

Efficient Online Timed Pattern Matching by Automata-Based Skipping

Masaki Waga¹, Ichiro Hasuo², and Kohei Suenaga³

¹ University of Tokyo, Tokyo, Japan

² National Institute of Informatics, Tokyo, Japan

³ Kyoto University and JST PRESTO, Kyoto, Japan

Abstract. The *timed pattern matching* problem is an actively studied topic because of its relevance in *monitoring* of real-time systems. There one is given a log w and a specification \mathcal{A} (given by a *timed word* and a *timed automaton* in this paper), and one wishes to return the set of intervals for which the log w , when restricted to the interval, satisfies the specification \mathcal{A} . In our previous work we presented an efficient timed pattern matching algorithm: it adopts a skipping mechanism inspired by the classic Boyer–Moore (BM) string matching algorithm. In this work we tackle the problem of *online* timed pattern matching, towards embedded applications where it is vital to process a vast amount of incoming data in a timely manner. Specifically, we start with the Franek–Jennings–Smyth (FJS) string matching algorithm—a recent variant of the BM algorithm—and extend it to timed pattern matching. Our experiments indicate the efficiency of our FJS-type algorithm in online and offline timed pattern matching.

1 Introduction

Monitoring of real-time properties is an actively studied topic with numerous applications such as automotive systems [19], medical systems [8], data classification [6], web service [26], and quantitative performance measuring [12]. Given a specification \mathcal{A} and a log w of activities, monitoring would ask questions like: *if w has a segment that matches \mathcal{A} ; all the segments of w that match \mathcal{A} ; and so on.*

For a monitoring algorithm *efficiency* is a critical matter. Since we often need to monitor a large number of logs, each of which tends to be very long, one monitoring task can take hours. Therefore even *constant* speed up can make significant practical differences. Another important issue is an algorithm’s performance in *online usage scenarios*. Monitoring algorithms are often deployed in *embedded* applications [18], and this incurs the following online requirements:

- *Real-time properties*, such as: on prefixes of the log w , we want to know their monitoring result soon, possibly before the whole log w arrives.
- *Memory consumption*, such as: early prefixes of w should not affect the monitoring task of later segments of w , so that we can throw the prefixes away and free memory (that tends to be quite limited in embedded applications).
- *Speed* of the algorithm. In an online setting this means: if the log w arrives at a speed faster than the algorithm processes it, then the data that waits to be processed will fill up the memory.

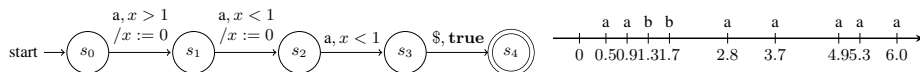


Fig. 1. An example of timed pattern matching. For the pattern timed automaton \mathcal{A} and the target timed word w , as shown, the output is the set of matching intervals $\{(t, t') \mid w|_{(t, t')} \in L(\mathcal{A})\} = \{(t, t') \mid t \in [3.7, 3.9), t' \in (6.0, \infty)\}$. Here \$ is a special terminal character.

Table 1. Matching problems

	log, target	specification, pattern	output
string matching	a word $w \in \Sigma^*$	a word $pat \in \Sigma^*$	$\{(i, j) \in (\mathbb{Z}_{>0})^2 \mid w(i, j) = pat\}$
pattern matching	a word $w \in \Sigma^*$	an NFA \mathcal{A}	$\{(i, j) \in (\mathbb{Z}_{>0})^2 \mid w(i, j) \in L(\mathcal{A})\}$
timed pattern matching	a timed word $w \in (\Sigma \times \mathbb{R}_{>0})^*$	a timed automaton \mathcal{A}	$\{(t, t') \in (\mathbb{R}_{>0})^2 \mid w _{(t, t')} \in L(\mathcal{A})\}$

Constant improvement in aspects like speed and memory consumption will be appreciated in online settings, too: if an algorithm is twice as fast, then this means the same monitoring task can be conducted with cheaper hardware that is twice slower.

The goal of the current paper is thus monitoring algorithms that perform well both in offline and online settings. We take a framework where *timed words*—they are essentially sequences of time-stamped events—stand for logs, and *timed automata* express a specification. Both constructs are well-known in the community of real-time systems. The problem we solve is that of *timed pattern matching*: see §2.1 for its definition; Fig. 1 for an example; and Table 1 for comparison with other matching problems.

Towards the goal our strategy is to exploit the idea of *skip values* in efficient string matching algorithms (such as Boyer–Moore (BM) [7]), together with their *automata-based extension* for pattern matching by Watson & Watson [34], to skip unnecessary matching trials. In our previous work [32] we took the strategy and introduced a timed pattern matching algorithm with BM-type skipping. The current work improves on this previous BM algorithm: it is based on the more recent *Franek–Jennings–Smyth (FJS) algorithm* [13] for string matching (instead of BM); and our new algorithm is faster than our previous BM-type one. Moreover, in online usage, our FJS-type algorithm better addresses the online requirements that we listed in the above. This is in contrast with our previous BM-type algorithm that works necessarily in an offline manner (it must wait for the whole log w before it starts).

Contributions Our main contribution is an efficient algorithm for timed pattern matching that employs (an automata-theoretic extension of) skip values from the Franek–Jennings–Smyth (FJS) algorithm for string matching [13]. By experiments we show that the algorithm generally outperforms a brute-force one and our previous BM algorithm [32]: it is twice as fast for some realistic automotive examples. Through our theoretical analysis as well as experiments on memory consumption, we claim that our algorithm is suited for online usage scenarios, too. We also compare its performance with a recent tool *Montre* for timed pattern matching [29], and observe that ours is faster, at least in terms of the implementations currently available.

In its course we have obtained an FJS-type algorithm for *untimed* pattern matching, which is one of the main contributions too. The algorithm is explained rather in detail, so that it paves the way to our FJS-type *timed* pattern matching that is more complex.

A central theme of the paper is benefits of the formalism of *automata*, a mathematical tool whose use is nowadays widespread in fields like temporal logic, model

checking, and so on. We follow Watson & Watson’s idea of extending skipping from string matching to pattern matching [34], where the key is overapproximation of words and languages by states of automata. Our main contribution on the conceptual side is that the same idea applies to *timed* automata as well, where we rely on *zone*-based abstraction (see e.g. [4, 5, 14]) for computing reachability.

Related Works Several algorithms have been proposed for online monitoring of real-time temporal logic specifications. An online monitoring algorithm for ptMTL (a past time fragment of MTL) is in [27] and an algorithm for MTL[U,S] (a variant of MTL with both forward and backward temporal modalities) is in [15]. In addition, a case study on an autonomous research vehicle monitoring [19] shows such procedures can be performed in an actual vehicle—this is where our motivation comes from, too.

We have chosen timed automata as a specification formalism. This is because of their expressivity as well as various techniques that operate on them. Some other formalisms can be translated to timed automata, and via translation, our algorithm offers to these formalisms an online monitoring algorithm. In [3], a variant of *timed regular expressions (TREs)* are proved to have the same expressive power as timed automata. For MTL and MITL, transformations into automata are introduced for many different settings; see e.g. [2, 10, 20, 22, 24].

The work with closest interests to ours is by Ulus, Ferrère, Asarin, Maler and their colleagues [29–31]. In their series of work, logs are presented by *signals*, i.e. values that vary over time. Their logs are thus *state-based* rather than *event-based* like timed words. Their specification formalism is timed regular expressions (TREs). An offline monitoring algorithm is presented in [30] and an online one is in [31]. These algorithms are implemented in the tool *Montre* [29], with which we conduct performance comparison. The difference between different specification formalisms (TREs, timed automata, temporal logics, etc.) are subtle, but for many realistic examples the difference does not matter. In the current paper we exploit various operations on automata, most notably zone-based abstraction.

Notations Let Σ be an alphabet and $w = a_1 a_2 \dots a_n \in \Sigma^*$ be a string over Σ , where $a_i \in \Sigma$ for each $i \in [1, n]$. We let $w(i)$ denote the i -th character a_i of w . Furthermore, for $i, j \in [1, n]$, when $i \leq j$ we let $w(i, j)$ denote the substring $a_i a_{i+1} \dots a_j$, otherwise we let $w(i, j)$ denote the empty string ε . The length n of the string w is denoted by $|w|$.

Organization of the Paper In §2 are preliminaries on: our formulation of the problem of timed pattern matching; and the FJS algorithm for string matching. The FJS-type skipping is extended to (untimed) pattern matching in §3, where we describe the algorithm in detail. This paves the way to our FJS-type timed pattern matching algorithm in §4. In §4 we also sketch zone-based abstraction of timed automata, a key technical ingredient in the algorithm. In §5 we present our experiment results. They indicate our algorithm’s performance advantage in both offline and online usage scenarios.

2 Preliminaries

2.1 Timed Pattern Matching

Here we formulate our problem. Our target strings are *timed words* [1], that are time-stamped words over an alphabet Σ . Our patterns are given by *timed automata* [1].

Definition 2.1 (timed word, timed word segment) For an alphabet Σ , a *timed word* is a sequence w of pairs $(a_i, \tau_i) \in (\Sigma \times \mathbb{R}_{>0})$ satisfying $\tau_i < \tau_{i+1}$ for any $i \in [1, |w| - 1]$. Let $w = (\bar{a}, \bar{\tau})$ be a timed word. We denote the subsequence $(a_i, \tau_i), (a_{i+1}, \tau_{i+1}), \dots, (a_j, \tau_j)$ by $w(i, j)$. For $t \in \mathbb{R}_{\geq 0}$, the *t-shift* of w is $(\bar{a}, \bar{\tau}) + t = (\bar{a}, \bar{\tau} + t)$ where $\bar{\tau} + t = \tau_1 + t, \tau_2 + t, \dots, \tau_{|w|} + t$. For timed words $w = (\bar{a}, \bar{\tau})$ and $w' = (\bar{a}', \bar{\tau}')$, their *absorbing concatenation* is $w \circ w' = (\bar{a} \circ \bar{a}', \bar{\tau} \circ \bar{\tau}')$ where $\bar{a} \circ \bar{a}'$ and $\bar{\tau} \circ \bar{\tau}'$ are usual concatenations, and their *non-absorbing concatenation* is $w \cdot w' = w \circ (w' + \tau_{|w|})$. We note that the absorbing concatenation $w \circ w'$ is defined only when $\tau_{|w|} < \tau'_1$.

For a timed word $w = (\bar{a}, \bar{\tau})$ on Σ and $t, t' \in \mathbb{R}_{>0}$ satisfying $t < t'$, a *timed word segment* $w|_{(t, t')}$ is defined by the timed word $(w(i, j) - t) \circ (\$, t')$ on the augmented alphabet $\Sigma \sqcup \{\$\}$, where i, j are chosen so that $\tau_{i-1} \leq t < \tau_i$ and $\tau_j < t' \leq \tau_{j+1}$. Here the fresh symbol $\$$ is called the *terminal character*.

Definition 2.2 (timed automaton) Let C be a finite set of *clock variables*, and $\Phi(C)$ denote the set of conjunctions of inequalities $x \bowtie c$ where $x \in C$, $c \in \mathbb{Z}_{\geq 0}$, and $\bowtie \in \{>, \geq, <, \leq\}$. A *timed automaton* $\mathcal{A} = (\Sigma, S, S_0, C, E, F)$ is a tuple where: Σ is an alphabet; S is a finite set of states; $S_0 \subseteq S$ is a set of initial states; $E \subseteq S \times S \times \Sigma \times \mathcal{P}(C) \times \Phi(C)$ is a set of transitions; and $F \subseteq S$ is a set of accepting states. The components of a transition $(s, s', a, \lambda, \delta) \in E$ represent: the source, target, action, reset variables and guard of the transition, respectively.

We define a *clock valuation* ν as a function $\nu : C \rightarrow \mathbb{R}_{\geq 0}$. We define the *t-shift* $\nu + t$ of a clock valuation ν , where $t \in \mathbb{R}_{\geq 0}$, by $(\nu + t)(x) = \nu(x) + t$ for any $x \in C$. For a timed automaton $\mathcal{A} = (\Sigma, S, S_0, E, C, F)$ and a timed word $w = (\bar{a}, \bar{\tau})$, a *run* of \mathcal{A} over w is a sequence r of pairs $(s_i, \nu_i) \in S \times (\mathbb{R}_{\geq 0})^C$ satisfying the following: (initiation) $s_0 \in S_0$ and $\nu_0(x) = 0$ for any $x \in C$; and (consecution) for any $i \in [1, |w|]$, there exists a transition $(s_{i-1}, s_i, a_i, \lambda, \delta) \in E$ such that $\nu_{i-1} + \tau_i - \tau_{i-1} \models \delta$ and $\nu_i(x) = 0$ (for $x \in \lambda$) and $\nu_i(x) = \nu_{i-1}(x) + \tau_i - \tau_{i-1}$ (for $x \notin \lambda$). A run only satisfying the consecution condition is a *path*. A run $r = (\bar{s}, \bar{\nu})$ is *accepting* if the last element $s_{|s|-1}$ of s belongs to F . The *language* $L(\mathcal{A})$ is defined to be the set $\{w \mid \text{there is an accepting run of } \mathcal{A} \text{ over } w\}$ of timed words.

Definition 2.3 (timed pattern matching) Let \mathcal{A} be a timed automaton, and w be a timed word, over a common alphabet Σ . The *timed pattern matching* problem requires all the intervals (t, t') for which the segment $w|_{(t, t')}$ is accepted by \mathcal{A} . That is, it requires the *match set* $\mathcal{M}(w, \mathcal{A}) = \{(t, t') \mid w|_{(t, t')} \in L(\mathcal{A})\}$.

The match set $\mathcal{M}(w, \mathcal{A})$ is in general uncountable; however it allows finitary representation, as a finite union of special polyhedra called *zones*. See [32].

2.2 String Matching and the FJS Algorithm

String matching is a fundamental problem in computer science. Given a *pattern string* pat and a *target string* w , it requires the set $\{(i, j) \in (\mathbb{Z}_{>0})^2 \mid w(i, j) = pat\}$ of all the occurrences of pat in w . A brute-force algorithm, by trying to match $|pat|$ characters for all the possible $|w| - |pat|$ positions of the pattern string, solves the string matching problem in $O(|pat||w|)$. Efficient algorithms for this classic problem have been sought

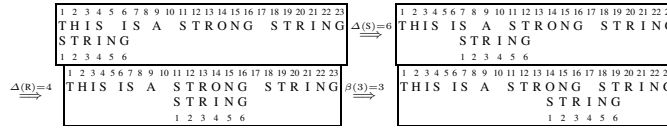


Fig. 2. The Franek–Jennings–Smyth (FJS) algorithm for string matching: an example

for a long time, with significant progress made as recently as in the last decade [11]. Among them the *Knuth–Morris–Pratt (KMP) algorithm* [21] and the *Boyer–Moore (BM) algorithm* [7] are well-known, where unnecessary matching trials are *skipped* utilizing *skip value functions*. Empirical studies have shown speed advantage of BM—and its variants like *Quick Search* [28]—over KMP, while theoretically KMP exhibits better worst-case complexity $O(|pat| + |w|)$. By combining KMP and Quick Search, the *Franek–Jennings–Smyth (FJS) algorithm* [13], proposed in 2007, achieves both linear worst-case complexity and good practical performance.

The current paper’s goal is to introduce FJS-like optimization to timed pattern matching. We therefore take the FJS algorithm as an example and demonstrate how skip values are utilized in the string matching algorithms we have mentioned.⁴

The FJS algorithm combines two skip value functions: $\Delta: \Sigma \rightarrow [1, |pat| + 1]$ and $\beta: [0, |pat|] \rightarrow [1, |pat|]$; the former Δ comes from Quick Search and the latter β comes from KMP (the choice of symbols follows [13]). See Fig. 2 where the pattern string $pat = \text{STRING}$ is shifted by 6, 4 and 3 (instead of one-by-one).

In the first shift we use the Quick Search skip value $\Delta(S) = 6$: we try matching the tail of pat ; it fails ($pat(6) \neq w(6)$); then we find that the next character $w(7) = S$ of the target only occurs in the first position of the pattern. Formally we define Δ by

$$\Delta(a) = \min(\{i \in [1, |pat|] \mid a = pat(|pat| - i + 1)\} \cup \{|pat| + 1\}) \quad \text{for } a \in \Sigma. \quad (1)$$

In the example of Fig. 2 we have $\Delta(I) = 3$ and $\Delta(Q) = 7$.

Now we are in the second configuration of Fig. 2 and we try matching the tail $pat(6) = G$ with $w(12)$. It fails and we invoke the Quick Search skip value function Δ ; this results in a shift by $\Delta(R) = 4$ positions.

For the shift from the third configuration to the fourth in Fig. 2 we employ the KMP skip value function β . It is defined as follows. Observe first that, in the third configuration of Fig. 2, matching trials from the head succeed for three positions and then fail ($w(11, 13) = pat(1, 3), w(14) \neq pat(4)$). From this information alone we can see that, for a potential string match, the pattern string must be shifted at least by $\beta(3) = 3$. See Fig. 3 where shifting the pattern string pat by one or two positions necessarily leads to a mismatch with $pat(1, 3)$. It is important here that we know $pat(1, 3)$ coincides with $w(11, 13)$ from the previous successful matching trials. Formally:

$$\beta(p) = \min\{n \in [1, |pat|] \mid pat(1, p - n) = pat(1 + n, p)\} \quad \text{for } p \in [0, |pat|]. \quad (2)$$

⁴ The FJS-type algorithm we present here is a simplified version of the original FJS algorithm. Our simplification is equipped with all the features that we will exploit later for pattern matching and timed pattern matching; the original algorithm further omits some other trivially unnecessary matching trials. We note that, because of the difference (that is conceptually inessential), our simplified algorithm (for string matching) no longer enjoys linear worst-case complexity.

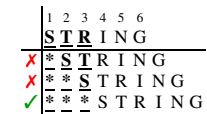


Fig. 3. $\beta(3) = 3$, where the argument 3 is the length of the successful partial match.

Algorithm 1 The FJS string matching algorithm (simplified)

Require: A target string w and a pattern string pat .
Ensure: Z is the set of matching intervals.

```

1:  $n \leftarrow 1$ ; ▷  $n$  is the position in  $w$  of the head of  $pat$ 
2: while  $n \leq |w| - |pat| + 1$  do
3:   while  $w(n + |pat| - 1) \neq pat(|pat|)$  do ▷ Try matching the tail of  $pat$ 
4:      $n \leftarrow n + \Delta(w(n + |pat|))$  ▷ Quick Search-type skipping
5:     if  $n > |w| - m + 1$  then return
6:   if  $pat = w(n, n + |pat| - 1)$  then ▷ We try matching from left to right
7:      $p \leftarrow |pat| + 1$ ;  $Z \leftarrow Z \cup \{[n, n + |pat| - 1]\}$ 
8:   else
9:      $p \leftarrow \min\{p' \mid pat(p') \neq w(n + p' - 1)\}$  ▷ Matching trials fail at position  $p$  for the first time
10:     $n \leftarrow n + \beta(p)$  ▷ KMP-type skipping

```

In the FJS algorithm we combine the two skip value function Δ and β . Specifically: let us be in a configuration where $pat(1)$ is in the position of $w(1 + n)$. We first try matching the pattern's tail $pat(|pat|)$ with its counterpart $w(|pat| + n)$; if it fails we invoke the Quick Search skipping Δ ; otherwise we turn to the pattern's head $pat(1)$ try matching from left to right. After its success or failure we invoke the KMP skipping β . Note that preference is given to the Quick Search skipping. See Algorithm 1.

It is important that the skip value functions $\Delta: \Sigma \rightarrow [1, |pat| + 1]$ and $\beta: [0, |pat|] \rightarrow [1, |pat|]$ rely only on the pattern string pat . Therefore it is possible to pre-compute the function values in advance (i.e. before a target string w arrives); moreover since $|pat|$ is usually not large those values can be stored effectively in look-up tables. Skipping by these skip values does not improve the worst-case complexity, but practically it brings pleasing constant speed up, as demonstrated in Fig. 2.

Finally we note the following alternative presentation of Δ and β .

$$\begin{aligned}
\Delta(a) &= \min\{n \in \mathbb{Z}_{>0} \mid \Sigma^n pat \cap \Sigma^{|pat|} a \Sigma^* \neq \emptyset\} & \text{for each } a \in \Sigma, \\
\beta(p) &= \min\{n \in \mathbb{Z}_{>0} \mid \Sigma^n pat(1, p) \cap pat(1, p) \Sigma^* \neq \emptyset\} & \text{for each } p \in [0, |pat|].
\end{aligned} \tag{3}$$

3 An FJS-Type Algorithm for Pattern Matching

In this section we present our first main contribution, namely an adaptation of the FJS algorithm (§2.2) from string matching to *pattern matching*.

Definition 3.1 (pattern matching) Let \mathcal{A} be a nondeterministic finite automaton over an alphabet Σ (a *pattern NFA*), and $w \in \Sigma^*$ be a *target string*. The *pattern matching* problem requires all the intervals (i, j) for which the substring $w(i, j)$ is accepted by \mathcal{A} . That is, it requires the set $\{(i, j) \mid 1 \leq i \leq j \leq |w| \text{ and } w(i, j) \in L(\mathcal{A})\}$.

For an example see Fig. 4, where the automaton \mathcal{A} satisfies $L(\mathcal{A}) = L(\{\mathbf{ab}, \mathbf{cd}\} \mathbf{cc}^* \mathbf{d})$.

A brute-force algorithm solves pattern matching in $O(|S||w|^2)$, where S is the state space of the pattern \mathcal{A} (the factor $|S|$ is there due to nondeterminism). Some optimizations are known, among which is the adaptation of the Boyer–Moore algorithm by Watson & Watson [34]. In their algorithm they adapt the BM-type skip values to pattern matching: the core idea in doing so is to *overapproximate* languages and substrings, so that the skip value function can be organized as a finite table and hence can be computed in advance. Our adaptation of the FJS algorithm employs similar overapproximation.



Fig. 4. Pattern matching. For the pattern NFA \mathcal{A} on the left, for which it is easy to see that $L(\mathcal{A}) = L(\{ab, cd\}cc^*d)$, the output is $\{(9, 12)\}$ as shown on the right.

$$L(\mathcal{A}) = \left\{ \begin{array}{l} abcd, \quad cdcd, \\ abccd, \quad cdccd, \\ abcccd, \quad cdcccd, \\ \vdots \end{array} \right\} \rightsquigarrow L'' = L' \cdot \Sigma^* = \left\{ \begin{array}{l} abcd, \quad cdcd, \\ abcc, \quad cdcc \end{array} \right\} \Sigma^*$$

Fig. 5. Overapproximation of the language $L(\mathcal{A})$

In the original FJS algorithm (for string matching) one uses skip value functions

$$\Delta: \Sigma \rightarrow [1, |pat| + 1] \quad \text{and} \quad \beta: [0, |pat|] \rightarrow [1, |pat|] . \quad (4)$$

One may wonder what we can use in place of $|pat|$, now that the pattern \mathcal{A} can accept infinitely many words that are unboundedly long.

It turns out that our adaptations have the types

$$\Delta: \Sigma \rightarrow [1, m + 1] \quad \text{and} \quad \beta: S \rightarrow [1, m] , \quad (5)$$

where m is the length of the shortest words accepted by \mathcal{A} and S is the state space of \mathcal{A} . Intuitively, the original Δ does a comparison of the pattern pat with a character $a \in \Sigma$ and the original β does a comparison of pat with the substring $w(i, j)$ of the target string we actually read in the last matching trial. Thus the adaptation can be done by a finite presentation of the overapproximation of $L(\mathcal{A})$ and $w(i, j)$.

More specifically, for the approximation of $L(\mathcal{A})$: 1) we focus on the length m of the shortest accepted strings (four in the example of Fig. 4); 2) we collect all the prefixes of length m that appear in $L(\mathcal{A})$ ($abcd, cdcd, abcc, cdcc$ in the same example); and 3) we let an overapproximation L'' consist of any word that starts with those prefixes. See Fig. 5 for illustration; precise definitions are as follows.

$$m = \min\{|w| \mid w \in L(\mathcal{A})\} \quad L' = \{w' \in \Sigma^m \mid \exists w'' \in \Sigma^*. w'w'' \in L(\mathcal{A})\} \quad L'' = L' \cdot \Sigma^*$$

Here $L' \subseteq \Sigma^m$ is necessarily a finite set; thus $L'' = L' \cdot \Sigma^*$ is an overapproximation of $L(\mathcal{A})$ with a finite representation L' .

For the overapproximation of the substring $w(i, j)$ that we actually read at the last matching trial, we exploit the set $\mathcal{S}(w(i, j)) = \{s \in S \mid s_0 \xrightarrow{w(i, j)} s \text{ in } \mathcal{A}\}$ of states of \mathcal{A} . We have $w(i, j) \in \{w' \mid \forall s \in \mathcal{S}(w(i, j)), \exists s_0 \in S_0. s_0 \xrightarrow{w'} s \text{ in } \mathcal{A}\}$, when $\mathcal{S}(w(i, j)) \neq \emptyset$. Using the overapproximation same as the one for L' , we obtain an overapproximation of such $w(i, j)$ represented by at most $2^{|S|}$ sets.

Let us demonstrate our two skip value functions Δ and β using the example in Fig. 4; the execution trace of our algorithm is in Fig. 6. In the first configuration we try to match the tail of all the possible length-4 prefixes of $L(\mathcal{A})$ with $w(4) = a$, which fails. Then we invoke the Quick Search-type skipping $\Delta(w(5)) = \Delta(b)$; since b occurs no later than in the second position in $L' = \{abcd, abcc, cdcd, cdcc\}$, we can skip by three positions and reach the second configuration.

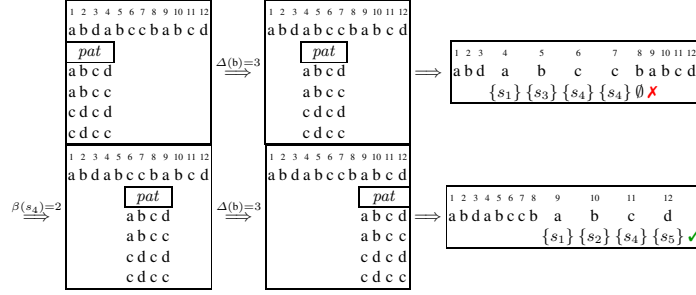


Fig. 6. Our FJS-type algorithm for pattern matching, for the example in Fig. 4

Algorithm 2 The FJS algorithm for pattern matching, for a target w and a pattern \mathcal{A}

Ensure: Z is the set of matching intervals.

- 1: $n \leftarrow 1;$ ▷ n is the position in w of the head of pat
- 2: **while** $n \leq |w| - m + 1$ **do**
- 3: **while** $\forall w' \in L'. w(n + m - 1) \neq w'(m)$ ▷ Try matching the tail of L'
- 4: $n \leftarrow n + \Delta(w(n + m))$ ▷ Quick Search-type skipping
- 5: **if** $n > |w| - m + 1$ **then return**
- 6: $Z \leftarrow Z \cup \{(n, n') \mid w(n, n') \in L(\mathcal{A})\}$ ▷ We try matching by feeding $w(n, |w|)$ to \mathcal{A}
- 7: $n' \leftarrow \max\{n' \in [1, |w|] \mid \exists s_0 \in S_0, s \in S. s_0 \xrightarrow{w(n, n')} s\}$ ▷ n' is the position of the last successful match
- 8: $S' \leftarrow \{s \in S \mid \exists s_0 \in S_0. s_0 \xrightarrow{w(n, n')} s\}$ ▷ Matching trials stack at the states S'
- 9: $n \leftarrow n + \max_{s \in S'} \beta(s)$ ▷ KMP-type skipping

We again try matching from the tail; this time we succeed since $w(7) = c$ appears as a tail in L' . We subsequently move to the phase where we match from left to right, much like in the original FJS algorithm (§2.2). Concretely this means we feed the automaton \mathcal{A} (see Fig. 6) the remaining segment $w(4)w(5) \dots$ from left to right; we obtain $\{s_1\}\{s_3\}\{s_4\}\{s_4\}\emptyset$ as the sequence of reachable sets. Since no accepting states occur therein and we have reached the emptyset, we conclude that the matching trial starting at the position $w(4)$ is unsuccessful.

Now we invoke the KMP-type skipping β . In the original FJS algorithm we used the data of successful partial matching ($w(4, 7) = abcc$ in the current case) for computing β ; this is not possible, however, since it is infeasible to prepare skip values for all possible $w(i, j)$. Instead we use the data $\mathcal{S}(w(4, 7)) = \{s_4\}$ and the set $L'_{s_4} = \{abc, cdc\}$ as an overapproximation of the partial match $w(4, 7) = abcc$. The intuition of the set L'_{s_4} is that: for a word w'

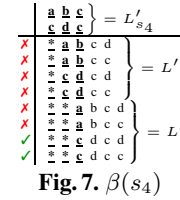


Fig. 7. $\beta(s_4)$

to drive \mathcal{A} from an initial state to s_4 , w' must have either abc or cdc as its prefix. In Fig. 7 is how we compute the skip value $\beta(s_4)$, using the approximant L'_{s_4} of the partial match and the approximant L' of the pattern. Note also that it follows the same pattern as Fig. 3.

We are now in the fourth configuration in Fig. 6. The matching trial at the position 9 fails and we invoke the Quick Search-type skipping, much like before. In the fifth configuration, the matching trial at the position 12 succeeds, which makes us try matching from the left, feeding \mathcal{A} with $w(9, 12)$. We reach s_5 and thus succeed.

Overall our FJS-type algorithm for pattern matching is as in Algorithm 2. The skip value functions therein are defined as follows. They are similar to the ones in (3). Since L' and L'_s are all finite, computing Δ and β is straightforward.

Definition 3.2 (Skip values in our FJS-type pattern matching algorithm) Let $\mathcal{A} = (\Sigma, S, S_0, E, F)$ be a pattern NFA, $a \in \Sigma$ be a character, s be a state of \mathcal{A} , and $\mathcal{A}_s = (\Sigma, S, S_0, E, \{s\})$ be the automaton where s is the only accepting state. Let $m_s = \min\{|w| \mid w \in L(\mathcal{A}_s)\}$ (the length of a shortest word that leads to s) and $m = \min_{s \in F} m_s$ (the length of a shortest accepted word). The skip value functions $\Delta : \Sigma \rightarrow [1, m + 1]$ and $\beta : S \rightarrow [1, m]$ are defined as follows.

$$\begin{aligned} L' &= \{w(1, m) \mid w \in L(\mathcal{A})\} & L'_s &= \{w(1, \min\{m_s, m\}) \mid w \in L(\mathcal{A}_s)\} \\ \Delta(a) &= \min\{n \in \mathbb{Z}_{>0} \mid \Sigma^n L' \cap \Sigma^m a \Sigma^* \neq \emptyset\} \\ \beta(s) &= \min\{n \in \mathbb{Z}_{>0} \mid \Sigma^n L' \cap L'_s \Sigma^* \neq \emptyset\} \end{aligned}$$

4 An FJS-Type Algorithm for Timed Pattern Matching

Here we present our second main contribution: an FJS-type algorithm for timed pattern matching. It is superior to our previous Boyer–Moore-type algorithm [32], in its performance both in offline and online scenarios. We fix a target timed word $w = (\bar{a}, \bar{\tau})$ and a pattern timed automaton $\mathcal{A} = (\Sigma \sqcup \{\$, \}, S, S_0, C, E, F)$. We further assume the following that means \mathcal{A} is a suitable pattern for timed pattern matching.

Assumption 4.1 \mathcal{A} satisfies the following: any transition to an accepting state is labelled with the terminal character $\$$; no other transition is labelled with $\$$; and there is no transition from an accepting state.

The basic idea of our FJS-type algorithm here is the same as in §3: we use two skip value functions Δ and β ; and for their finitary representation we let states of automata overapproximate various infinitary data, as we explain later. In the current timed setting, however, we cannot use a pattern timed automaton \mathcal{A} itself to play the same role—in a run of \mathcal{A} a state is always accompanied with a clock valuation that takes continuous values. We overcome this difficulty relying on the *zone abstraction* of timed automata, a construction that turns a timed automaton into an NFA maintaining reachability (see e.g. [14]).⁵

Definition 4.2 (zone) Let \mathcal{A} be a timed automaton over the set C of clock variables, and M be the maximum constant occurring in the guards of \mathcal{A} . A *zone* is a $|C|$ -dimensional polyhedron specified with a conjunction of the constraints of the form $\nu(x_j) - \nu(x_i) \prec c$, $\nu(x_i) \prec c$ or $-\nu(x_i) \prec c$, where $\prec \in \{<, \leq\}$ and $c \in [-M, M]$.

A *zone automaton* \mathcal{Z} for a timed automaton \mathcal{A} is an NFA whose states are pairs (s, α) of a state s of \mathcal{A} and a zone α ; it is meant to be a finite abstraction of the timed automaton \mathcal{A} via which we study properties of \mathcal{A} . There are many different known constructions of zone automata (see e.g. [4, 14]): they come with different efficiency (i.e. the size of the resulting NFA), and with different preservation properties (bisimilarity to \mathcal{A} , similarity, etc.). For our current purpose it does not matter which precise construction we use; we chose a common construction SG^a from [14], mainly for its ease of implementation.

A *path* of a zone automaton \mathcal{Z} is much like a run, but it is allowed to start at a possibly non-initial state. A path $r = (\bar{s}, \bar{\nu})$ of a timed automaton \mathcal{A} is called an *instance*

⁵ In our previous work [32] we used *regions* [1] in place of zones. Though equivalent in terms of finiteness, zones give more efficient abstraction than regions.

Algorithm 3 Our FJS-type algorithm for timed pattern matching, for a target w and a pattern \mathcal{A}

Ensure: Z is the match set $\mathcal{M}(w, \mathcal{A})$ in Def. 2.3.

```

1:  $n \leftarrow 1$ ;  $\triangleright n$  is the position in  $w$  of the beginning of the current matching trial
2:  $\nu_0 \leftarrow$  (the clock valuation that returns 0 for any clock variable)
3: while  $n \leq |w| - m + 2$  do
4:   while  $\forall \bar{\tau} \in L'. \bar{\alpha}_{n+m-2} \neq a'$  (where  $a'$  is such that  $\bar{\tau}_{m-2} \xrightarrow{a'} \bar{\tau}_{m-1}$ ) do  $\triangleright$  Try matching the tail of  $L'$ 
5:      $n \leftarrow n + \Delta(\bar{\alpha}_{n+m-1})$   $\triangleright$  Quick Search-type skipping
6:     if  $n > |w| - m + 2$  then return
7:      $Z \leftarrow Z \cup \{(t, t') \in [\bar{\tau}_{n-1}, \bar{\tau}_n) \times (\bar{\tau}_{n-1}, \infty) \mid w|_{(t, t')} \in L(\mathcal{A})\}$   $\triangleright$  Try matching from left to right
8:      $n' \leftarrow \max\{n' \in [1, |w|] \mid \exists s_0 \in S_0, s \in S, \nu \in (\mathbb{R}_{\geq 0})^C. (s_0, \nu_0) \xrightarrow{w(n, n')} (s, \nu)\}$ 
9:      $S' \leftarrow \{s \in S \mid \exists s_0 \in S_0, \nu \in (\mathbb{R}_{\geq 0})^C. (s_0, \nu_0) \xrightarrow{w(n, n')} (s, \nu)\}$   $\triangleright$  Matching trials stack at the states  $S'$ 
10:     $n \leftarrow n + \max_{s \in S'} \beta(s)$   $\triangleright$  KMP-type skipping

```

of a path $\bar{r} = (\bar{s}, \bar{\alpha})$ of a zone automaton \mathcal{Z} for \mathcal{A} if, for any $n \in [0, |s| - 1]$, we have $\bar{\nu}_n \in \bar{\alpha}_n$. Conversely, such \bar{r} is called an *abstraction* of r . In this paper we rely on the following preservation property of the specific construction $\mathcal{Z} = SG^a(\mathcal{A})$ of zone automata: every run in $SG^a(\mathcal{A})$ is an abstraction of some run of \mathcal{A} ; conversely every run of \mathcal{A} is an instance of some run in $SG^a(\mathcal{A})$. See [14] for details.

Our algorithm is in Algorithm 3. The constructs therein are defined as follows.

Definition 4.3 (FJS-type skip values for timed pattern matching) Let \bar{r} be a path of the zone automaton $SG^a(\mathcal{A})$. The set $\mathcal{W}(\bar{r})$ of timed words represented by \bar{r} is:

$$\mathcal{W}(\bar{r}) = \{w \in (\Sigma \times \mathbb{R}_{>0})^* \mid \text{there is a path } r \text{ of } \mathcal{A} \text{ over } w \text{ that is an instance of } \bar{r}\} .$$

For a set K of paths of $SG^a(\mathcal{A})$, the definition naturally extends by $\mathcal{W}(K) = \bigcup_{\bar{r} \in K} \mathcal{W}(\bar{r})$. Let $\mathcal{A}_s = (\Sigma, S, S_0, E, C, \{s\})$ be the modification of \mathcal{A} in which s is the only accepting state. Let $m_s = \min\{|w| \mid w \in L(\mathcal{A}_s)\}$ and $m = \min_{s \in F} m_s$. Following the discussion in §3, we define the overapproximations L'' of $L(\mathcal{A})$ and L'_s , as follows. Note that L' and L'_s are in fact sets of runs of $SG^a(\mathcal{A})$; L'' is a set of timed words.

$$\begin{aligned}
L' &= \{\bar{r}(0, m-1) \mid \bar{r} \text{ is a run of } SG^a(\mathcal{A}), \text{ and } \mathcal{W}(\bar{r}) \cap L(\mathcal{A}) \neq \emptyset\} \\
L'' &= \mathcal{W}(L') \cdot (\Sigma \times \mathbb{R}_{>0})^* \\
L'_s &= \{\bar{r}(0, \min\{m_s, m-1\}) \mid \bar{r} \text{ is a run of } SG^a(\mathcal{A}), \text{ and } \mathcal{W}(\bar{r}) \cap L(\mathcal{A}_s) \neq \emptyset\}
\end{aligned}$$

These are used in the following definition of skip values. Here $a \in \Sigma$ and $s \in S$.

$$\begin{aligned}
\Delta(a) &= \min\{n \in \mathbb{Z}_{>0} \mid \\
&\quad \exists t \in \mathbb{R}_{>0}. (\Sigma \times \mathbb{R}_{>0})^n \cdot \mathcal{W}(L') \cap (\Sigma \times \mathbb{R}_{>0})^{m-1} \cdot (a, t) \cdot (\Sigma \times \mathbb{R}_{>0})^* \neq \emptyset\} \quad (6) \\
\beta(s) &= \min\{n \in \mathbb{Z}_{>0} \mid (\Sigma \times \mathbb{R}_{>0})^n \cdot \mathcal{W}(L') \cap \mathcal{W}(L'_s) \cdot (\Sigma \times \mathbb{R}_{>0})^* \neq \emptyset\}
\end{aligned}$$

Note the similarity between the last definition and (3).

Explanation is in order how some operations in Algorithm 3 (and in Def. 4.3) can be implemented. First note that $\mathcal{W}(\bar{r})$ is an infinite set. The set L' is finite and computable nevertheless: due to the preservation property of the zone automaton $SG^a(\mathcal{A})$, the condition $\mathcal{W}(\bar{r}) \cap L(\mathcal{A}) \neq \emptyset$ simply means \bar{r} is accepting. The same goes for L'_s . For Δ , we realize that the second argument $(\Sigma \times \mathbb{R}_{>0})^{m-1} \cdot (a, t) \cdot (\Sigma \times \mathbb{R}_{>0})^*$ of the intersection does not pose any timing constraint. Therefore the timed nonemptiness problem reduces to an untimed one that is readily solved. Solving the timed nonemptiness problem for β

in (6) is nontrivial. Here we use emptiness check in $SG^a(\mathcal{A} \times \mathcal{A})$ —the zone automaton of the product of \mathcal{A} with itself, changing its initial state suitably in order to address shift of words—to check whether the intersection of the two relevant languages is empty. Finally, the left-to-right matching on Line 7 is done by accumulating constraints on t in the course of necessary transitions. Further details are in Appendices A–B.

A correctness proof (i.e. our skipping does not affect the output) is in Appendix C.

One important idea in our algorithm is that we use timing constraints—in addition to character constraints like in Fig. 3 & 7—in calculating skip values. By this we achieve greater skip values, while keeping the computational overhead minimal by the use of the zone automaton $SG^a(\mathcal{A} \times \mathcal{A})$.

The way our algorithm (Algorithm 3) operates is very similar to the one in §3 for (untimed) pattern matching, as we already described earlier. There the zone automaton $SG^a(\mathcal{A})$ plays important roles in the calculation of skip values. For the record we include in Appendix B the illustration of our algorithm using the example in Fig. 1.

Online Properties We claim that the current FJS-type algorithm is much better suited to online usage scenarios than our previous BM-type one [32]. See Fig. 8. In our FJS-type algorithm we can sometimes increment n before reading the whole target timed word w (“unnec.” for “unnecessary” in Fig. 8); this is the case when we observe that no further transition is possible in the pattern automaton \mathcal{A} . (Additionally, thanks to the skip values Δ and β , sometimes we can increment n by more than one). For real-world examples we can assume that matches tend to be much shorter than the whole log w ; this means the “unnec.” parts are often big.

In the BM-type algorithm, in contrast, matching trials start almost at the tail of w ,⁶ and we have to wait until the arrival of the whole target word. This contrast is witnessed in our experimental results, specifically on those for memory usage.

5 Experiments

We implemented our FJS-type algorithm for timed pattern matching—its online and offline variations difference between which will be elaborated later. We compared its performance with that of: brute-force algorithms (online and offline); the BM-type algorithm [32]; and the tool *Montre* [29] for timed pattern matching.

	brute-force	BM	FJS	Montre
offline	from [32]	from [32]	new	from [29]
online	from [32]	—	new	from [29]

The BM- and FJS-type algorithms employ zone-based abstraction; it is implemented using *difference bound matrices*, following [9]. Zone construction and calculation of skip values are done in the preprocessing stage, where the most expensive is the emptiness checking for $\beta(s)$ (see (6)). We optimized this part, memorizing parts of zone automata and reusing them in computing $\beta(s)$ for different s . As a result the preprocessing stage takes a fraction of a second for each of our benchmark problems. See Appendix E for details.

For brute-force and FJS, the algorithms are the same in their online and offline implementations. In the online implementations, a target timed word is read lazily and

⁶ To be precise we can start without the last $m - 1$ characters, where m is the length of a shortest word accepted by \mathcal{A} . Usually m is by magnitude smaller than $|w|$.

a memory cell is deallocated as soon as we realize it is no longer needed. In the offline implementations, the whole target timed word is read and stored in memory in the beginning, and the memory cells are not deallocated until the end. The tool *Montre* employs different algorithms in its online and offline usage modes. See [29] for details.

In our current implementations, we hardcode a pattern timed automaton in the code. Developing a parser for user-defined timed automata should not be hard.

The benchmark problems we used are in Fig. 9–14 (the pattern automata \mathcal{A} and the set W of target words). They are from automotive scenarios except for the first two.

5.1 Comparison with the Brute Force and BM-Type Algorithms

We implemented the brute-force, BM, FJS algorithms in C++ [33] and we compiled them by `clang-800.0.42.1`. All the experiments are done on MacBook Pro Early 2013 with 2.6 GHz Intel Core i5 processor and 8 GB 1600MHz DDR3 RAM.

Speed (i.e. Permissible Density in Online Usage) In Fig. 15–20 are the comparison of the offline implementations of the brute-force, BM and FJS algorithms, respectively (average of five runs). Preprocessing time is excluded (it is anyway negligible, see Appendix E). We also exclude time of loading the input timed word in memory; this is because in many deployment scenarios like embedded ones, I/O is pipelined by, for example, DMA.

The pattern automata for the benchmarks *TORQUE*, *SETTING*, and *GEAR* look similar to each other. However their input timed words—generated by a suitable Simulink model for each benchmark—exhibit different characteristics, such as how often the characters in the pattern automaton occur in the input timed words. Accordingly the performance of the timed pattern matching algorithms varies, as we see in Fig. 17–19.

We observe that our FJS algorithm generally outperforms the BM and brute-force ones. For *SETTLING* and *ACCEL* the performance gap is roughly twice, and it possibly makes a big practical difference e.g. when a data set is huge and the monitoring task takes hours. For *LARGE CONSTRAINTS* it seems to depend on specific words which algorithm performs better. The advantage in performance is as we expected, given that the FJS algorithm combines the KMP-type skipping (that works well roughly when the BM-type one does) and the Quick Search-type skipping (that complements KMP). After all, it is encouraging to observe that our FJS algorithm performs better in the automotive examples, where our motivation is drawn.

In every benchmark except for *LARGE CONSTRAINT*, the execution time grows roughly linearly on the length of the input word. This is a pleasant property for monitoring algorithms for which an input word can be very long.

These results for *offline* implementations also support our claim of FJS’s superiority in *online* usage scenarios. In online usage we must process an input word faster than the speed with which the word arrives; otherwise the word eventually floods memory. Thus running twice as fast means that our algorithm can handle twice as dense input—or that we can use cheaper hardware to conduct the same monitoring task. Note that the difference between our online and offline implementations is only in the memory management and I/O. Thus their speed should be similar.

Memory Usage In Table 6 is the memory consumption of our *online* FJS implementation and that of BM, for the *SETTLING* benchmark (the tendency is the same for the

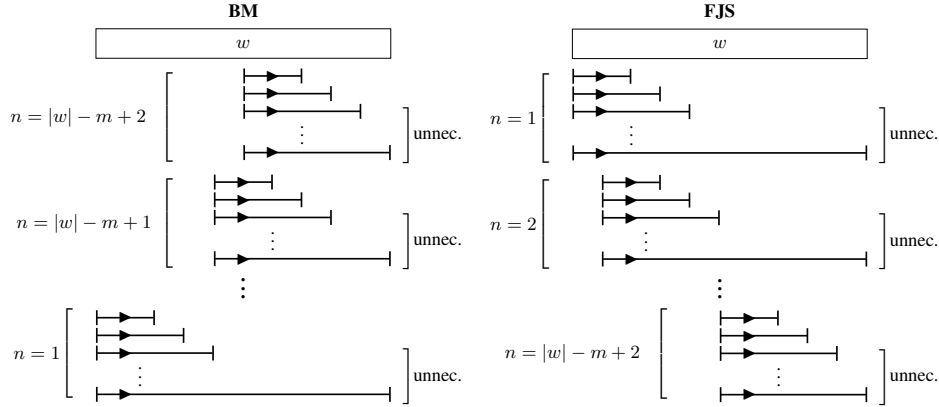


Fig. 8. How matching trials proceed: our previous BM-type algorithm (on the left) and our current FJS-type algorithm (on the right).

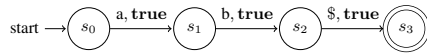


Fig. 9. SIMPLE from [32]. The set W consists of alternations of a and b whose length is from 20 to 1,024,000. Timing is random.

$\langle\langle\langle(p \cdot \neg p)_{(0,10]}^* \wedge (q \cdot \neg q)_{(0,10]}^* \rangle\rangle_{(0,80)} \cdot \$ \rangle_{(0,80)}$

Fig. 10. LARGE CONSTRAINTS from [32]. The pattern \mathcal{A} is a translation of the above timed regular expression (5 states and 9 transitions). The set W consists of superpositions of the alternations $p, \neg p, p, \neg p, \dots$ and $q, \neg q, q, \neg q, \dots$ whose timing follows a certain exponential distribution. The length of words in W is from 1,934 to 31,935. The pattern \mathcal{A} is in Fig. 24

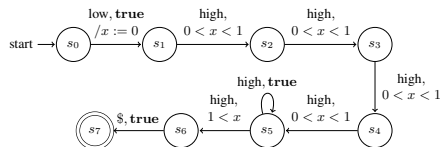


Fig. 11. TORQUE, an automotive example from [32]. It monitors for five or more consecutive occurrences of high in one second. The target words in W (length 242,808–4,873,207) are generated by the model `sldemo_enginewc.slx` in the Simulink Demo palette [23] with random input.

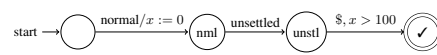


Fig. 12. SETTLING. The set W (length 472–47,200,000) is generated by the Simulink powertrain model in [17]. The pattern (Requirement (32) in [17]) is for an event in which the system remains unsettled for 100 seconds after moving to the normal mode.



Fig. 13. GEAR. The set W (length 307–1,011,427) is generated by the automatic transmission system model in [16]. The pattern, from ϕ_5^{AT} in [16], is for an event in which gear shift occurs too quickly (from the 1st to 2nd).

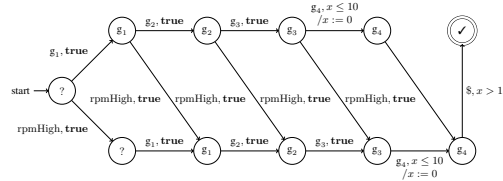


Fig. 14. ACCEL. The set W (length 25,002–17,280,002) is generated by the same automatic transmission system model as in GEAR. The pattern is from ϕ_8^{AT} in [16]: although the gear shifts from 1st to 4th and RPM is high enough somewhere in its course, the vehicle velocity is not high enough (i.e. the character `veloHigh` is absent).

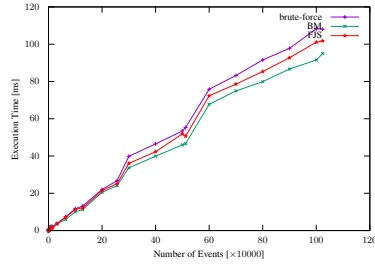


Fig. 15. SIMPLE: exec. time

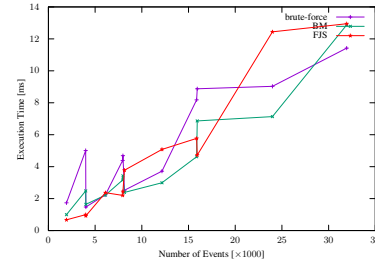


Fig. 16. LARGE CONSTRAINTS: exec. time

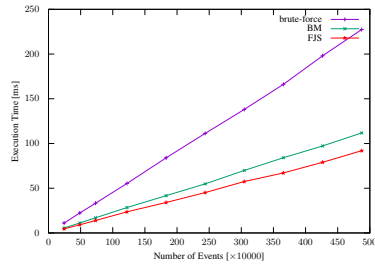


Fig. 17. TORQUE: exec. time

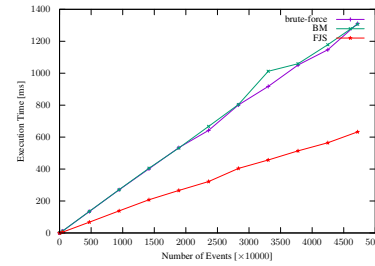


Fig. 18. SETTLING: exec. time

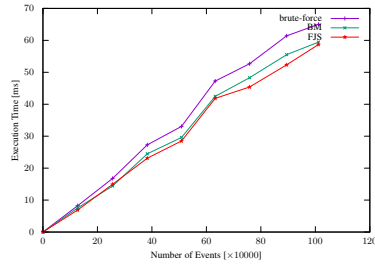


Fig. 19. GEAR: exec. time

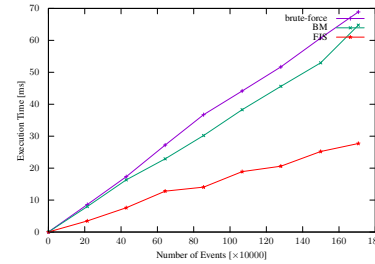


Fig. 20. ACCEL: exec. time

Table 2. SIMPLE (sec.)

$ w $	FJS (online)	Montre (offline)	Montre (online)
32,000	0.01	0.05	3.00
64,000	0.02	0.10	6.06
100,000	0.03	0.16	9.41
128,000	0.04	0.20	12.54
200,000	0.07	0.31	18.89
256,000	0.09	0.40	23.76
300,000	0.10	0.48	28.19
400,000	0.14	0.63	38.24
500,000	0.18	0.78	46.33
512,000	0.18	0.81	48.77
600,000	0.21	0.96	56.76
700,000	0.25	1.13	66.53
800,000	0.28	1.28	74.91
900,000	0.32	1.43	84.58
1,000,000	0.36	1.60	93.52
1,024,000	0.37	1.62	95.62

Table 3. SETTLING (sec.)

$ w $	FJS (online)	Montre (offline)	Montre (online)
300	0.00	0.01	0.01
30,000	0.01	0.01	0.01
300,000	0.11	0.01	0.01
3,000,000	1.11	3.85	299.85
6,000,000	2.23	7.74	600.66
9,000,000	3.34	11.66	893.88
12,000,000	4.46	15.65	1,188.02
15,000,000	5.58	19.75	1,475.89
18,000,000	6.72	24.48	1,788.18
21,000,000	9.27	27.80	Timeout
24,000,000	8.96	31.78	Timeout
27,000,000	10.09	37.10	Timeout
30,000,000	11.21	41.10	Timeout

Table 4. GEAR (sec.)

$ w $	FJS (online)	Montre (offline)	Montre (online)
1,000	0.00	0.01	0.04
86,400	0.04	0.15	11.63
172,800	0.08	0.29	23.48
259,200	0.13	0.42	37.51
345,600	0.17	0.54	47.20
432,000	0.21	0.67	57.99
518,400	0.25	0.85	69.76
604,800	0.30	0.96	87.59
691,200	0.34	1.09	90.36

Table 6. Memory consumption of FJS (online) and BM

$ w $	BM (MB)	FJS (MB)
300	1.16	1.16
30,000	2.61	1.16
300,000	15.55	1.16
3,000,000	145.21	1.16
6,000,000	289.25	1.16
9,000,000	433.31	1.16
12,000,000	577.32	1.19
15,000,000	721.37	1.18
18,000,000	865.42	1.19
21,000,000	1,009.46	1.16
24,000,000	1,153.50	1.16
27,000,000	1,297.57	1.16
30,000,000	1,441.61	1.16

Table 5. ACCEL (sec.)

$ w $	FJS (online)	Montre (offline)	Montre (online)
1,000	0.00	0.01	69.05
86,400	0.06	0.63	Timeout
172,800	0.13	1.25	Timeout
259,200	0.20	1.88	Timeout
345,600	0.26	2.50	Timeout
432,000	0.33	3.12	Timeout
518,400	0.40	3.75	Timeout
604,800	0.46	4.38	Timeout
691,200	0.53	4.99	Timeout

other benchmarks). The absolute values are not very important because they include our program and dynamically linked libraries; what matters is the tendency that memory consumption is almost constant for online FJS while it increases for BM. Constant memory consumption is an important property for monitoring algorithms, especially in online usage. The results here also concurs with our theoretical observation at the end of §4 (see Fig. 8).

5.2 Comparison with Montre

Here we compare with Montre, a recent tool for (both online and offline) timed pattern matching [29]. Montre’s online and offline algorithms differ from each other; both of them are quite different from our FJS algorithm, too. Montre’s emphasis is on the algebraic structure of timed regular expressions and compositional reasoning thereby, while our algorithm features automata-theoretic views on the problem.

Since we had difficulty running Montre in the same environment as in §5.1, we instead used GCC 4.9.3 as a compiler, and conducted experiments on an Amazon EC2 c4.xlarge instance (April 2017, 4 vCPUs and 7.5 GB RAM) that runs Ubuntu 14.04 LTS (64 bit). The timeout is set to thirty minutes.

In Tables 2–5 are the results. Here we use the benchmarks SIMPLE, SETTLING, GEAR, and ACCEL, for which the translation between timed words (our input) and signals (Montre’s input) makes sense. Our (online) FJS implementation is about 3 to 8 times faster than offline Montre and about 250 times faster than online Montre. The big performance advantage over *online* Montre can be attributed to various reasons, including: 1) online Montre needs to frequently compute derivatives of TREs; 2) online Montre is comparable to our brute-force algorithm in that there is no skipping involved; and 3) Montre is implemented in a functional language (Pure [25]) that is in general slower. The reason for the advantage over *offline* Montre is yet to be seen: given that the algorithms are very different, the advantage may well be solely attributed to implementation details. We claim however that good online performance of our FJS algorithm is a big advantage for monitoring applications.

6 Conclusions and Future Work

We continued [32] and presented an algorithm for timed pattern matching. Based on the FJS algorithm [13] it exhibits better online properties, as witnessed in our experiments. As future work we wish to implement an interface of our experimental implementation and distribute as a tool. We also wish to try the algorithm in actual embedded hardware, like [18].

Acknowledgments Thanks are due to Sean Sedwards for useful discussions and comments. The authors are supported by JSPS Grant-in-Aid 15KT0012. M.W. and I.H. are supported by JST ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), and JSPS Grant-in-Aid No. 15K11984. K.S. is supported by JST PRESTO (No. JPMJPR15E5) and JSPS Grant-in-Aid No. 70633692.

References

1. R. Alur and D.L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
2. R. Alur and T.A. Henzinger. Back to the future: Towards a theory of timed regular languages. In *33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, Pennsylvania, USA, 24-27 October 1992*, pp. 177–186. IEEE Computer Society, 1992.
3. E. Asarin, P. Caspi and O. Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, 2002.
4. G. Behrmann, P. Bouyer, E. Fleury and K.G. Larsen. Static guard analysis in timed automata verification. In H. Garavel and J. Hatcliff, editors, *TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, vol. 2619 of *Lecture Notes in Computer Science*, pp. 254–277. Springer, 2003.
5. G. Behrmann, P. Bouyer, K.G. Larsen and R. Pelánek. Lower and upper bounds in zone-based abstractions of timed automata. *STTT*, 8(3):204–215, 2006.
6. G. Bombara, C.I. Vasile, F. Penedo, H. Yasuoka and C. Belta. A decision tree approach to data classification using signal temporal logic. In A. Abate and G.E. Fainekos, editors, *HSCC 2016, Vienna, Austria, April 12-14, 2016*, pp. 1–10. ACM, 2016.
7. R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
8. S. Chen, O. Sokolsky, J. Weimer and I. Lee. Data-driven adaptive safety monitoring using virtual subjects in medical cyber-physical systems: A glucose control case study. *JCSE*, 10(3), 2016.
9. D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings*, vol. 407 of *Lecture Notes in Computer Science*, pp. 197–212. Springer, 1989.
10. D. DSouza and R. Matteplackel. A clock-optimal hierarchical monitoring automaton construction for mitl. Tech. rep., 2013.
11. S. Faro and T. Lecroq. The exact online string matching problem: A review of the most recent results. *ACM Comput. Surv.*, 45(2):13:1–13:42, 2013.
12. T. Ferrère, O. Maler, D. Nickovic and D. Ulus. Measuring with timed patterns. In D. Kroening and C.S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, vol. 9207 of *Lecture Notes in Computer Science*, pp. 322–337. Springer, 2015.
13. F. Franek, C.G. Jennings and W.F. Smyth. A simple fast hybrid pattern-matching algorithm. *J. Discrete Algorithms*, 5(4):682–695, 2007.
14. F. Herbretreau, B. Srivathsan and I. Walukiewicz. Efficient emptiness check for timed büchi automata. In T. Touili, B. Cook and P.B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, vol. 6174 of *Lecture Notes in Computer Science*, pp. 148–161. Springer, 2010.
15. H. Ho, J. Ouaknine and J. Worrell. Online monitoring of metric temporal logic. In B. Bonakdarpour and S.A. Smolka, editors, *RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, vol. 8734 of *Lecture Notes in Computer Science*, pp. 178–192. Springer, 2014.
16. B. Hoxha, H. Abbas and G.E. Fainekos. Benchmarks for temporal logic requirements for automotive systems. In G. Frehse and M. Althoff, editors, *1st and 2nd International Workshop on Applied verification for Continuous and Hybrid Systems, ARCH@CPSWeek 2014, Berlin, Germany, April 14, 2014 / ARCH@CPSWeek 2015, Seattle, WA, USA, April 13, 2015.*, vol. 34 of *EPiC Series in Computing*, pp. 25–30. EasyChair, 2014.

17. X. Jin, J.V. Deshmukh, J. Kapinski, K. Ueda and K.R. Butts. Powertrain control verification benchmark. In M. Fränzle and J. Lygeros, editors, *HSCC'14, Berlin, Germany, April 15-17, 2014*, pp. 253–262. ACM, 2014.
18. A. Kane. *Runtime monitoring for safety-critical embedded systems*. PhD thesis, PhD thesis, Carnegie Mellon University, USA, 2015.
19. A. Kane, O. Chowdhury, A. Datta and P. Koopman. A case study on runtime monitoring of an autonomous research vehicle (ARV) system. In E. Bartocci and R. Majumdar, editors, *RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, vol. 9333 of *Lecture Notes in Computer Science*, pp. 102–117. Springer, 2015.
20. D.R. Kini, S.N. Krishna and P.K. Pandya. On construction of safety signal automata for $mitl[U, S]$ using temporal projections. In U. Fahrenberg and S. Tripakis, editors, *FORMATS 2011, Aalborg, Denmark, September 21-23, 2011. Proceedings*, vol. 6919 of *Lecture Notes in Computer Science*, pp. 225–239. Springer, 2011.
21. D.E. Knuth, J.H.M. Jr. and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.
22. O. Maler, D. Nickovic and A. Pnueli. From MITL to timed automata. In E. Asarin and P. Bouyer, editors, *FORMATS 2006, Paris, France, September 25-27, 2006, Proceedings*, vol. 4202 of *Lecture Notes in Computer Science*, pp. 274–289. Springer, 2006.
23. The MathWorks, Inc., Natick, MA, USA. *Simulink User's Guide*, 2015.
24. D. Nickovic and N. Piterman. From mtl to deterministic timed automata. In K. Chatterjee and T.A. Henzinger, editors, *FORMATS 2010, Klosterneuburg, Austria, September 8-10, 2010. Proceedings*, vol. 6246 of *Lecture Notes in Computer Science*, pp. 152–167. Springer, 2010.
25. Pure Programming Language. <https://purelang.bitbucket.io>.
26. F. Raimondi, J. Skene and W. Emmerich. Efficient online monitoring of web-service slas. In M.J. Harrold and G.C. Murphy, editors, *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pp. 170–180. ACM, 2008.
27. T. Reinbacher, M. Függer and J. Brauer. Runtime verification of embedded real-time systems. *Formal Methods in System Design*, 44(3):203–239, 2014.
28. D. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.
29. D. Ulus. Montre: A tool for monitoring timed regular expressions. *CoRR*, abs/1605.05963, 2016.
30. D. Ulus, T. Ferrère, E. Asarin and O. Maler. Timed pattern matching. In A. Legay and M. Bozga, editors, *FORMATS 2014, Florence, Italy, September 8-10, 2014. Proceedings*, vol. 8711 of *Lecture Notes in Computer Science*, pp. 222–236. Springer, 2014.
31. D. Ulus, T. Ferrère, E. Asarin and O. Maler. Online timed pattern matching using derivatives. In M. Chechik and J. Raskin, editors, *TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, vol. 9636 of *Lecture Notes in Computer Science*, pp. 736–751. Springer, 2016.
32. M. Waga, T. Akazaki and I. Hasuo. A boyer-moore type algorithm for timed pattern matching. In M. Fränzle and N. Markey, editors, *FORMATS 2016, Quebec, QC, Canada, August 24-26, 2016, Proceedings*, vol. 9884 of *Lecture Notes in Computer Science*, pp. 121–139. Springer, 2016.
33. M. Waga, I. Hasuo and K. Suenaga. Code that Accompanies "Efficient Online Timed Pattern Matching by Automata-Based Skipping.". <https://github.com/MasWag/timed-pattern-matching>.
34. B.W. Watson and R.E. Watson. A boyer-moore-style algorithm for regular expression pattern matching. *Sci. Comput. Program.*, 48(2-3):99–117, 2003.

Algorithm 4 Detail of our FJS-type algorithm for timed pattern matching**Require:** A timed word $w = (\bar{a}, \bar{\tau})$, and a timed automaton $\mathcal{A} = (\Sigma, S, S_0, C, E, F)$.**Ensure:** $\bigcup Z$ is the match set $\mathcal{M}(w, \mathcal{A})$ in Def. 2.3.

```

1:  $n \leftarrow 1$  ▷  $n$  is the position in  $w$  of the head of the current matching trial
2:  $CurrConf \leftarrow \emptyset$ ;  $Z \leftarrow \emptyset$ 
3: while  $n \leq |w| - m + 2$  do
4:   while  $\forall \bar{\tau} \in L'. \bar{a}_{n+m-2} \neq a'$  where  $\bar{\tau}_{m-2} \xrightarrow{a'} \bar{\tau}_{m-1}$  do ▷ Try to match the tail of  $L'$ 
5:      $n \leftarrow n + \Delta(\bar{a}_{n+m-1})$  ▷ Quick Search-type skipping
6:     if  $n > |w| - m + 2$  then return
7:      $CurrConf \leftarrow \{(s, \rho_\emptyset, [\tau_{n-1}, \tau_n]) \mid s \in S_0\}$ 
8:     for  $n' \in \{n, n+1, \dots, |w|\}$  do ▷ We try matching in the same way as [32]
9:        $NextConf \leftarrow \emptyset$ 
10:      for  $(s, \rho, T) \in CurrConf$  do
11:        for  $(s', a_n, \lambda, \delta) \in E$  do
12:           $T' \leftarrow \{t_0 \in T \mid \text{eval}(\rho, \tau_n, t_0) \models \delta\}$ 
13:          if  $T' \neq \emptyset$  then
14:             $\rho' \leftarrow \rho$ 
15:            for  $x \in \lambda$  do
16:               $\rho' \leftarrow \text{reset}(\rho', x, \tau_n)$ 
17:               $NextConf \leftarrow NextConf \cup (s', \rho', T')$ 
18:              for  $s_f \in F, (s', s_f, \$, \lambda', \delta') \in E$  do
19:                 $T'' \leftarrow (\tau_{n'}, \tau_{n'+1})$ 
20:                 $Z \leftarrow Z \cup \text{solConstr}(T', T'', \rho', \delta')$ 
21:            if  $NextConf = \emptyset$  then break
22:             $CurrConf \leftarrow NextConf$ 
23:      for  $k \in \{n+1, \dots, n + \max\{\beta(s) \mid (s, \rho, T) \in CurrConf\} - 1\}$  do
24:        ▷ Matching trial stacks at the states  $\{s \mid (s, \rho, T) \in CurrConf\}$ 
25:        for  $s \in S_0, s_f \in F, (s, s_f, \$, \rho, \delta) \in E$  do
26:           $Z \leftarrow Z \cup \text{solConstr}([\tau_{k-1}, \tau_k], (\tau_{k-1}, \tau_k], \rho, \delta)$ 
27:         $n \leftarrow n + \max\{\beta(s) \mid (s, \rho, T) \in CurrConf\}$  ▷ KMP-type skipping

```

A Detailed Pseudocode of Our FJS-type Algorithm for Timed Pattern Matching

Definition A.1 (eval, reset, solConstr) Let a pattern timed automaton be $\mathcal{A} = (\Sigma, S, S_0, C, E, F)$. For a partial function $\rho : C \rightarrow \mathbb{R}_{>0}$ and $t, t_0 \in \mathbb{R}_{>0}$, the clock interpretation $\text{eval}(\rho, t, t_0) : C \rightarrow \mathbb{R}_{\geq 0}$ is $\text{eval}(\rho, t, t_0)(x) = t - \rho(x)$ (if $\rho(x)$ is defined) and $\text{eval}(\rho, t, t_0)(x) = t - t_0$ (otherwise). For a partial function $\rho : C \rightarrow \mathbb{R}_{>0}$, $t_r \in \mathbb{R}_{>0}$ and $x \in C$, $\text{reset}(\rho, x, t_r) : C \rightarrow \mathbb{R}_{>0}$ is the following partial function : $\text{reset}(\rho, x, t_r)(x) = t_r$; and $\text{reset}(\rho, x, t_r)(y) = \rho(y)$ for each $y \in C \setminus \{x\}$. (The latter is Kleene's equality between partial functions, to be precise.) For intervals $T, T' \subseteq \mathbb{R}_{>0}$, a partial function $\rho : C \rightarrow \mathbb{R}_{\geq 0}$, and a clock constraint $\delta \in \Phi(C)$ (§2.1), we define $\text{solConstr}(T, T', \rho, \delta) = \{(t, t') \mid t \in T, t' \in T', \text{eval}(\rho, t', t) \models \delta\}$.

The detail of our FJS-type algorithm for timed pattern matching is in Algorithm 4.

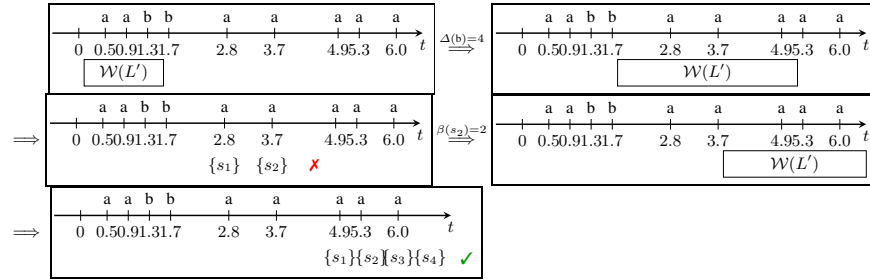


Fig. 21. Our FJS-type algorithm for pattern matching, for the example in Fig. 1

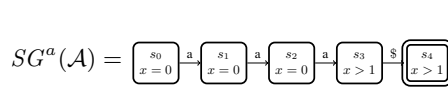


Fig. 22. The zone automaton $SG^a(\mathcal{A})$ for \mathcal{A} in Fig. 1

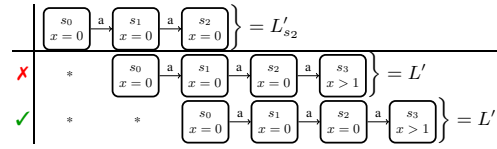


Fig. 23. Table for $\beta(s_2) = 2$

B Our FJS-Type Timed Pattern Matching Problem, Illustrated

Let us look at the example in Fig. 1. The zone automaton $SG^a(\mathcal{A})$ is in Fig. 22; the execution of our algorithm is illustrated in Fig. 21.

The first configuration in Fig. 21 means we are after possible matches that start at $t \in [0, 0.5)$. With $m = 4$ (the length of the shortest accepted word), we try matching of the third target character $\bar{a}_{m-1} = \bar{a}_3 = b$ with the tail of every length-3 prefix of $L(\mathcal{A})$ using the zone automaton $SG^a(\mathcal{A})$ in Fig. 22. The trial fails and we invoke Quick Search-type skipping $\Delta(\bar{a}_4 = b)$. Since $\bar{a}_4 = b$ does not appear in any transition of $SG^a(\mathcal{A})$, we can skip four events and reach the second configuration where we look for potential matches that start at $t \in [1.7, 2.8)$.

We again try matching from the tail. This time it succeeds because $\bar{a}_7 = a$ appears in the third character of a word accepted by $SG^a(\mathcal{A})$. Then we move to Line 7 of Algorithm 4 where we try matching from left to right. After the trial stacks at $s_2 \in S$, we invoke the KMP-type skipping.

The KMP-type skip value $\beta(s_2)$ is computed as shown in Fig. 23. Here it is much more intricate how to decide \checkmark or \times , i.e. if the “prefix” on the top (L'_{s_2}) matches the shifts of L' below. Previously for string or (untimed) pattern matching we just compared characters (Fig. 3 & 7); here the question is if there exists a timed word that causes both a transition in the prefix (on the top) and the corresponding transition in a shift (below). For this purpose we employ the zone automaton $SG^a(\mathcal{A} \times \mathcal{A})$ of the product timed automaton $\mathcal{A} \times \mathcal{A}$. For example, the shift by one position does not match (\times) in Fig. 23 because there is no transition $\begin{matrix} (s_0, s_1) \\ x = x' = 0 \end{matrix} \xrightarrow{a} \begin{matrix} (s_1, s_2) \\ x = x' = 0 \end{matrix}$ in $SG^a(\mathcal{A} \times \mathcal{A})$.

In the fourth configuration, we try matching from $t \in [3.7, 4.9)$. We again try matching from the tail; it succeeds; we try matching from left to right; and we find a matching $\{(t, t') \mid t \in [3.7, 3.9), t' \in (6.0, \infty)\}$.

C Correctness of Our FJS-Type Timed Pattern Matching Algorithm

Theorem C.1 (Correctness of Δ and β) Let $Opt(n) = \min\{i \in \mathbb{Z}_{>0} \mid \exists t \in [\tau_{n+i-1}, \tau_{n+i}), t' \in (t, \infty). (t, t') \in \mathcal{M}(w, \mathcal{A})\}$. For $n \in [1, |w|]$, we have both $Opt(n) \geq \Delta(\bar{a}_{n+m-1})$ and $Opt(n) \geq \max_{s \in S'} \beta(s)$ where ν_0 is the clock valuation assigning 0 for any $x \in C$, $n' = \max\{n' \in [1, |w|] \mid \exists s_0 \in S_0, s \in S, \nu \in (\mathbb{R}_{\geq 0})^C. (s_0, \nu_0) \xrightarrow{w(n, n')} (s, \nu)\}$ and $S' = \{s \in S \mid \exists s_0 \in S_0, \nu \in (\mathbb{R}_{\geq 0})^C. (s_0, \nu_0) \xrightarrow{w(n, n')} (s, \nu)\}$.

Proof. When $Opt(i) > m$, both $Opt(i) \geq \Delta(w(i+m-1))$ and $Opt(i) \geq \max\{\beta(s) \mid (s, \rho, T) \in Conf(i, j)\}$ hold because for any $a \in \Sigma$ and $s \in S$, we have $m+1 \geq \Delta(a)$ and $m+1 \geq \beta(s)$. Assume $Opt(i) \leq m$ in the following. Let $L_{-\$}(\mathcal{A})$ be $\{w(1, |w|-1) \mid w \in L(\mathcal{A})\}$.

The membership of a timed word segment leads the membership in the approximated languages, as follows.

$$\begin{aligned}
& \exists t \in [i+n-1, i+n), t' \in (t, \infty). (t, t') \in \mathcal{M}(w, \mathcal{A}) \\
\iff & \exists t \in [i+n-1, i+n), t' \in (t, \infty). w|_{(t, t')} \in L(\mathcal{A}) \\
& \Rightarrow \exists t \in [i+n-1, i+n), t' \in (t, \infty), k \in [i+n-1, |w|]. \\
& \quad (w(i+n, k) - t) \circ (\$, t') \in L(\mathcal{A}) \\
& \Rightarrow \exists t \in [i+n-1, i+n), k \in [i+n-1, |w|]. \\
& \quad (w(i+n, k) - t) \in L_{-\$}(\mathcal{A}) \\
& \Rightarrow \exists t \in [i+n-1, i+n). (w(i+n, |w|) - t) \in L_{-\$}(\mathcal{A}) \cdot (\Sigma \times \mathbb{R}_{>0})^* \\
& \Rightarrow (w(i, |w|) - \tau_i) \in (\Sigma \times \mathbb{R}_{>0})^n \cdot L_{-\$}(\mathcal{A}) \cdot (\Sigma \times \mathbb{R}_{>0})^* \\
& \Rightarrow (w(i, |w|) - \tau_i) \in (\Sigma \times \mathbb{R}_{>0})^n \cdot \mathcal{W}(L') \cdot (\Sigma \times \mathbb{R}_{>0})^*
\end{aligned}$$

We have $Opt(i) \geq \Delta(w(i+m-1))$ because of the follows.

$$\begin{aligned}
& (w(i, |w|) - \tau_i) \in (\Sigma \times \mathbb{R}_{>0})^n \cdot \mathcal{W}(L') \cdot (\Sigma \times \mathbb{R}_{>0})^* \\
& \Rightarrow (w(i, i+m-1) - \tau_i) \cdot (\Sigma \times \mathbb{R}_{>0})^* \cap \\
& \quad (\Sigma \times \mathbb{R}_{>0})^n \cdot \mathcal{W}(L') \cdot (\Sigma \times \mathbb{R}_{>0})^* \neq \emptyset \\
& \Rightarrow (\Sigma \times \mathbb{R}_{>0})^m \cdot w(i+m-1) \cdot (\Sigma \times \mathbb{R}_{>0})^* \cap \\
& \quad (\Sigma \times \mathbb{R}_{>0})^n \cdot \mathcal{W}(L') \cdot (\Sigma \times \mathbb{R}_{>0})^* \neq \emptyset
\end{aligned}$$

Similarly, we have $Opt(i) \geq \max\{\beta(s) \mid (s, \rho, T) \in Conf(i, j)\}$ because of the follows.

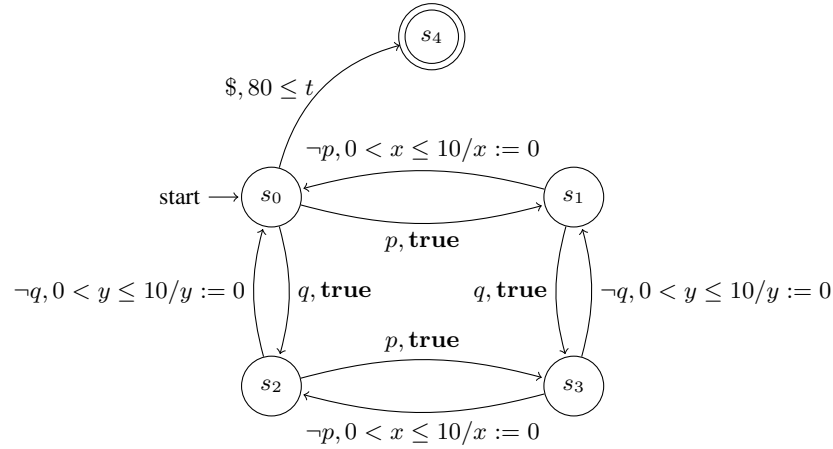
$$\begin{aligned}
& (w(i, |w|) - \tau_i) \in (\Sigma \times \mathbb{R}_{>0})^n \cdot \mathcal{W}(L') \cdot (\Sigma \times \mathbb{R}_{>0})^* \\
& \Rightarrow \forall (s, \rho, T). \mathcal{W}(L_s) \cdot (\Sigma \times \mathbb{R}_{>0})^* \cap (\Sigma \times \mathbb{R}_{>0})^n \cdot \mathcal{W}(L') \cdot (\Sigma \times \mathbb{R}_{>0})^*
\end{aligned}$$

□

Table 7. The duration of preprocessing (ms). The timeout is set to three minutes.

	BM (region)	BM (zone)	FJS (zone)
SIMPLE	1.09e-03	2.53e-03	3.46e-03
LARGE CONSTRAINTS	Timeout	1.02e-03	1.05e-03
TORQUE	1.74e+00	4.25e-01	2.09e-01
SETTLING	1.69e+04	4.60e-03	5.40e-03
GEAR	1.00e-03	4.47e-03	1.00e-03
ACCEL	Timeout	2.95e-02	1.00e-03

D The pattern timed automaton in LARGE CONSTRAINTS

**Fig. 24.** The pattern timed automaton in LARGE CONSTRAINTS

E Optimization of Preprocessing for Zone Abstraction and Skip Value Computation

In Table 7 is how long our preprocessing takes for each of our benchmark problems. We see that our implementation is efficient in the preprocessing stage; this is largely due to our memorization technique in which we reuse parts of zone automata.

For reference we also present results for *region-based* abstraction [1]: though equivalent in terms of finiteness, zones give more efficient abstraction than regions. In our previous work [32] we used regions in place of zones, and that posed a bottleneck, as we can see in Table 7.