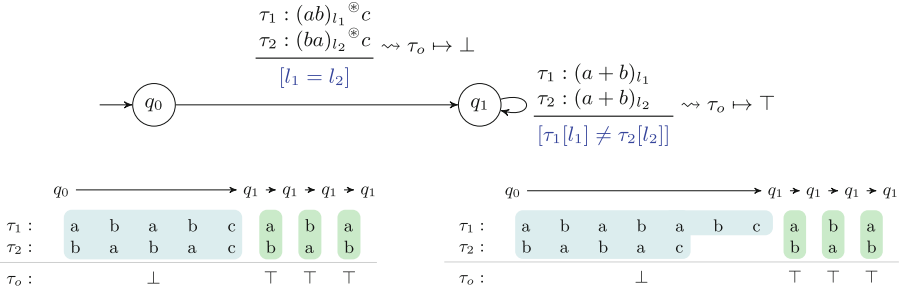
The satisfaction relation of MPE conditions is defined w.r.t an m-map  $M$ , and a *trace assignment*  $\sigma : V_\tau \rightarrow \Sigma^*$  which is a map from a set of trace variables  $V_\tau$  into words. We write  $\sigma, M \models \varphi$  when  $\sigma$  and  $M$  satisfy condition  $\varphi$ . Because of the space restrictions, we refer to the extended version of this paper [16] for formal definition of the satisfaction relation for MPE conditions. Nonetheless, we believe that MPE conditions are intuitive enough from the examples.

Given an MPE  $\alpha$ , we denote as  $\alpha(\tau)$  the PE associated with trace variable  $\tau$  and if we want to highlight that  $\alpha$  has the condition  $\varphi$ , we write  $\alpha[\varphi]$ . We denote the set of all MPEs over trace variables  $V_\tau$  and alphabet  $\Sigma$  as  $MPE(V_\tau, \Sigma)$ . An MPE  $\alpha[\varphi]$  over trace variables  $V_I = \{\tau_1, \dots, \tau_k\}$  satisfies a trace assignment  $\sigma$ , written  $\sigma \models \alpha$ , iff  $\forall \tau \in V_I : \rho(\alpha(\tau), \sigma(\tau)) = (u_\tau, M_\tau, v_\tau)$  and  $\sigma, M_{\tau_1} \odot \dots \odot M_{\tau_k} \models \varphi$ . That is,  $\sigma \models \alpha$  if every prefix expression of  $\alpha$  has matched a prefix on its trace and the condition  $\varphi$  is satisfied.

### 3.2 Multi-trace Prefix Transducers

*Multi-trace prefix transducers (MPT)* are finite transducers [36] with MPEs as guards on the transitions. If a transition is taken, one or more symbols are appended to one or more output words, the state is changed to the target state of the transition and the matched prefixes of input words are consumed. The evaluation then continues matching new prefixes of the shortened input words.

<sup>3</sup> The conditions can be almost arbitrary formulae, depending on how much we are willing to pay for their evaluation. They could even contain nested MPEs. As we will see later, MPE conditions are evaluated only after matching the prefixes which gives us a lot of freedom in choosing the logic. For the purposes of this work, however, we use only simple conditions that we need in our examples and evaluation.



**Fig. 3.** The MPT from Example 1 and a demonstration of its two runs. Colored regions show parts of words as they are matched by the transitions, the sequence of passed states is shown above the traces.

Combining MPEs with finite state transducers allows to read input words asynchronously (evaluating MPEs) while having synchronization points and finite memory (states of the transducer).

**Definition 4 (Multi-trace prefix transducer).** A multi-trace prefix expression transducer (MPT) is a tuple  $(V_I, V_O, \Sigma_I, \Sigma_O, Q, q_0, \Delta)$  where

- $V_I$  is a finite set of input trace variables
- $V_O$  is a finite set of output trace variables
- $\Sigma_I$  is an input alphabet
- $\Sigma_O$  is an output alphabet
- $Q$  is a finite non-empty set of states
- $q_0 \in Q$  is the initial state
- $\Delta : Q \times MPE(V_I, \Sigma_I) \times (V_O \hookrightarrow \Sigma_O^*) \times Q$  is the transition relation; we call the partial mappings  $(V_O \hookrightarrow \Sigma_O^*)$  output assignments.

A run of an MPT  $(V_I, V_O, \Sigma_I, \Sigma_O, L, Q, q_0, \Delta)$  on trace assignment  $\sigma_0$  is a sequence  $\pi = (q_0, \sigma_0) \xrightarrow{\nu_0} (q_1, \sigma_1) \xrightarrow{\nu_1} \dots \xrightarrow{\nu_{k-1}} (q_k, \sigma_k)$  of alternating states and trace assignments  $(q_i, \sigma_i)$  with output assignments  $\nu_i$ , such that for each  $(q_i, \sigma_i) \xrightarrow{\nu_i} (q_{i+1}, \sigma_{i+1})$  there is a transition  $(q_i, \alpha, \nu_i, q_{i+1}) \in \Delta$  such that  $\sigma_i \models \alpha$  and  $\forall \tau \in V_I : \sigma_{i+1}(\tau) = v_\tau$  where  $(\_, \_, v_\tau) = \rho(\alpha(\tau), \sigma_i(\tau))$ . That is, taking a transition in an MPT is conditioned by satisfying its MPE and its effect is that every matched prefix is removed from its word and the output assignment is put to output.

The output  $O(\pi)$  of the run  $\pi$  is the concatenation of the output assignments  $\nu_0 \cdot \nu_1 \cdot \dots \cdot \nu_{k-1}$ , where a missing assignment to a trace is considered to be  $\epsilon$ . Formally, for any  $t \in V_O$ ,  $\nu_i \cdot \nu_j$  takes the value

$$(\nu_i \cdot \nu_j)(t) = \begin{cases} \nu_i(t) \cdot \nu_j(t) & \text{if } \nu_i(t) \text{ and } \nu_j(t) \text{ are defined} \\ \nu_i(t) & \text{if } \nu_i(t) \text{ is defined and } \nu_j(t) \text{ is undefined} \\ \nu_j(t) & \text{if } \nu_i(t) \text{ is undefined and } \nu_j(t) \text{ is defined} \end{cases}$$

*Example 1.* Consider the MPT in Fig. 3 and words  $t_1 = ababcaba$  and  $t_2 = babacbab$  and the assignment  $\sigma = \{\tau_1 \mapsto t_1, \tau_2 \mapsto t_2\}$ . The run on this assignment is depicted in the same figure on the bottom left. The output of the MPT on  $\sigma$  is  $\perp \top \top \top$ . For any words that are entirely consumed by the MPT without getting stuck, it holds that  $\tau_1$  starts with a sequence of  $ab$ 's and  $\tau_2$  with a sequence of  $ba$ 's of the same length. Then there is one  $c$  on both words and the words end with a sequence of  $a$  or  $b$  but such that when there is  $a$  in one word, there must be  $b$  in the other word and vice versa.

Now assume that the words are  $t_1 = abababcaba$  and  $t_2 = babacbab$  with the same assignment. The situation changes as now the expression on trace  $t_1$  matches the prefix  $(ab)^3$  while on  $t_2$  the prefix  $(ba)^2$ . Thus  $l_1 = (0, 6) \neq (0, 4) = l_2$  and the match fails. Finally, assume that we remove the condition  $[l_1 = l_2]$  from the first transition. Then for the new words the MPT matches again and the match is depicted on the bottom right in Fig. 3.

In the next section, we work with *deterministic* MPTs. We say that an MPT is deterministic if it can take at most one transition in any situation.

**Definition 5 (Deterministic MPT).** Let  $T = (V_I, V_O, \Sigma_I, \Sigma_O, Q, q_0, \Delta)$  be an MPT. We say that  $T$  is *deterministic (DMPT)* if for any state  $q \in Q$ , and an arbitrary trace assignment  $\sigma : V_I \rightarrow \Sigma_I^*$ , if there are multiple transitions  $(q, \alpha_1, \nu_1, q_1), \dots, (q, \alpha_k, \nu_k, q_k)$  such that  $\forall i : \sigma \models \alpha_i$ , it holds that there exists a proper prefix  $\eta$  of  $\sigma$ , (i.e.,  $\forall \tau \in V_I : \eta(\tau) \leq \sigma(\tau)$  and for some  $\tau' it holds that  $\eta(\tau') < \sigma(\tau')$ ), and there exist  $i$  such that  $\eta \models \alpha_i$  and  $\forall j \neq i : \eta \not\models \alpha_j$ .$

Intuitively, an MPT is DMPT if whenever there is a trace assignment that satisfies more than one transition from a state, one of the transitions matches “earlier” than any other of those transitions.

## 4 Hypertrace Transformations

A *hyperproperty* is a set of sets of infinite traces. In this section, we discuss an algorithm for monitoring *k-safety hyperproperties*, which are those whose violation can be witnessed by at most  $k$  finite traces:

**Definition 6 (*k-safety hyperproperty*).** A hyperproperty  $S$  is *k-safety hyperproperty* iff

$$\forall T \subseteq \Sigma^\omega : T \notin S \implies \exists M \subseteq \Sigma^* : M \leq T \wedge |M| \leq k \wedge (\forall T' \subseteq \Sigma^\omega : M \leq T' \implies T' \notin S)$$

where  $\Sigma^\omega$  is the set of infinite words over alphabet  $\Sigma$ , and  $M \leq T$  means that each word in  $M$  is a (finite) prefix of a word in  $T$ .

We assume *unbounded parallel input model* [25], where there may be arbitrary many traces digested in parallel, and new traces may be announced at any time. Our algorithm is basically the combinatorial algorithm for monitoring hyperproperties of Finkbeiner et al. [23, 28] where we exchange automata generated from HyperLTL specifications with MPTs. That is, to monitor a  $k$ -safety hyperproperty, we instantiate an MPT for every  $k$ -tuple of input traces. An advantage of using MPTs instead of monitor automata in the algorithm of

Finkbeiner et al. is that we automatically get a monitoring solution for asynchronous hyperproperties. A disadvantage is that we cannot automatically use some of the optimizations designed for HyperLTL monitors that are based on the structure (e.g., symmetry) of the HyperLTL formula [24].

The presented version of our algorithm assumes that the input is DMPT as it is preferable to have deterministic monitors. Deciding whether a given MPT is deterministic depends a lot on the chosen logic used for MPE constraints. In the rest of the paper, we assume that the used MPTs are *known* to be DMPTs (which is also the case of MPTs used in our evaluation). We make the remark that in cases where the input MPT is not known to be deterministic and/or a check is impractical, one may resort to a way how to resolve possible non-determinism instead, such as using priorities on the edges. This is a completely valid solution and it is easy to modify our algorithm to work this way. In fact, the algorithm works also with non-deterministic MPTs with a small modification.

#### 4.1 Algorithm for Online Monitoring of $k$ -safety Hyperproperties

Our algorithm is depicted in Algorithm 1 and Algorithm 2 (auxiliary procedures). In essence, the algorithm maintains a set of *configurations* where one configuration corresponds to the state of evaluation of one edge of an DMPT instance. Whenever the algorithm may make a progress in some configuration, it does so and acts according to whether matching the edge succeeds, fails, or needs more events.

Now we discuss functioning of the algorithm in more detail. The input is an DMPT  $(\{V_I, \{\tau_O\}, \Sigma_I, \{\perp, \top\}, Q, q_0, \Delta\})$ . W.l.o.g we assume that  $V_I = \{\tau_1, \dots, \tau_k\}$ . The DMPT outputs a sequence of verdicts  $\{\top, \perp\}^*$ . A violation of the property is represented by  $\perp$ , so whenever  $\perp$  appears on the output, the algorithm terminates and reports the violation.

A configuration is a 4-tuple  $(\sigma, (p_1, \dots, p_k), M, e)$  where  $\sigma$  is a function that maps trace variables to traces, the vector  $(p_1, \dots, p_k)$  keeps track of reading positions in the input traces,  $M$  is the current m-map gathered while evaluating the MPE of  $e$ , and  $e$  is the edge that is being evaluated. More precisely,  $e$  is the edge that still needs to be evaluated in the future as its MPE gets repeatedly rewritten during the run of the algorithm. If the edge has MPE  $E$ , we write  $E[\tau \mapsto \xi]$  for the MPE created from  $E$  by setting the PE for  $\tau$  to  $\xi$ .

The algorithm uses three global variables. Variable *workbag* stores configurations to be processed. Variable *traces* is a map that remembers the so-far-seen contents of all traces. Whenever a new event arrives on a trace  $t$ , we append it to  $traces(t)$ . Traces on which a new event may still arrive are stored in variable *onlinetraces*. Note that to follow the spirit of online monitoring setup, in this section, we treat traces as *opaque* objects that we query for next events.

In each iteration of the main loop (line 5), the algorithm first calls the procedure `update_traces` (line 6, the procedure is defined in Algorithm 2). This procedure adds new traces to *onlinetraces* and updates *workbag* with new configurations if there are any new traces, and extends traces in *traces* with new events. The core of the algorithm are lines 9–39 that take all configuration sets and update them with unprocessed events.

---

**Algorithm 1:** Online algorithm for monitoring hyperproperties with MPTs
 

---

**Input:** an DMPT  $(\{\{\tau_1, \dots, \tau_k\}, \{\tau_O\}, \Sigma_I, \{\perp, \top\}, Q, q_0, \Delta\})$ 
**Output:** *false* + witness if an DMPT instance outputs  $\perp$ , *true* if no DMPT instance outputs  $\perp$  and there are finitely many traces. The algorithm does not terminate otherwise.

```

1  traces  $\leftarrow \emptyset$  // Stored contents of all traces
2  onlinetraces  $\leftarrow \emptyset$  // Traces that are still being extended
3  workbag  $\leftarrow \emptyset$  // Sets of configurations to process
4
5  while true do
6    update_traces (workbag, onlinetraces, traces)
7    workbag'  $\leftarrow \emptyset$  // The new contents of workbag
8
9    foreach  $C \in$  workbag do
10      $C' \leftarrow \emptyset$  // The rewritten set of configurations
11
12     // Try to move each configuration in the set of configurations
13     foreach  $c = (\sigma, (p_1, \dots, p_k), M, q \xrightarrow{E[\varphi] \rightsquigarrow \nu} q') \in C$  do
14        $E', M' \leftarrow E, M$ 
15        $(p'_1, \dots, p'_k) \leftarrow (p_1, \dots, p_k)$ 
16       // Progress on each trace where possible
17       foreach  $1 \leq i \leq k$  s.t.  $p_i < |\text{traces}(\sigma(\tau_i))| \wedge E(\tau_i) \neq \epsilon$  do
18          $E' \leftarrow E'[\tau_i \mapsto \xi]$  where  $E(\tau_i), M' \xrightarrow{\text{traces}(\sigma(\tau_i))[p_i], p_i} \xi, M''$ 
19          $M' \leftarrow M''$ 
20          $p'_i \leftarrow p'_i + 1$ 
21         if  $\xi = \perp$  then // Configuration failed
22           continue with next configuration (line 13)
23         if  $\forall j. E'(\tau_j) = \epsilon$  then // All prefix expressions matched
24           if  $\sigma, M' \models \varphi$  then // The condition is satisfied
25             // Compare  $p'_1, \dots, p'_k$  against positions in other
26             // configurations from this set to see if this must be
27             // the shortest match
28             if  $\perp \in \nu$  then // Violation found
29               return false +  $\sigma$ 
30             // Edge is matched, no violation found, queue
31             // successor edges
32             workbag'  $\leftarrow$ 
33             workbag'  $\cup \{\text{cfigs}(q', (\sigma(\tau_1), \dots, \sigma(\tau_k)), (p'_1, \dots, p'_k))\}$ 
34             // This set of configurations is done
35             continue outer-most loop (line 9)
36           else
37             continue with next configuration (line 13)
38         // If the configuration has matched or it can still make a
39         // progress, put it back (modified) to the set
40         if  $E'$  has matched or
41          $\neg (\forall 1 \leq i \leq k : \sigma(\tau_i) \notin \text{onlinetraces} \wedge p_i = |\text{traces}(\sigma(\tau_i))|)$  then
42            $C' \leftarrow C' \cup \{(\sigma, (p'_1, \dots, p'_k), M', q \xrightarrow{E'[\varphi] \rightsquigarrow \nu} q')\}$ 
43         if  $C' \neq \emptyset$  then
44           workbag'  $\leftarrow$  workbag'  $\cup \{C'\}$  // Queue the modified set of
45           configurations
46
47     workbag  $\leftarrow$  workbag'
48     if workbag =  $\emptyset$  and no new trace will appear then
49       return true

```

---

**Algorithm 2:** Auxiliary procedures for Algorithm 1

---

```

1 // Auxiliary procedure that returns a set of configurations for
  outgoing edges of  $q$ 
2 Procedure  $\text{cfigs}(q, (t_1, \dots, t_k), (p_1, \dots, p_k))$ 
3    $\sigma \leftarrow \{\tau_i \mapsto t_i \mid 1 \leq i \leq k\}$ 
4   return  $\{(\sigma, (p_1, \dots, p_k), (0, \dots, 0), \emptyset, e) \mid e \text{ is an outgoing edge from } q\}$ 
5
6 // Auxiliary procedure to add new traces and update the current
  ones
7 Procedure  $\text{update\_traces}(\text{workbag}, \text{onlinetraces}, \text{traces})$ 
8   if there is a new trace  $t$  then // Update traces and workbag with the
  new trace
9      $\text{onlinetraces} \leftarrow \text{onlinetraces} \cup \{t\}$ 
10     $\text{traces} \leftarrow \text{traces}[t \mapsto \epsilon]$ 
11     $\text{tuples} \leftarrow \{(t_1, \dots, t_k) \mid t_j \in \text{Dom}(\text{traces}), t = t_i \text{ for some } i\}$ 
12     $\text{workbag} \leftarrow \text{workbag} \cup \{\text{cfigs}(q_0, (t_1, \dots, t_k), (0, \dots, 0)) \mid (t_1, \dots, t_k) \in$ 
  tuples $\}$ 
13
14  foreach  $t \in \text{onlinetraces}$  that has a new event  $e$  do // Update traces
  with new events
15     $\text{traces}(t) = \text{traces}(t) \cdot e$ 
16    if  $e$  was the last event on  $t$  then // Remove finished traces from
  onlinetraces
17    |  $\text{onlinetraces} \leftarrow \text{onlinetraces} \setminus \{t\}$ 

```

---

The algorithm goes over every set of configurations from *workbag* (line 9) and attempts to make a progress on every configuration in the set (line 13). For each trace where a progress can be made in the current configuration (line 17), i.e., there is an unprocessed event on the trace  $\tau_i$  ( $p_i < |\text{traces}(\sigma(\tau_i))|$ ), and the corresponding PE on the edge still has not matched ( $E(\tau_i) \neq \epsilon$ ), we do a step on this PE (line 18). The new state of the configuration is aggregated into the primed temporary variables ( $E', M', \dots$ ). If the MPE matches (lines 23 and 24), we check if other configurations from the set have progressed enough for us to be sure that this configuration has matched the shortest prefix (line 26). That is, we compare  $p'_1, \dots, p'_k$  against positions  $p''_1, \dots, p''_k$  from each other configuration in  $C$  if it is strictly smaller (i.e.,  $p'_i \leq p''_i$  for all  $i$  and there is  $j$  s.t.,  $p'_j < p''_j$ ). If this is true, we can be sure that there is no other edge that can match a shorter prefix and that has not matched it yet because it was waiting for events. If this configuration is the shortest match, the output of the edge is checked if it contains  $\perp$  (line 27) and if so, *false* with the counterexample is returned on line 28 because the monitored property is violated. Else, the code falls-through to line 30 that queues new configurations for successor edges as the current edge has been successfully matched and then continues with a new iteration of the outer-most loop (line 32). The continue statement has the effect that all other configurations derived from the same state (other edges) are dropped and therefore progress is made only on the configuration (edge) that matched. If any progress on the MPE can be made in the future, or it has already matched but we do not know if it is the shortest match yet,

the modified configuration is pushed into the set of configurations instead of the original one (line 37). If not all the configurations from  $C$  were dropped because they could not proceed, line 39 pushes the modified set of configurations back to *workbag* and a new iteration starts.

## 4.2 Discussion

To see that the algorithm is correct, let us follow the evolution of the set of configurations for a single instance of the DMPT on traces  $t_1, \dots, t_k$ . The initial set of configurations corresponding to outgoing edges from the initial state is created and put to *workbag* exactly once on line 12 in Algorithm 2. When it is later taken from *workbag* on line 9 (we are back in Algorithm 1), every configuration (edge) is updated – a step is taken on every PE from the edge’s MPE (lines 17–18) where a step can be made. If matching the MPE fails, the configuration is discarded due to the jump on line 22 or line 34. If matching the MPE has neither failed nor succeeded (and no violation has been found, in which case the algorithm would immediately terminate), the updated configuration is pushed back to *workbag* and revisited in later iterations. If the MPE has been successfully matched and it is not known to be the shortest match, it is put back to *workbag* and revisited later when other configurations may have proceeded and we may again check if it is the shortest match or not. If it is the shortest match, its successor edges are queued to *workbag* on line 30 (if no violation is found). Note that the check for the shortest match may fail because of some configuration that has failed in the current iteration but is still in  $C$ . Such configurations, however, will get discarded in the current iteration and in the next iteration the shortest match is checked again without these. This way we incrementally first match the first edge on the run of the DMPT (or find out that no edge matches), then the second edge after it gets queued into *workbag* on line 30, and so on.

The algorithm terminates if the number of traces is bounded. If it has not terminated because of finding a violation on line 28, it will terminate on line 43. To see that the condition on line 42 will eventually get true if the number of traces is bounded, it is enough to realize that unless a configuration gets matched or failed, it is discarded at latest when failing the condition on line 36 after reading entirely (finite) input traces. Otherwise, if a configuration fails, the set is never put back to *workbag* and if it gets matched, it can get back to *workbag* repeatedly only until the shortest match is identified. But if every event comes in finite time, some of the configurations in the set will eventually be identified as the shortest match (because the MPT is deterministic), and the set of configurations will be done. Therefore, *workbag* will eventually become empty.

Worth remark is that if we give up on checking if the matched MPE is the shortest match on line 26 (we set the condition to *true*) and on line 32, we continue with the loop on line 13 instead of with the outer-most loop, i.e., we do not discard the set of configurations upon a successfully taken edge, the algorithm will work also for generic *non-deterministic* MPTs.

Even though this algorithm is very close to the algorithm of Finkbeiner et al. [23, 25, 28] where we replace monitoring automata with prefix transducers, there is an important difference. In our algorithm, we assume

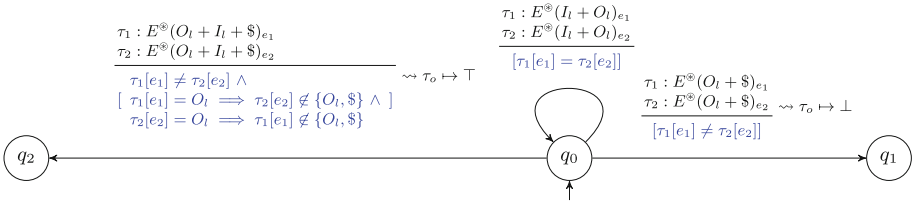


that existing traces may be extended at any time until the last event has been seen. This is also the reason why we need the explicit check whether a matched configuration is the shortest match. The algorithm of Finkbeiner et al. assumes that when a new trace appears, its contents is entirely known. So their algorithm is incremental on the level of traces, while our algorithm is incremental on the level of traces *and* events.

The monitor templates in the algorithm of Finkbeiner et al. are automata whose edges match propositions on different traces. Therefore, they can be seen as trivial DMPTs where each prefix expression is a single letter or  $\epsilon$ . Realizing this, we could say that our monitoring algorithm is an asynchronous extension of the algorithm of Finkbeiner et al. where we allow to read multiple letters on edges, or, alternatively, that in the context of monitoring hyperproperties, DMPTs are a generalization of HyperLTL template automata to asynchronous settings.

## 5 Empirical Evaluation

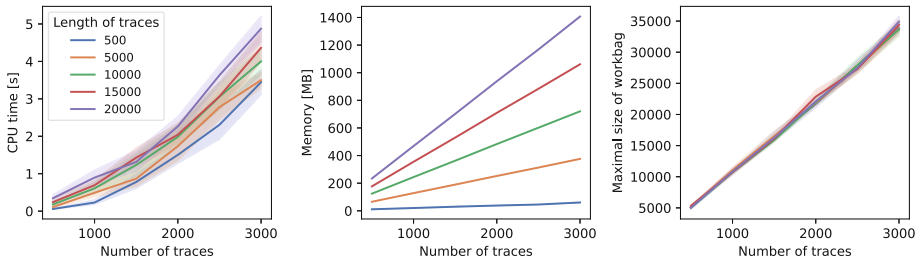
We conducted a set of experiments about monitoring asynchronous version of OD on random and semi-random traces. The traces contain input and output events  $I(\mathbf{t}, \mathbf{n})$  and  $O(\mathbf{t}, \mathbf{n})$  with  $\mathbf{t} \in \{1, h\}$ , and  $\mathbf{n}$  a 64-bit unsigned number. Further, a trace can contain the event  $E$  without parameters that abstracts any event that have occurred in the system, but that is irrelevant to OD.



**Fig. 4.** The DMPT used for monitoring asynchronous OD in the experiments.

The DMPT used for monitoring OD is a modified version of the DMPT for monitoring OD from Sect. 1, and is shown in Fig. 4. The modification makes the DMPT handle also traces with different number and order of input and output events. The letter  $\$$  represents the end of trace and is automatically appended to the input traces. We abuse the notation and write  $\tau_1[e_1] = O_l$  for the expression that would be formally a disjunction comparing  $\tau_1[e_1]$  to all possible constants represented by  $O_l$ . However, in the implementation, this is a simple constant-time check of the type of the event, identical to checking that an event matches  $O_l$  when evaluating prefix expressions. The term  $\tau_i[e_i] \notin \{O_l, \$\}$  is just a shortcut for  $\tau_i[e_i] \neq O_l \wedge \tau_i[e_i] \neq \$$ .

The self-loop transition in the DMPT in Fig. 4 has no output and we enabled the algorithm to stop processing traces whenever  $\top$  is detected on the output of the transducer because that means that OD holds for the input traces. Also, we used the reduction of traces [24] – because OD is symmetric and reflexive, then



**Fig. 5.** CPU time and maximal memory consumption of monitoring asynchronous OD on random traces with approx. 10% of low input events and 10% of low output events. Values are the average of 10 runs.

if we evaluate it on the tuple of traces  $(t_1, t_2)$ , we do not have to evaluate it for  $(t_2, t_1)$  and  $(t_i, t_i)$ .

Monitors were implemented in C++ with the help of the framework VAMOS [17]. The experiments were run on a machine with *AMD EPYC* CPU with the frequency 3.1 GHz. An artifact with the implementation of the algorithm and scripts to reproduce the experiments can be found on Zenodo<sup>4</sup>.

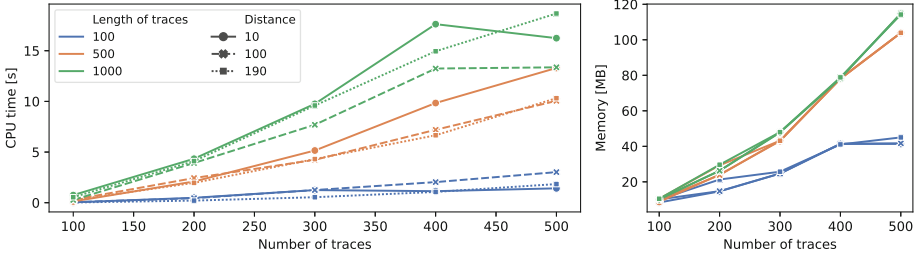
**Experiments on Random Traces.** In this experiment, input traces of different lengths were generated such that approx. 10% were low input and 10% low output events. These events always carried the value 0 or 1 to increase the chance that some traces coincide on inputs and outputs.

Results of this experiment are depicted in Fig. 5. The left plot shows that the monitor is capable of processing hundreds of traces in a short time and seem to scale well with the number of traces, irrespective of the length of traces. The memory consumption is depending more on the length of traces as shown in the middle plot. This is expected as all the input traces are stored in memory. Finally, the maximal size of the workbag grows linearly with the number of traces but not with the length of traces, as the right plot shows.

**Experiments on Periodic Traces.** In this experiment, we generated a single trace that contains low input and output events periodically with fixed distances. Multiple instances of this trace were used as the input traces. The goal of the experiment is to see how the monitor performs on traces that must be processed to the very end and if the performance is affected by the layout of events.

The plots in Fig. 6 show that the monitor scales worse than on random traces as it has to always process the traces to their end. For the same reason, the performance of the monitor depends more on the length of the traces. Still, it can process hundreds of traces in a reasonable time. The data do not provide a clear hint on how the distances between events change the runtime, but they do not affect it significantly. The memory consumption remains unaffected by distances.

<sup>4</sup> <https://doi.org/10.5281/zenodo.8191723>.



**Fig. 6.** The plot shows CPU time and memory consumption of monitoring asynchronous OD on instances of the same trace with low input and output events laid out periodically with fixed distances.

## 6 Related Work

In this section, we review the most closely related work. More exhaustive review can be found in the extended version of this paper [16].

**Logics for Hyperproperties.** Logics for hyperproperties are typically created by extending a logic for trace properties. Hyperlogics *HyperLTL* [19] and *HyperCTL\** [19] extend *LTL* and *CTL\**, resp., with explicit quantification over traces. The logic *FO[<,E]* [26] and *SIS[E]* [21] are first- and second- order logics with successors extended with the *equal level predicate* that relates the same time points on different traces.

All of the hitherto mentioned logics use *synchronous time model*. *Asynchronous HyperLTL* [9], *Stuttering HyperLTL* [12], and *Context HyperLTL* [12] are extensions of *HyperLTL* to asynchronous time model. Gutsfeld et al. [27] introduce *Multi-tape Alternating Asynchronous Word Automata (AAWA)* and the temporal fixpoint calculus  $H_\mu$  for the specification and analysis of asynchronous hyperproperties. AAAs are so far the only automata-based formalism for specification of asynchronous hyperproperties. Beutner et al. define *HyperATL\** [11], an extension of the logic *ATL\** [4] that can capture asynchronous hyperproperties via quantification over *strategies* of a scheduler. *Hypernode automata* [8] introduced by Bartocci et al. combine finite-state automata with the *hypertrace logic* which allows to describe properties that have multiple „phases”. The hypertrace logic ignores stuttering and prefixing to enable asynchronous time model.

**Runtime Monitoring of Hyperproperties.** The first paper on runtime monitoring of hyperproperties is due to Agrawal and Bonakdarpur [3]. They consider monitoring  $k$ -safety hyperproperties specified with *HyperLTL*. In general, monitoring algorithms for hyperproperties can be classified as combinatorial or constraint-based [28]. Combinatorial algorithms [23,24,28] construct multiple instances of an monitoring automaton and therefore our algorithm fall into this category. Constraint-based algorithms [2,13,28,29] translate the monitoring task into a set of constraints (e.g., SMT formulae) and apply rewriting and solving of the constraints to monitor a given hyperproperty.

*Stream runtime verification (SRV)* [35] specifies monitoring as transformation of streams of data, which makes SRV also related to transducers. It is common that there are multiple input and output streams in an SRV specification, and languages like TeSSLa [32] support asynchronous time model. So far as we know, no one has used SRV languages in the context of hyperproperties yet.

**Automata and Regular Expressions.** *Automata* and *transducers* [36] are the basis of MPTs and are well explored. *Multi-track automata* [15] are automata that read  $n$ -tuples of letters. They commonly use also a special letter  $\lambda$  for a gap (no letter) and thus can describe asynchronous reading of words. *Regular expressions (RE)* are an ubiquitous formalism with many uses and many restrictions/extensions. *Prefixed regular expressions (PRE)* [6] are a subset of REs with some properties similar to PEs. Semantically, *prefix-free* REs [30] are closer to PEs than PREs, because PEs give raise to prefix-free languages as follows from [31, Lemma 1] and the fact that a PE corresponds to a prefix-free transducer [16]. REs with the *shortest-match semantics* [18] are very close to PEs, however, unlike PEs, they can be ambiguous. MPTs with a single input word could be seen as a modification of *expression automata* [31], which are automata with REs on edges.

*Backreferences* in regular expressions refer to parts of the word that were already matched [34]. They can be even named [10], raising more similarities to our labels. In REs, backreferences bring a great power as they allow non-regular languages to be matched [10]. PEs can also recognize some non-regular patterns, however, labels are much weaker than backreferences because MPE constraints are evaluated only a posteriori, while backreferences modify the way how REs are matched.

## 7 Conclusion and Future Work

We introduced prefix expressions and multi-trace prefix transducers, a formalism that we see as a natural executable specification for the monitoring of synchronous and asynchronous hyperproperties. Prefix expressions are similar to regular expressions, but match only prefixes of words. The reason why we prefer prefix expressions over regular expressions (that could also be used to match prefixes) is that our prefix expressions are deterministic and unambiguous. These properties make evaluating prefix expressions efficient. The matched prefixes, more precisely their parts that were explicitly labeled, can be then reasoned about using logical formulae, which are a part of multi-trace prefix expressions that extend prefix expressions to multiple words. Multi-trace prefix expressions are used as guards on edges in multi-trace prefix transducers, which incrementally match and consume prefixes of input words and transform them into output words. Combining prefix expressions with finite state transducers allows us to read input words asynchronously (matching prefix expressions) with synchronisation points (states of the transducer).

We use prefix transducers to monitor synchronous and asynchronous  $k$ -safety hyperproperties. Our experimental evaluation of monitoring asynchronous obser-

vational determinism shows that a prefix-transducer-based monitoring algorithm can scale to thousands of traces.

Prefix transducers provide a flexible formalism for optimizing monitoring algorithms. We currently implement an asynchronous monitoring algorithm that uses prefix transducers to summarize the seen traces, similar to the constraint-based algorithms for monitoring synchronous hyperproperties [28]. We also want to analyze the transducers to avoid instantiating them on redundant tuples of traces, similar to the optimizations for HyperLTL monitors [24]. Furthermore, the evaluation of prefix transducers provides many opportunities for parallelization, ranging from parallelizing the workbag in Algorithm 1 to evaluating prefix expressions for different traces in parallel. Finally, we work on compiling prefix transducers from a high-level logical specification languages for asynchronous hyperproperties, namely [8]. All our implementation work is carried out to extend the VAMOS [17] software infrastructure for monitoring.

**Acknowledgements.** This work was supported in part by the ERC-2020-AdG 101020093. The authors would like to thank Ana Oliveira da Costa for commenting on a draft of the paper.

## References

1. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification - Introductory and Advanced Topics. LNCS, vol. 10457. Springer, Cham (2018). <https://doi.org/10.1007/978-3-319-75632-5>
2. Aceto, L., Achilleos, A., Anastasiadi, E., Francalanza, A.: Monitoring hyperproperties with circuits. In: Mousavi, M.R., Philippou, A. (eds.) FORTE 2022. LNCS, vol. 13273, pp. 1–10. Springer, Cham (2022). [https://doi.org/10.1007/978-3-031-08679-3\\_1](https://doi.org/10.1007/978-3-031-08679-3_1)
3. Agrawal, S., Bonakdarpour, B.: Runtime verification of  $k$ -safety hyperproperties in HyperLTL. In: CSF 2016, pp. 239–252. IEEE (2016). <https://doi.org/10.1109/CSF.2016.24>
4. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. *J. ACM* **49**(5), 672–713 (2002). <https://doi.org/10.1145/585265.585270>
5. Antimirov, V.M.: Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.* **155**(2), 291–319 (1996). [https://doi.org/10.1016/0304-3975\(95\)00182-4](https://doi.org/10.1016/0304-3975(95)00182-4)
6. Baeza-Yates, R.A., Gonnet, G.H.: Fast text searching for regular expressions or automaton searching on tries. *J. ACM* **43**(6), 915–936 (1996). <https://doi.org/10.1145/235809.235810>
7. Bartocci, E., Ferrère, T., Henzinger, T.A., Nickovic, D., da Costa, A.O.: Flavors of sequential information flow. In: Finkbeiner, B., Wies, T. (eds.) VMCAI 2022. LNCS, vol. 13182, pp. 1–19. Springer, Cham (2022). [https://doi.org/10.1007/978-3-030-94583-1\\_1](https://doi.org/10.1007/978-3-030-94583-1_1)
8. Bartocci, E., Henzinger, T.A., Nickovic, D., da Costa, A.O.: Hypernode automata (2023). <https://doi.org/10.48550/arXiv.2305.02836>
9. Baumeister, J., Coenen, N., Bonakdarpour, B., Finkbeiner, B., Sánchez, C.: A temporal logic for asynchronous hyperproperties. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 694–717. Springer, Cham (2021). [https://doi.org/10.1007/978-3-030-81685-8\\_33](https://doi.org/10.1007/978-3-030-81685-8_33)

10. Berglund, M., van der Merwe, B.: Regular expressions with backreferences re-examined. In: Stringology Conference 2017, pp. 30–41. Czech Technical University in Prague (2017). <http://www.stringology.org/event/2017/p04.html>
11. Beutner, R., Finkbeiner, B.: A temporal logic for strategic hyperproperties. In: CONCUR 2021. LIPIcs, vol. 203, pp. 24:1–24:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). <https://doi.org/10.4230/LIPIcs.CONCUR.2021.24>
12. Bozzelli, L., Peron, A., Sánchez, C.: Asynchronous extensions of HyperLTL. In: LICS 2021, pp. 1–13. IEEE (2021). <https://doi.org/10.1109/LICS52264.2021.9470583>
13. Brett, N., Siddique, U., Bonakdarpour, B.: Rewriting-based runtime verification for alternation-free HyperLTL. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 77–93. Springer, Heidelberg (2017). [https://doi.org/10.1007/978-3-662-54580-5\\_5](https://doi.org/10.1007/978-3-662-54580-5_5)
14. Brzozowski, J.A.: Derivatives of regular expressions. *J. ACM* **11**(4), 481–494 (1964). <https://doi.org/10.1145/321239.321249>
15. Bultan, T., Yu, F., Alkhalaf, M., Aydin, A.: Relational string analysis. In: Bultan, T., Yu, F., Alkhalaf, M., Aydin, A. (eds.) String Analysis for Software Verification and Security, pp. 57–68. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-68670-7\\_5](https://doi.org/10.1007/978-3-319-68670-7_5)
16. Chalupa, M., Henzinger, T.A.: Monitoring hyperproperties with prefix transducers (2023). <https://doi.org/10.48550/arXiv.2308.03626>
17. Chalupa, M., Muehlboeck, F., Lei, S.M., Henzinger, T.A.: VAMOS: middleware for best-effort third-party monitoring. In: Lambers, L., Uchitel, S. (eds.) FASE 2023. LNCS, vol. 13991, pp. 260–281. Springer, Cham (2023). [https://doi.org/10.1007/978-3-031-30826-0\\_15](https://doi.org/10.1007/978-3-031-30826-0_15)
18. Clarke, C.L.A., Cormack, G.V.: On the use of regular expressions for searching text. *ACM Trans. Program. Lang. Syst.* **19**(3), 413–426 (1997). <https://doi.org/10.1145/256167.256174>
19. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 265–284. Springer, Heidelberg (2014). [https://doi.org/10.1007/978-3-642-54792-8\\_15](https://doi.org/10.1007/978-3-642-54792-8_15)
20. Clarkson, M.R., Schneider, F.B.: Hyperproperties. *J. Comput. Secur.* **18**(6), 1157–1210 (2010). <https://doi.org/10.3233/JCS-2009-0393>
21. Coenen, N., Finkbeiner, B., Hahn, C., Hofmann, J.: The hierarchy of hyperlogics. In: LICS 2019, pp. 1–13. IEEE (2019). <https://doi.org/10.1109/LICS.2019.8785713>
22. Finkbeiner, B., Haas, L., Torfah, H.: Canonical representations of k-safety hyperproperties. In: CSF 2019, pp. 17–31. IEEE (2019). <https://doi.org/10.1109/CSF.2019.00009>
23. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: Monitoring hyperproperties. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 190–207. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-67531-2\\_12](https://doi.org/10.1007/978-3-319-67531-2_12)
24. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: RVHyper: a runtime verification tool for temporal hyperproperties. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 194–200. Springer, Cham (2018). [https://doi.org/10.1007/978-3-319-89963-3\\_11](https://doi.org/10.1007/978-3-319-89963-3_11)
25. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: Monitoring hyperproperties. *Formal Methods Syst. Des.* **54**(3), 336–363 (2019). <https://doi.org/10.1007/s10703-019-00334-z>

26. Finkbeiner, B., Zimmermann, M.: The first-order logic of hyperproperties. In: STACS 2017. LIPIcs, vol. 66, pp. 30:1–30:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). <https://doi.org/10.4230/LIPIcs.STACS.2017.30>
27. Gutsfeld, J.O., Müller-Olm, M., Ohrem, C.: Automata and fixpoints for asynchronous hyperproperties. In: POPL 2021, pp. 1–29 (2021). <https://doi.org/10.1145/3434319>
28. Hahn, C.: Algorithms for monitoring hyperproperties. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 70–90. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-32079-9\\_5](https://doi.org/10.1007/978-3-030-32079-9_5)
29. Hahn, C., Stenger, M., Tentrup, L.: Constraint-based monitoring of hyperproperties. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11428, pp. 115–131. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-17465-1\\_7](https://doi.org/10.1007/978-3-030-17465-1_7)
30. Han, Y.-S., Wang, Y., Wood, D.: Prefix-free regular-expression matching. In: Apostolico, A., Crochemore, M., Park, K. (eds.) CPM 2005. LNCS, vol. 3537, pp. 298–309. Springer, Heidelberg (2005). [https://doi.org/10.1007/11496656\\_26](https://doi.org/10.1007/11496656_26)
31. Han, Y.-S., Wood, D.: The generalization of generalized automata: expression automata. In: Domaratzki, M., Okhotin, A., Salomaa, K., Yu, S. (eds.) CIAA 2004. LNCS, vol. 3317, pp. 156–166. Springer, Heidelberg (2005). [https://doi.org/10.1007/978-3-540-30500-2\\_15](https://doi.org/10.1007/978-3-540-30500-2_15)
32. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: TeSSLa: runtime verification of non-synchronized real-time streams. In: SAC 2018, pp. 1925–1933. ACM (2018). <https://doi.org/10.1145/3167132.3167338>
33. McLean, J.: Security models and information flow. In: SP 1990, pp. 180–189. IEEE (1990). <https://doi.org/10.1109/RISP.1990.63849>
34. Penna, G.D., Intrigila, B., Tronci, E., Zilli, M.V.: Synchronized regular expressions. *Acta Informatica* **39**(1), 31–70 (2003). <https://doi.org/10.1007/s00236-002-0099-y>
35. Sánchez, C.: Synchronous and asynchronous stream runtime verification. In: VORTEX 2021, pp. 5–7. ACM (2021). <https://doi.org/10.1145/3464974.3468453>
36. Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjørner, N.S.: Symbolic finite state transducers: algorithms and applications. In: POPL 2012, pp. 137–150. ACM (2012). <https://doi.org/10.1145/2103656.2103674>
37. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: CSFW 2003, p. 29. IEEE (2003). <https://doi.org/10.1109/CSFW.2003.1212703>

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

