MONTRE: A Tool for Monitoring Timed Regular Expressions

Dogan Ulus
Verimag, Université Grenoble-Alpes,
Grenoble, France

***Abstract*—We present MONTRE, a monitoring tool to search patterns specified by timed regular expressions in symbolic behaviors of real-time systems. We use timed regular expressions as a compact, natural and highly-expressive specification language for monitoring applications involving timing constraints. Our tool essentially incorporates online and offline timed pattern matching algorithms so it is capable of finding all occurrences of a given pattern over both logged and streaming behaviors. Executed standalone MONTRE provides a user experience similar to command line utilities such as `grep` and friends. Besides it is designed to work nicely with other tools via standard interfaces to perform more complex and versatile verification and validation tasks. As the first of its kind, we think MONTRE will enable a new line of inquiries and techniques to analyze real-time behaviors.**

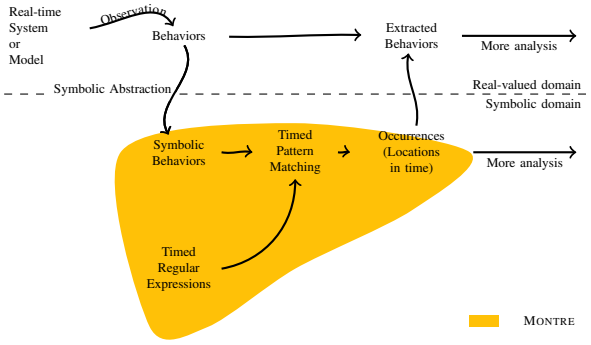

Fig. 1. The work flow and extent of the monitoring tool MONTRE

I. INTRODUCTION

Real-time systems are systems continuously interacting with other systems and the environment through channels with certain timing requirements. For the correct operation of real-time systems, from system to physical levels, the timing of occurrences (events, states, processes) can be critical such that a long pulse may have different meaning than a short pulse in a communication protocol or rapid stirring of a chemical tank may cause different results than stirring slowly. Compared to real-time temporal logics [1], timed regular expressions (TRE) [2, 3], an extension of regular expressions with timing constraints, can formally express such patterns requiring both qualitative and quantitative temporal reasoning in a more compact and natural way.

Regular expressions is one of success stories of computer science that actually solves real problems with an extremely clean and elegant theory. Originally introduced in 50s [10] they were meant to represent patterns in biological behaviors and not long after that the close connection between regular expressions and automata theory was shown in various aspects. On the practical front searching patterns in text using regular expressions is immensely popular. Starting from Ken Thompson's algorithm [13], *regex engines* have been written for every possible platform and for every taste and enhanced (and sometimes bloated) by many features over years. Furthermore regular expressions as a way to specify sequences of events/states naturally fit monitoring purposes apart from text search [12]. Today existing digital assertion languages such as Property Specification Language (PSL) [8] and SystemVerilog Assertions (SVA) [5] extensively use regular expressions.

Regular expressions on real-time behaviors are studied to enhance runtime monitoring typically performed by tools using temporal logics such as [6, 11]. Havlicek and Little in [9] propose a real-time extension of SVA regular expressions similar to TRE. In [15] Ulus et al. introduce an offline timed pattern matching algorithm to search and report all segments satisfied by a TRE in a continuous-time input signal. In this paper we summarize our contributions as below:

- We present the tool MONTRE[1], which implements online and offline timed pattern matching algorithms [14, 15].
- We extend timed pattern matching by some practical features such anchors and the Boolean layer.

MONTRE is a monitoring tool designed to do just one thing: searching occurrences of the pattern given in continuous-time symbolic behaviors. In Figure 1 we illustrate the work flow and extent of MONTRE. It is not an assertion checker in the sense that it does not reason about the correctness of the behavior but it returns a set of locations where the pattern occurred. Then surely another tool can process the resulting set, reason about the correctness or just simply visualize occurrences on the behavior. Similarly MONTRE accepts symbolic behaviors that may be obtained by symbolic abstraction of either sensor measurements or numerical simulations. We consider such activities as preprocessing and post-processing for MONTRE. These activities are no doubt important for the quality and effectiveness of MONTRE but they are orthogonal to what we do and they usually have to be application-dependent.

In the following we explain symbolic behaviors in Section II and timed regular expressions in Section III. We give an overview of algorithms in Section IV and we evaluate our implementation in Section V then we discuss the results.

[1] Available at http://doganulus.com/montre

II. Symbolic Behaviors of Real-time Systems

Symbolic treatment of numerical values coming from observations enables the use of formalisms such as regular expressions over such data. By *symbolic* here we understand that the numerical range (or the state-space of the system) is partitioned according to a scheme and each partition is associated by a unique symbol. Then behaviors of the system can be represented by sequences of symbols. Such symbolic representation provides a very useful level of abstraction as well as offering efficiency and intuitiveness. For example we usually use the symbols *hot*(h), *warm*(w) or *cold*(c) for temperature rather than precise numerical values in our daily communication. A week of symbolic temperature observations such as {h;h;h;w;w;c;w} captures the essential behavior.

An important question in symbolic behaviors is the sampling rate of observations, or other way around, the duration of each symbol. Choosing a fixed sampling rate is the most common solution and a review of such digital symbolic transformation techniques can be found in [7]. However real-time monitoring systems often require very high sampling rates not to miss something happened between samples therefore digital symbolic transformation may cause a repetition of the same symbol many times known as a *stuttered* signal. Such redundancy affects the quality and efficiency of higher-level analysis of real-time systems and the dependence on sampling rate is not desirable. Therefore timed formalisms (either event-based or state-based) is often preferred in modeling and specification of real-time systems. For our tool we accept state-based continuous-time symbolic behaviors (equivalently known as signals or timed state sequences) as the input data and note that digital and event-based symbolic behaviors can be converted into state-based format. In state-based models symbols appear on the timeline continuously and stuttered symbols can be merged into one by simply associating a duration value. One may think it as a sampling with infinitesimal periods and grouping stuttered symbols accordingly. For illustration, in Figure 2, we partition the value-axis into b and t regions and we show (i) a digital symbolic behavior {b;b;b;b;t;t;t} obtained from a continuous signal and (ii) a state-based continuous-time symbolic behavior $\{(d_1, \mathtt{b});(d_2, \mathtt{t})\}$ where repeated symbols are merged and associated with corresponding duration value.

In many cases we need to monitor more than one observables at the same time. One common approach when representing concurrent behaviors is to use sets of symbols instead of individual symbols as values. Then we can construct a synchronized behavior by making sets of symbols associated

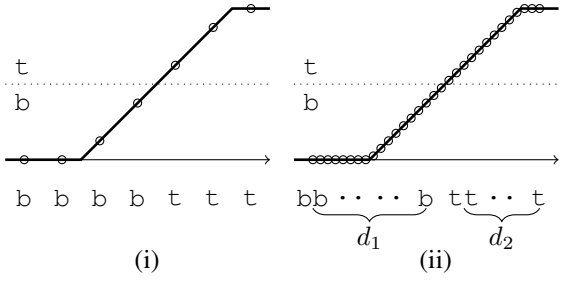

Fig. 2. Stuttered symbols can be merged and associated with a duration.

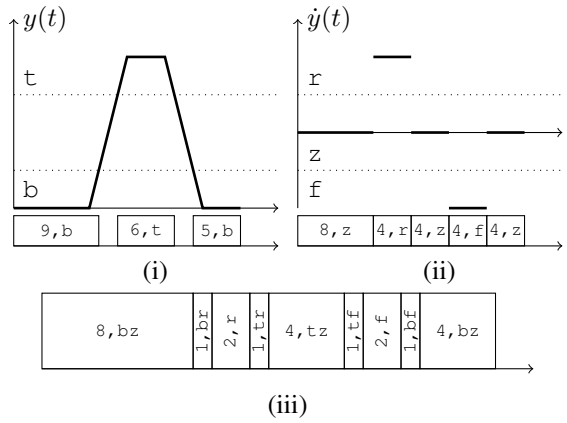

Fig. 3. Synchronization of two symbolic behaviors

with durations. For example, in Figure 3, we depict an observation of a real variable $y(t)$ and its numerical derivative $\dot{y}(t)$ side-by-side and we partition value domains and labeled them with symbols t,b and r,z,f as shown, respectively. In this paper we denote symbols by small letters and a set of symbols by strings such that a set $S = \{\mathtt{p},\mathtt{q}\}$ is simply written as pq. Then the synchronized behavior would be (8,bz);(1,br);(2,r);(1,tr);(2,tz);(1,tf);(2,f);(1,bf);(4,bz) shown graphically in Figure 3-iii.

III. Timed Regular Expressions

Timed regular expressions is an extension of regular expressions allowing us to specify patterns with real-time constraints. An atom of a timed regular expression is a propositional symbol related to either a predicate over real-time observables or a Boolean combination of such predicates. The operators (!), (||), (&&) on atomic expressions carry their usual Boolean meanings. The below are some examples of atomic expressions for timed regular expressions:

$$\mathtt{p} := \mathit{holds}(b), \qquad \mathtt{q} := \mathit{sin}(t) < 0.5 \qquad \mathtt{p \ \&\& \ !q}$$

Before proceeding, here let us point out a crucial difference between predicates over observables and the related atomic expressions. Predicates are to perform symbolic abstraction and their truth values are evaluated over time points. However, atomic expressions are evaluated over time intervals and we say that an atomic expression occurs on interval (t, t') if the related predicate holds from t to t' continuously. For example consider an atomic expression $\mathtt{p} := x < 5$ where the predicate evaluates true for all points in $[0, 3]$. Then we can say that p occurs on an interval (t, t') satisfying $0 \leq t \leq t' \leq 3$.

Complex expressions are built from other expressions by using TRE operators, namely concatenation (;), time restriction (%), alternation (|), intersection (&) and zero-or-more repetition ($\star$). Further we add one-or-more repetition (+) and two anchoring (<: and :>) operators to the set. Typically parentheses are used to group expressions. In the following we denote atomic expressions by stylized small letters such as p, q, r while stylized capital letters such as E, F denote arbitrary timed regular expressions. For instance, the expression E below is a complex expression forming a syntax tree where atomic expressions lie at leaves.

```
E := p;((q;r)%(2,5)) & ((p;q)%(3,6));r
```

Indeed E is an example of non-trivial properties frequently seen in timed systems research. We will look into it more closely when talking about the intersection. Now we start describing the meaning of each operator.

1) Concatenation: We use the concatenation operator, the semi-colon (;), to specify temporally ordered patterns such as the execution of a procedure step-by-step. For two timed regular expressions E and F, the concatenation E;F describes a pattern such that E *occurs first and then* F *occurs*. More precisely, we say that E;F occurs on a time-interval (t, t') if there exists a point t'' in between such that E occurs on (t, t'') and F occurs on (t'', t). For example, consider an expression p;q composed of two atomic expressions p and q over some observables x and y as in Figure 4. Here we can read the pattern such that there is an occurrence of x greater than 3 and then an occurrence of y less than 2 follows it. In the figure it is seen that p occurs on $(1, 4)$ and then q occurs on $(4, 6)$ therefore p;q occurs on $(1, 6)$.

2) Time restriction: Restricting duration of patterns is arguably the most characteristic feature of timed regular expressions. Time restriction operator, the percent sign (%), is a binary infix operator such that it accepts an expression E at left hand side and a numerical range (i,j) at right hand side for the duration specification such that (E)%(i,j). For example a time restricted expression (p;q;p)%(8,10) specifies a pattern (p;q;p) with a total duration between 8 and 10 time units. In Figure 5 we depict three occurrences (#1, #2 and #3) of the pattern (p;q;p) with durations 8, 9 and 11, respectively. Since the occurrence #3 is simply too long for the pattern F := (p;q;p)%(8,10) and we can not say that there is an occurrence of F on the interval $(1, 12)$ while such are there on intervals $(2, 10)$ and $(2, 11)$.

Moreover duration specifications can be freely nested to express more complex timing requirements. Indeed we expect that expressions with a few nesting levels would be the most used class of timed regular expressions. For example consider an expression specifying an occurrence of {E;F;G} where the duration of F is between 2 and 5 time units while total duration is less than 8 time units as such:

```
(E;(F)%(2,5);G)%(0,8)
```

This type of requirements is abundant for real-time systems where each subprocess/subtask/subgoal has own deadlines and the main process/task/goal has a global time budget. Also note that these requirements cannot be expressed easily and elegantly using point-based temporal logics, which are very popular both in industry and academia.

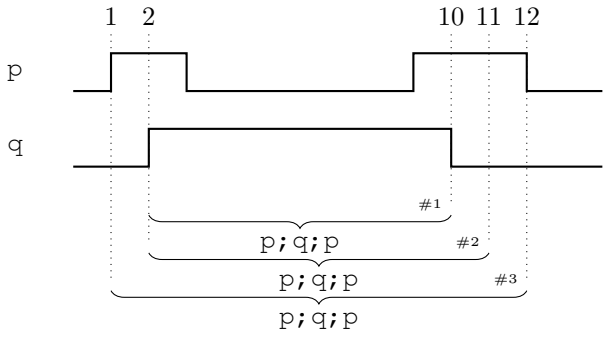

Fig. 5. Time restriction operation filters occurrences according to the duration.

The zero and the underscore sign (_) can be used for the lower and upper bound respectively if they are unbounded. Obviously, if both sides are unbounded, such a (pseudo) restricted expression is equivalent to the expression inside such that (E)%(0,_) ≡ E. Besides writing nested restricted expressions with infeasible bounds is legal but meaningless. For example (E;(F)%(4,6);G)%(0,3) is a valid expression but it won't match anything under Newtonian notion of time.

Lastly we remark that time restriction is a first-class operation in TRE and it does not rely on the enumeration of time points therefore the expressions (E)%(8,10) and (E)%(800,1000) is structurally and computationally equivalent. Similar functioning operators in digital assertion languages like PSL's ranged repetition operator [*i:j] simply expand the formula as much as specified numbers or they have to perform some tricks in the implementation.

3) Alternation: The pipe sign (|) specifies alternative ways of a pattern to be satisfied therefore we say an expression E|F occurs on an interval if E or F occurs on the interval. An important caveat here is that the alternation operation should not be confused with the Boolean *or* operator (||) that can be applied over atomic expressions. For example consider two expressions p||q and p|q in Figure 6. The expression p|q does not occur on the interval $(3, 11)$ because it requires that either p or q occurs on the interval where both are not satisfied. On the contrary the expression p||q is an atomic expression stating a Boolean or operation initially over p and q then the

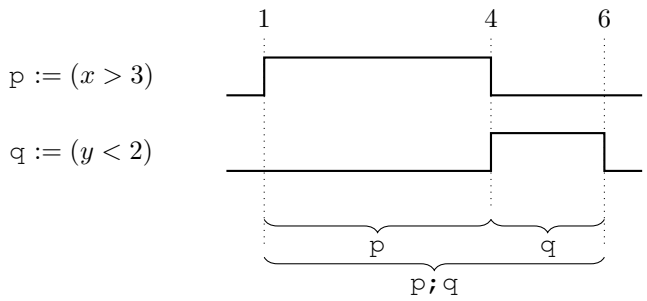

Fig. 4. Concatenation operation joins two occurrences.

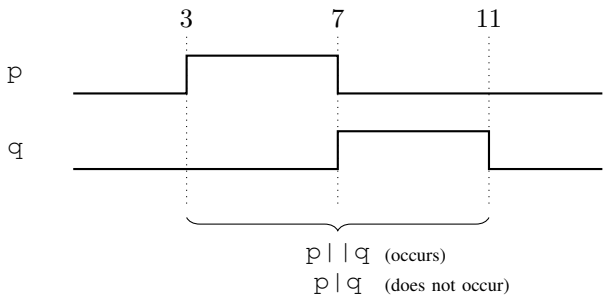

Fig. 6. Alternation and disjunction are different operations.

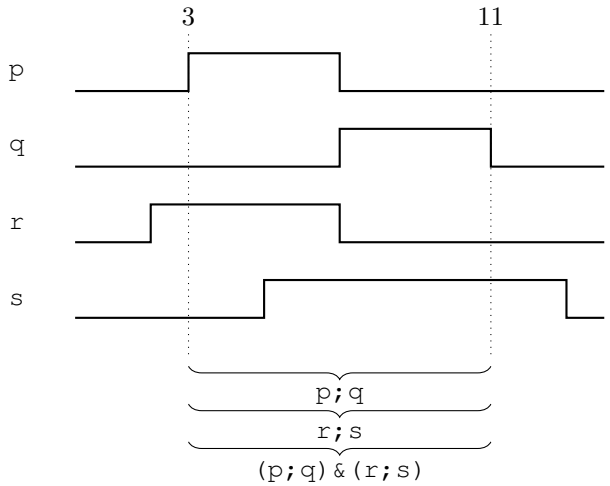

Fig. 7. Intersection operator requires that begin and end points are equal.

atomic expression `p||q` is evaluated over resulting behavior.

4) Intersection: The intersection operator (`&`) specifies parallel composition of two patterns such that we say an expression `E&F` occurs on an interval if `E` and `F` occurs on the interval concurrently. Therefore the intersection requires that begin points, end points and duration values of the occurrences of `E` and `F` are equal. Moreover we can use intersection to specify overlapping timing constraints, which cannot be expressed by nesting only. For example consider an expression `((p;q)%(6,8);r)&(p;(q;r)%(6,8))` specifying a pattern `p;q;r` where overlapped segments `p;q` and `q;r` requires to last between 6 and 8 time units. In Figure 8 we depict a pattern `p;q;r` occurs on $(2, 12)$ which satisfies both timing constraints over segment `p;q` and `q;r`.

5) Repetition: We implement two repetition operators, the star ($\star$) and the plus (+). The expressions `E`$\star$ and `E+` specifies consecutive occurrences of `E` with a difference that `E`$\star$ can be skipped (zero repetition) but `E+` requires at least one occurrence of `E`. For example, in Figure 9, the pattern `p;(q;r)`$\star$ occurs on intervals $\#1$, $\#2$ and $\#3$ but the pattern `p;(q;r)+` occurs on $\#2$ and $\#3$ but not on $\#1$.

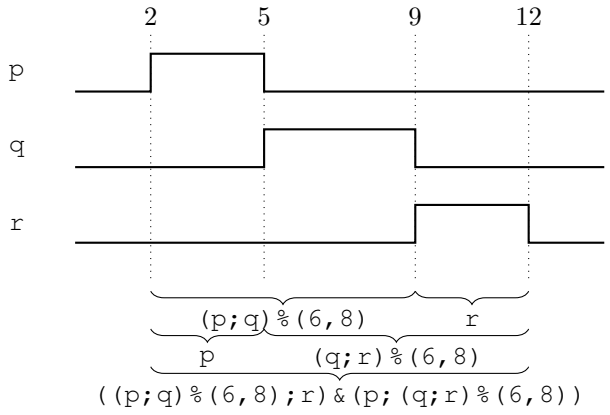

Fig. 8. Overlapping timing constraints can be expressed using intersection.

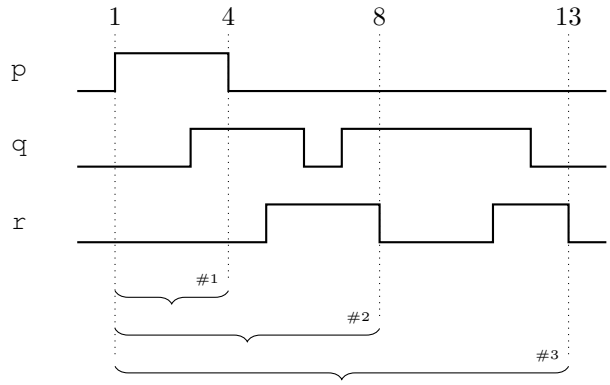

Fig. 9. Repetition operators.

6) Anchors on atoms: Until now we did not put any restriction over where occurrences can begin and where they can end. However sometimes it is useful to anchor the beginning and the end points of occurrences to the transition points of corresponding atoms. Inspired from *anchors* in practical regular expressions and *rose/fell* operators of PSL we introduce prefix (`<:`) and postfix (`:>`) operators to anchor the beginning and the end points of occurrences to rising and falling points of the symbol, respectively. For example, in Figure 10, the pattern `<:p` does not occur on $(5, 8)$ because the begin point 5 is not a rising point for the symbol `p` while it holds on $(1, 4)$. Similarly `p:>` does not occur on $(5, 8)$ because the end point 8 is not a falling point for the symbol `p` while it holds on $(10, 13)$. Besides it is possible to write `<:p:>` to anchor from both sides then it will only occur on maximal intervals of `p` but none of their sub-intervals.

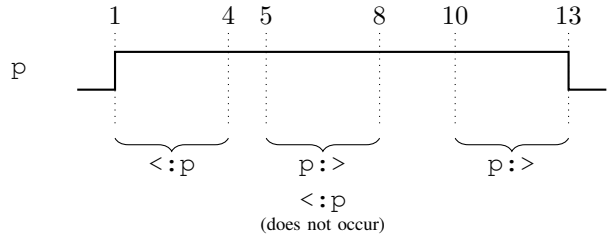

Fig. 10. Anchor operators. `<:p` and `p:>` occur on $(1, 4)$ and $(10, 13)$, respectively, however these expressions do not occur on $(5, 8)$.

IV. Overview of Algorithms

The tool MONTRE consists of online and offline timed pattern matching algorithms. To store all occurrences (or incomplete occurrences in case of online algorithm), both algorithms use the idea that an occurrence on an interval (t_1, t_2) can be represented as a point on a two-dimensional plane shown in Figure 11-i. Then the set of occurrences over continuous-time forms enclosed regions on the plane called (two-dimensional) zones, which are basic data objects of timed pattern matching. Zones are simple geometric objects defined by six (possibly strict) inequalities as shown in Figure 11-ii but the number of zones can be high and operations between zone sets can be prohibitive with a naive complexity of $\mathcal{O}(n^2)$ where n is the size of the input behavior. However, when zone sets

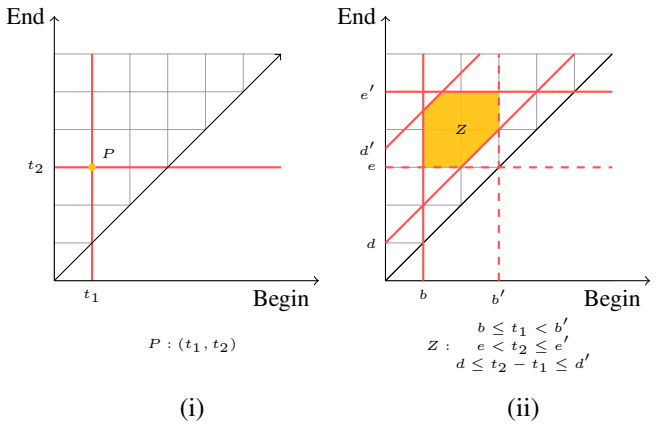

Fig. 11. (i) An occurrence as a point (ii) A set of occurrences as a zone

are sorted to exploit intrinsic ordering of data, which does exist in realistic cases, and the complexity is reduced to $\mathcal{O}(nlogn)$ in average. This is still a pessimistic result since we mostly deal with (either intrinsic or explicit) time-restricted patterns in practice. In case of time-restricted patterns the effective time window of search is constant during one operation, which means that the number of zones to be interacted would remain constant. This gives us an average complexity of one operation is linear under assumption of realistic data and time-restricted patterns while the worst-case is always $\mathcal{O}(n^2)$.

The offline algorithm [15] is a recursive computation over the syntax tree of the expression and it requires one pass over data per operation (except repetition operations which require several passes). The upper bound for the number of iterations required per repetition operator is a function of n therefore the worst-case complexity of offline algorithm is $\mathcal{O}((m+kn)\cdot n^2)$ where m is the size of the expression and k is the number of repetition operators.

In online timed pattern matching algorithm [14], however, we assume that the symbolic behavior $\mathtt{w}_{0..n} = \mathtt{w}_1; \mathtt{w}_2; \ldots; \mathtt{w}_n$ is available incrementally in a synchronized fashion where each increment $\mathtt{w}_i$ consists of a duration value and a set of symbols. Essentially, due to the causality, we can only report occurrences ending in the past $\mathtt{w}_{0..i}$ after reading a segment $\mathtt{w}_i$. Observing more segments adds more begin and end times for which we can report occurrences. Mechanically, each time we read segment $\mathtt{w}_i$ several things happen:

1) We spawn a new process that checks possible occurrences beginning in $\mathtt{w}_i$.
2) We update all the processes previously spawned.
3) We check all the updated processes if there are completed occurrences. If so, we extract and report them.

After having read the segment $\mathtt{w}_i$ the output/state of the online algorithm is the pair $(M(i), P(i))$ where $M(i)$ is the set of occurrences completed in $\mathtt{w}_i$ and $P(i)$ is the set of active processes. The output $M(i)$ has no further influence on future occurrences and it is not really part of the state of the algorithm. Currently we represent the other ingredient $P(i)$ as an algebraic expression and we use a set of rewriting rules similar to derivatives of regular expressions [4] to update $P(i)$ at each increment. Although derivatives are not known for their efficiency, they provide a simple and elegant method, and it is a good starting point to achieve more efficient algorithms.

V. Implementation and Evaluation

The monitoring tool MONTRE is a stand-alone command line program which uses structured text files for input/output specification. We have implemented it using Pure[2] and C++. Pure is a functional programming language based on term rewriting with a support for native code compilation and native calls to dynamic libraries. In implementation we use integer-valued time model where a time value is represented by a 64-bit integer. For majority of applications integers give us enough precision and range as well as we avoid rounding errors of floating point arithmetic. Regular expression operations (concatenation, etc.) over sets of zones are intensive numerical computations so we implement them in C++ to be compiled as a dynamic library, `libmontre`, which is independent from top-level algorithms.

For the offline algorithm we do not need sophisticated rewriting abilities of Pure and the code in Pure mostly handles recursive calls to `libmontre` and memoize previous computations. Offline mode is invoked by the option `-b` or `--offline` and requires a valid pattern (in single quotes) and a file as arguments as follows.

```
montre -b '(p;q)%(3,4)' 'my_sym_beh.txt'
```

The input file should be structured such that each line contains a symbolic interval formed by a duration value and a string for the symbol set as below. For the empty symbol set the double-dash (`--`) can be used.

```
2    pq
2    p
1    pq
3    --
...
```

After execution the output file includes a set of zones where 6 zone inequalities (see Figure 11-ii) are defined by 6 numbers $(b\ b'\ e\ e'\ d\ d')$ and a 6-bit binary vector showing the strictness of corresponding inequality. It is interpreted such that an inequality is strict if the corresponding bit value is 0 and it is not strict if the bit value is 1. An example output file is below.

```
(0 1 3 4 3 4) 100111
(5 6 8 9 3 4) 100111
...
```

Besides MONTRE provides an option to project zones over the end-axis. It is useful when you would like to link TRE with temporal logics similar to the suffix implication of PSL. This behavior can be enabled by `--output-type=end` option.

For the online algorithm we extensively use rewriting engine of Pure with calls to `libmontre` and similarly online mode is invoked by the option `-i` or `--online`. This time filename is optional and MONTRE will expect a duration value and a string for the symbol set from standard input if you do

[2] http://purelang.bitbucket.org

TABLE I. PATTERNS USED IN THE EVALUATION

#	Test Patterns
1	`p`
2	`p;q`
3	`(p;q;(p;q;p)%(0,4);q;p)%(0,8)`
4	`((p;q)%(6,8);r)&(p;(q;r)%(6,8))`
5	`p;(q;r)*`

not provide a file. Otherwise it will read the file line by line in one pass.

In order to evaluate the effectiveness of MONTRE we now present some brief experiments. Although performance results depend on the input file and the pattern, we show MONTRE is fast enough to be used for large files typically confronted in real-time system analysis. Table I shows 5 test patterns we used in the experiments and we search these patterns on input files with varying sizes. The size of an input file equals to the number of lines thus the number of symbolic intervals. We perform all experiments on a 3.3GHz machine running 64-bit Linux and we use the GNU `time` facility to measure execution times (user CPU time) and memory usage (maximum resident set size).

TABLE II. EXECUTION TIMES (in seconds)

	Offline Algorithm Input Size			Online Algorithm Input Size		
#	100K	500K	1M	100K	500K	1M
1	0.06	0.27	0.51	6.74	29.16	57.87
2	0.08	0.42	0.74	8.74	42.55	81.67
3	0.23	1.09	2.14	28.07	130.96	270.45
4	0.13	0.50	1.00	15.09	75.19	148.18
5	0.11	0.49	0.96	11.53	52.87	110.58

TABLE III. MEMORY USAGE (max. RSS in MB)

	Offline Algorithm Input Size			Online Algorithm Input Size		
#	100K	500K	1M	100K	500K	1M
1	17	24	33	14	14	14
2	21	46	77	14	14	14
3	28	77	140	14	14	14
4	23	51	86	15	15	15
5	20	37	60	15	15	15

In Table II we compare the execution times of offline and online algorithms. Both algorithms run in linear time with respect to input size for given inputs while online algorithm is two orders of magnitude slower than offline algorithm. One reason for the slowness is that the offline algorithm can utilize tight loops of regular expression operations whereas the online algorithm has to apply different operations at each step. Moreover runtime checks performed in rewriting process create an extra overhead for the online algorithm. However, not surprisingly, the online algorithm requires almost constant space in memory while the space requirement for the offline algorithm increases linearly with respect to the input size as seen in Table III.

VI. CONCLUSION

In this paper we presented the tool MONTRE to monitor timed regular expressions over symbolic behaviors of real-time systems. We introduced a novel online timed pattern matching algorithm and our tool integrates online and offline algorithms enhanced by practical features using the same interface. Experimental results suggest that the offline algorithm is faster but there are cases where the online algorithm should be preferred such as systems in operation or real-time monitoring with limited memory. However, there are a lot of room to optimize the online algorithm and we will study such techniques to increase the performance. Besides there might be more practical strategies to reduce memory requirements of the offline algorithm such as paging out zone sets or disabling memoization in expense of execution time.

REFERENCES

[1] Rajeev Alur and Thomas A Henzinger. "Logics and models of real time: A survey". In: *Real-Time: Theory in Practice*. 1992, pp. 74–106.

[2] Eugene Asarin, Paul Caspi, and Oded Maler. "A Kleene Theorem for Timed Automata". In: *Logic in Computer Science (LICS)*. 1997, pp. 160–171.

[3] Eugene Asarin, Paul Caspi, and Oded Maler. "Timed Regular Expressions". In: *Journal of ACM* 49.2 (2002), pp. 172–206.

[4] Janusz A. Brzozowski. "Derivatives of Regular Expressions". In: *Journal of the ACM* 11.4 (1964), pp. 481–494.

[5] Eduard Cerny et al. *SVA: The Power of Assertions in SystemVerilog*. Springer, 2014.

[6] Ben D'Angelo et al. "LOLA: Runtime monitoring of synchronous systems". In: *Temporal Representation and Reasoning (TIME)*. 2005, pp. 166–174.

[7] C Stuart Daw, Charles Edward Andrew Finney, and Eugene R Tracy. "A review of symbolic analysis of experimental data". In: *Review of Scientific Instruments* 74.2 (2003), pp. 915–930.

[8] Cindy Eisner and Dana Fisman. *A practical introduction to PSL*. Springer, 2007.

[9] John Havlicek and Scott Little. "Realtime regular expressions for analog and mixed-signal assertions". In: *Formal Methods in Computer-Aided Design (FMCAD)*. 2011, pp. 155–162.

[10] Stephen Cole Kleene. "Representations of events in nerve nets and finite automata". In: *Automata Studies: Annals of Mathematics Studies* 34 (1956), pp. 3–42.

[11] Dejan Nickovic and Oded Maler. "AMT: A Property-Based Monitoring Tool for Analog Systems". In: *Formal Modeling and Analysis of Timed Systems (FORMATS)* (2007), p. 304.

[12] Koushik Sen and Grigore Rosu. "Generating Optimal Monitors for Extended Regular Expressions". In: *Electronic Notes Theoretical Computer Science* 89.2 (2003), pp. 226–245.

[13] Ken Thompson. "Regular Expression Search Algorithm". In: *Communications of the ACM (CACM)* 11.6 (1968), pp. 419–422.

[14] Dogan Ulus et al. "Online Timed Pattern Matching using Derivatives". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2016, pp. 736–751.

[15] Dogan Ulus et al. "Timed Pattern Matching". In: *Formal Modeling and Analysis of Timed Systems (FORMATS)*. 2014, pp. 222–236.