Pattern Matching in Link Streams: a Token-based Approach

Clément Bertrand[1], Hanna Klaudel[1],
Matthieu Latapy[2], and Frédéric Peschanski[2]

[1] IBISC, Univ Evry, Université Paris-Saclay, 91025, Evry, France
[2] LIP6 – Sorbonne Université, Paris, France

Abstract. Link streams model the dynamics of interactions in complex distributed systems as sequences of links (interactions) occurring at a given time. Detecting patterns in such sequences is crucial for many applications but it raises several challenges. In particular, there is no generic approach for the specification and detection of link stream patterns in a way similar to regular expressions and automata for text patterns. To address this, we propose a novel automata framework integrating both timed constraints and finite memory together with a recognition algorithm. The algorithm uses structures similar to tokens in high-level Petri nets and includes non-determinism and concurrency. We illustrate the use of our framework in real-world cases and evaluate its practical performances.

Keywords: Timed pattern recognition, finite-memory automata, timed automata, complex networks, link streams.

1 Introduction

Large-scale distributed systems involve a great number of remote entities (computer nodes, applications, users, etc.) interacting in real-time following very complex network topologies and dynamics. One classical way to observe the behavior of such complex system is to take snapshots of the system at given times and represent the global state as a very large and complex graph. The behavior of the system is then observed as a timed sequence of graphs. The algorithmic detection of patterns of behaviors in such large and dynamic graph sequences is a very complex, most often intractable, problem. The *link stream* formalism [11] has been proposed to model complex interactions in a simpler way. A link stream is a sequence of timestamped links (t, u, v), meaning that an interaction (e.g. message exchange) occurred between u and v at time t. The challenge is to develop analysis techniques that can be performed on the link streams directly, without having to build the underlying global graph sequence. The patterns of interests in link stream involve both structural and temporal aspects, which raises serious challenges regarding the description of such patterns and the design of detection algorithms. The problems has been mostly approached from two different angles. First, recognition algorithms have been developed for specific patterns such as

triangles in [12]. The focus is on the performance concerns, involving non-trivial algorithmic issues. At the other end of the spectrum, *complex event processing* (CEP) has been proposed as a higher-level formalism to describe more complex interaction patterns in generic event streams [1,19]. These generic works do not handle the specificity of the input streams, in particular the real-time and graph-related properties of link streams. Our objective is to develop an intermediate approach, generic enough to cover a range of interesting structural and temporal properties, while taking into account the specificities of the link streams abstraction.

Our starting point is that of regular expressions and finite state automata for textual patterns. We interpret link streams as (finite) words and develop a pattern language involving both structural and temporal features. We propose a new kind of hybrid automata, the *timed ν-automata*, as recognizers for this pattern language. They are build upon finite state automata (FSA) with both timed [2,3] and finite-memory [10,6,7] features. The patterns themselves can be specified by enriched regular expressions, and "compiled" to timed ν-automata. The problem of timed pattern matching has been addressed only quite recently in e.g. [14,15,17,18]. while our model bears some resemblance with these propositions, the main novelty is the study of pattern matching in the presence of real-time constraints *together with* finite memory. To our knowledge this has not been addressed in the literature.

One interesting aspect of the automata model we propose is that the recognition principles are based on a non-trivial *token game*. Indeed, our main inspiration comes from high-level Petri nets. Based on this formalism, we developed a prototype tool that we applied to real-world link streams analysis. Performance issues are raised but the results are encourageing. In particular, our experiments confirm the following key fact: timing properties often help in reducing the performance cost induced by storage of information in memory.

The outline of the paper is as follows: In Section 2 we introduce the principles of finding patterns in link streams. The automata model and pattern language are formalized in Section 3. The prototype tool and practical experiments are then discussed in Section 4.

2 Patterns in link streams

We consider *link streams* [11] defined as sequences of triples (t_i, u_i, v_i), meaning that we observe a link between nodes u_i and v_i at time t_i. Figure 1 (left) shows an example of a link stream that models interactions between nodes a, b, c and d. For example at time $t = 6$ a link from node d to node b is observed, which corresponds to a triple $(6, d, b)$ in the stream.

A *pattern* in such a link stream can be seen as a series of (directed) subgraphs observed in a given time frame. For example, at time $t = 15$ we observe the subgraph described on the right of Figure 1. This graph has been formed in the depicted time frame of 7s. One trivial way to detect such patterns is to build all the intermediate graphs and solve the subgraph isomorphism problem

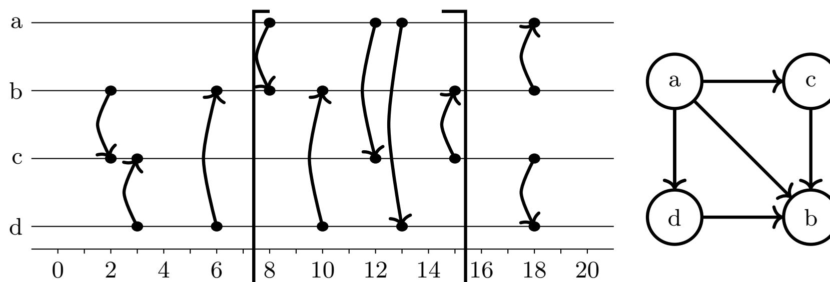


Fig. 1. A link stream (left) and its graph projection in time interval $[8, 15]$ (right).

at each time step. This is however out of reach in most situations most notably because: (1) real-world link streams involve very large graphs, and (2) subgraph isomorphism is a NP-complete problem. Hence, in practice dedicated algorithms are developed for specific kinds of subgraphs. One emblematic example is the *triangle* for which specialized algorithms have been developed. A triangle is simply the establishment of a complete subgraph of three nodes, in a directed way. In network security this is a known trigger for attacks: two nodes that may be identified as "attackers" negotiate to "attack" a third node identified as the "target". Such a trigger can be observed in Figure 1 (right) with a and d attackers targeting b. In real-world link streams, detecting such triangles is in fact not trivial, as explained for example in [12].

In this paper, our motivation is to develop a more generic approach able to handle not only such triangles but also other kinds of patterns: directed polygons, paths, alternations (e.g. links that appear periodically), etc. We also require the matching algorithms to be of practical use, hence with efficiency in mind. Our starting point is the theory of *finite-state automata* (FSA) and *regular expressions*. Indeed, if we ignore the timestamps, a link stream is similar to a finite word, each symbol being a directed link (a pair of nodes). For example in the time frame $(8, 15)$ we observe the following "word":

$$(a, b)(d, b)(a, c)(a, d)(c, b).$$

Based on such a view, we can use FSA as pattern recognizers and regular expressions as a high-level specification language. A regular pattern for the triangle example is as follows:

$$\Big(((a \to d) \mid (d \to a)) \cdot ((a \to b) \otimes (d \to b))\Big) \otimes (@ \to @)^*$$

This expression uses classical regular constructs such as concatenation $\cdot$, disjunction $|$, the *Kleene* star $*$ and shuffle $\otimes$. The symbol @ is used as a placeholder for any possible node, hence $(@ \to @)$ means "any possible link". Based on such specification, it is easy to build a finite-state automaton to recognize the triangles in an untimed link stream very efficiently.

However, the "regular language" approach fails to capture the timing properties of link streams. What we need is a form of real-time pattern matching.

Quite surprisingly, there are very few research works addressing this problematic, despite the broad success of timed automata [2] in general. An important starting point is the *timed regular expressions* formalism [3]. The basic principle is to interpret input words, hence link streams, as *timed event sequences*: a succession of either symbols or *delays* corresponding to a passage of time. Below is an example of a link stream as a timed event sequence:

$$(a,b)2(d,b)2(a,c)1(a,d)1(c,b).$$

A timed regular expression for the triangle pattern can then be specified, e.g.:

$$\Big(((a \to d) \mid (d \to a)) \cdot \langle (a \to b) \otimes (d \to b) \rangle_{[0,1]} \Big) \otimes (@ \to @)^*$$

The *delay* construction $\langle S \rangle_{[x,y]}$ says that the subpattern S must be detected in time interval $[x,y]$. For the triangle pattern it means that the nodes a and d are only observed as "attacking" target b if they simultaneously link to b in the time interval of one second.

Another fundamental aspect that we intend to capture in link stream patterns is that of *incomplete knowledge*. In classical and timed automata, symbols range over a fixed and finite alphabet. In link streams, this means that the nodes of the graphs must be known in advance, which is in general too strong an assumption. In an attack scenario, for example, we must consider an *open system*: it is very likely that only the target is known in advance, and the two attackers remain undisclosed.

The kind of pattern we intend to support is e.g.:

$$\Big((\sharp X \to \sharp Y) \cdot \langle (X \to b) \otimes (Y \to b) \rangle_{[0,1]} \Big) \otimes (@ \to @)^*.$$

In this pattern, the variables X and Y represent unknown nodes corresponding to two "attackers". The construction $\sharp X$ means that the input symbol (hence node) associated to X must be *fresh*, i.e., not previously encountered. In case of a match this node is associated to X and kept in memory. With the operator $X!$ (the dual of $\sharp X$), after matching a value associated to variable X, all the values associated to it are discarded (i.e. the associated set is cleared).

The sub-pattern $(\sharp X \to \sharp Y)$ describes a link between two fresh nodes. Note that since Y is matched after X, the freshness constraints impose that its associated node is distinct from the one of X. To match the sub-expression $(X \to b)$, the input must be a link from the node already associated with X in memory to node b. This is a potential attack on the target b.

To handle such dynamic matching, one must consider a (countably) infinite alphabet of unknown symbols. This has been studied in the context of *quasi-regular languages* and *finite memory automata* (FMA) [10]. In this paper, we build upon the model of ν-automata that we developed in a previous work [6,7]. It is a variant of FMA, which is tailor-made for the problem at hand. If compared to the classical FMA model, the ν-automata can be seen as a generalization to handle *freshness conditions* [13].

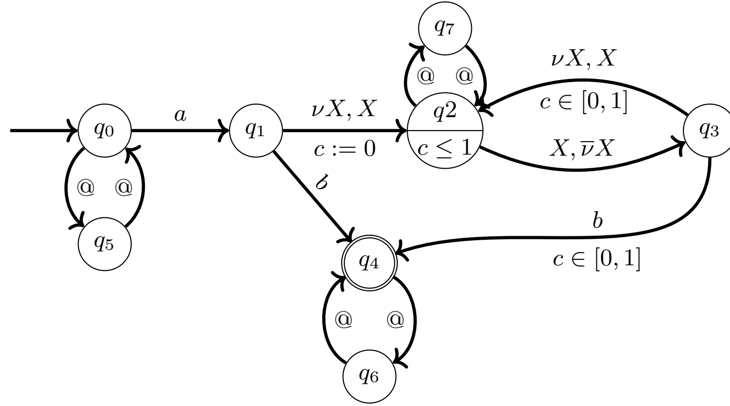


Fig. 2. Automaton for $(@ \to @)^* \otimes \Big((a \to b) \vee a \to \sharp X \cdot \langle (X! \to \sharp X)^* \cdot (X! \to b)\Big)\rangle_{[0,1]}$

The resulting mixed model of *timed ν-automata* is quite capable in terms of expressiveness. The automaton formalism is a combination of both the timed constraint and clocks reset from timed automaton and the memory management of the ν-automaton. As an illustration, Figure 2 depicts an automaton that detects in a link stream all the paths from a node a to a node b such that each link is established in at most one second. We suppose that the automaton is defined for the alphabet $\Sigma = \{a, b\}$, i.e., only the nodes a and b are initially known. The labels $\nu X, X$ and $X, \overline{\nu} X$ are the automata variants of the operators $\sharp X$ and $X!$ discussed previously. An example of an accepting input is:

$$(a, y)\ 0.1\ (y, z)\ 0.3\ (y, b).$$

Initially, in state q_0 the known symbol a is consumed while transiting to state q_1. The unknown symbol y is saved in the memory associated to variable X while transiting to state q_2. This only works because the symbol y is fresh, i.e., not previously encountered. The delay of 0.1 second is consumed in state q_2 while increasing the value of the clock c to 0.1. The state constraint $c \leq 1$ is still satisfied. The next input y may either lead to q_3 (because it was previously associated to X) or q_7 (because the symbol @ accepts any input). The recognition principle is non-deterministic so both possibilities will be tried:

- if the transition $q_2 \xrightarrow{X, \overline{\nu} X} q_3$ is taken, X is no longer associated to any symbol in q_3. The next input is the unknown symbol z. From q_3, only the transition $q_3 \xrightarrow[c \in [0,1]]{\nu X, X} q_2$ is enabled. In q_2 the variable X would be associated to z. However, this path is doomed because the next (and last) link does not start from z. Then at the end of the input sequence the path leads to state q_2 which is not a terminal state.
- if the transition $q_2 \xrightarrow{@} q_7$ is taken then the value associated to X is not discarded and the input z leads back to the state q_2 through transition

$q_7 \xrightarrow{@} q_2$. The input 0.3 increases the clock value to $c = 0.4$. The next input y may again lead either to q_7 or q_3 as in the previous case. In state q_3 the input b enables only the transition $q_3 \xrightarrow[c\in[0,1]]{b} q_4$, which leads to the final state q_4 (since $b \in \Sigma$).

We reach an accepting state because the clock value $c = 0.4$ is still under 1 second. On the other hand, if the second delay is not 0.3 but e.g., 1.0 then the link stream is not recognized because of a timeout in state q_2.

3 Automata model and recognition principles

The automata model we propose can be seen as a layered architecture with: (1) a classical (non-deterministic) finite-state automata layer, (2) a timed layer (based on [3]) and (3) a memory layer (based on [7]). These layers are obviously dependent but there is a rather clean interface between them.

3.1 The timed ν-automata

Definition 1. *A* timed ν-automaton *is a tuple:*

$$A = (\underbrace{\Sigma, Q, q_0, F, \Delta}_{\textit{finite-state}}, \underbrace{C, \Gamma}_{\textit{timed}}, \underbrace{\mathcal{U}, V}_{\textit{memory}})$$

The basic structure is that of a finite-state automaton. We first assume a finite alphabet of *known symbols* denoted by Σ. The finite set Q is that of *locations*[3]. The initial location is q_0 and F is the set of final locations. The component Δ is the set of transitions (explained in details below).

This basic structure is extended for the timed constraints with a set C of *clocks* (ranging over $c_0, c_1, \ldots$) and a map Γ that associates to each location a set of timed constraints. A time constraint is a time interval of the form $[min, max] \in \mathbb{I} = [\mathbb{R}_{\geq 0}, (\mathbb{R}_{\geq 0} \cup \{+\infty\})]$ giving the minimum and maximum values of the clock so that the automaton can "live" in the given location. A transition can also be annotated with time constraints to restrict its firing. Note that the maximum value may be infinite, which means there is no time limit for crossing the transition. The only operations we need on intervals is that of intersection $I_1 \cap I_2$ and membership $c \in I$.

The memory component is a finite set V of *variables* (ranging over $X, Y, \ldots$) for the memory constraints. Each variable will be associated to a (possibly empty) set of *unknown symbols* ranging over a countably infinite alphabet denoted by $\mathcal{U}$. These symbols are all the symbols that may appear in an input

[3] The notion of a location here corresponds to a state in classical automata theory. We rather use the term state in the sense of *actual state* or *configuration* (as in FMAs [10]), i.e., an element of the state-space: a location together with a memory content and clock values.

sequence, which are not in Σ. Unlike FMA, which are limited by the number of their registers, the ν-automata use variables of dynamic size, which allows to recognize words composed of an arbitrary number of distinct unknown symbols.

Definition 2. *A transition $t \in \Delta$ of a timed ν-automaton is of the form:*

$$q \xrightarrow[\gamma, \rho]{\nu, e, \overline{\nu}} q'$$

with q (res. q') the starting (resp. ending) location, $\nu \subset V$ a set of variable allocations, $\overline{\nu} \subset V$ a set of variable releases. The *event* e is either a symbol in the finite alphabet Σ, a use of a variable in V or an ε. The timed constraint γ is a guard function of type $C \to \mathbb{I}$, associating to each clock a unique time interval. Finally, ρ is the set of clocks to be reset to 0 while crossing the transition. To simplify the notation of transitions, the empty sets are omitted.

3.2 Dynamics

For a variable $X \in V$, an allocation is a set M_X of unknown symbols, a finite subset of $\mathcal{U}$, together with a flag. The flag may be $M_X^{\bullet}$ (read mode, default) or M_X° (write mode). In read mode, the only available operation is to check if an input symbol is already present in $M_X^{\bullet}$. In write mode, only a fresh symbol $\alpha \notin \bigcup_{X \in V} M_X$ may be added. An important property of the memory model is the following.

Definition 3. *A* token *is a pair $k = (k_{\mathsf{time}}, k_{\mathsf{mem}})$ with k_{time} a function from clocks to clock values, and k_{mem} a mapping from variables to sets of* allocations*.*

Definition 4. *A* configuration *of an automaton is a mapping S from locations Q to corresponding reachable clocks and memory valuations.* [4]*.*

We denote by $S(q)$ the set of tokens associated to location q.

Property 1. INJ (memory injectivity)
For any pair of distinct variables X, Y we have $M_X \cap M_Y = \emptyset$.

We denote by $\mathsf{INJ}(k)$ the fact that token k respects property 1. Although most memory models do not work like this, this injectivity property is an essential feature of finite-memory automata models (cf. [10]).

The initial configuration of every timed ν-automaton contains the single token $(\{X \to \emptyset^{\bullet} | \forall X \in V\}, \{c \to 0 | \forall c \in C\})$ in the initial location. The recognition of a pattern in an input sequence in this setting is a non-deterministic process. It corresponds to the propagation of tokens over locations of the automaton representing the pattern. The input is accepted if after reading the whole input sequence there is at least one token in some final location. The token itself allows to retrieve the admissible clock values and memory content.

[4] We reuse the token notion of high-level Petri nets because it is quite similar conceptually. The configuration roughly corresponds to the marking of a Petri net.

The core of the recognition principle is a partial function δ that takes a token, a transition, and an input symbol (ε if none) to produce either a new token to put into the destination location, or nothing ($\bot$) if the transition is not enabled.

Definition 5. *(update) Consider the transition* $t = q \xrightarrow[\gamma,\rho]{\nu,e,\overline{\nu}} q'$*, a token* $k = (k_{\mathsf{time}}, k_{\mathsf{mem}})$ *present in* q*, and* α *an input symbol. If we pose* $k'_{\mathsf{time}} = \delta_{\mathsf{time}}(t, k_{\mathsf{time}})$ *and* $k'_{\mathsf{mem}} = \delta_{\mathsf{mem}}(t, k_{\mathsf{mem}}, \alpha)$*, then the next token to put in location* q' *is:*

$$\delta(t, (k_{\mathsf{time}}, k_{\mathsf{mem}}), \alpha) = \begin{cases} (k'_{\mathsf{time}}, k'_{\mathsf{mem}}) \text{ if } k'_{\mathsf{time}} \neq \bot \text{ and } k'_{\mathsf{mem}} \neq \bot \\ \bot \text{ otherwise} \end{cases}$$

The next token only exists if neither the functions for time update δ_{time} or memory update δ_{mem} yield the undefined value $\bot$.

The time update function δ_{time} corresponds to the time model of [3].

Definition 6. *(time update) Let* C *be a set of clocks,* q *a location and* Γ_q *the time constraints function. The* δ_{time} *function is defined as follows:*

$$\delta_{\mathsf{time}}(t, k_{\mathsf{time}}) = \begin{cases} \bot \text{ if } \exists c \in C \setminus \rho,\ allow(c) \cap \Gamma_{q'}(c) = \emptyset & \text{(case 1.1)} \\ \qquad \vee \exists c \in \rho,\ allow(c) = \emptyset \vee 0 \notin \Gamma_{q'}(c) & \text{(case 1.2)} \\ \text{otherwise } \{c \mapsto k'_c \mid c \in C\} & \text{(case 2)} \\ \qquad \text{with } k'_c = \begin{cases} 0 \text{ if } c \in \rho \\ k_{\mathsf{time}}(c) \text{ otherwise} \end{cases} \end{cases}$$

where $allow(c) = k_{\mathsf{time}}(c) \cap \gamma(c) \cap \Gamma_q(c)$ *and* γ *is the time constraint on* t.

If at least one of the non-reseted clocks c fails to satisfy either the transition guard or the locations constraints (case 1.1) then no token is produced. Another case of failure is if a reseted clock fails to satisfy the initial location constraint and transitions guards, or zero is not accepted in the destination location as the outgoing value of the clock (case 1.2). A token is otherwise produced (case 2), which simply consists in updating the clock to the correct value (either 0 if there is a reset for the clock, or to the value prescribed by the input token).

The principle of updating the memory is a little bit more complex. The memory part of the next token is computed by the memory update function δ_{mem} from the previous memory component depending on an input symbol α. The computation respects the following ordering: the allocation of the variables in set ν is performed before checking the consistency between the input and transition label, and before releasing the variables in the set $\overline{\nu}$.

Definition 7. *(memory update) Let V be a set of variables, and $\mathcal{U}$ an infinite set of unknown symbols. The δ_{mem} function is defined as follows:*

$$\delta_{mem}(t, k_{mem}, \alpha) = \begin{cases} \perp \ \textit{if} \ e \notin V \wedge \alpha \neq e & (c.1.1) \\ \quad \vee e \in V \wedge \alpha \notin \mathcal{U} & (c.1.2) \\ \quad \vee e \in V \setminus \nu \wedge k_{mem}(e) = M_e^{\bullet} \wedge \alpha \notin M_e & (c.1.3) \\ \quad \vee (e \in \nu \vee k_{mem}(e) = M_e^{\circ}) \wedge \exists Y, \ \alpha \in k_{mem}(Y) & (c.1.4) \\ \textit{otherwise} \ \{X \mapsto k'_X \mid X \in V\} & \\ \qquad \textit{with} \ k'_X = \begin{cases} \emptyset^{\bullet}, \ \textit{if} \ X \in \overline{\nu} & (c.2.1) \\ (M_X \cup \{\alpha\})^{\bullet}, \ \textit{if} \ X = e & (c.2.2) \\ M_X^{\circ}, \ \textit{if} \ X \in \nu & (c.2.3) \\ k_{mem}(X), \ \textit{otherwise} & (c.2.4) \end{cases} & \end{cases}$$

where $e \in V \cup \Sigma \cup \{\varepsilon\}$ denote the input enabling transition t and the sets ν and $\overline{\nu}$ denote respectively the sets of allocated and freed variables.

In the first four cases no token can be produced. If the transition label e is a known symbol in Σ, then the input α must exactly match else it is a failure (c.1.1). If otherwise e corresponds to a variable, then α must be an unknown symbol in $\mathcal{U}$ (c.1.2). A more subtle failure is (c.1.3) for a variable $e \in V$ in read mode. In this situation the input symbol must be already recorded in the memory associated to e. Complementarily, if the variable e is in write mode (or is put in write mode along the transition), then the input symbol must be *fresh* (c.1.4).

If the next token is produced then for each variable X the associated memory content M_X is updated as follows. If X is to be released (in set $\overline{\nu}$) then the memory is cleared and put in read mode (c.2.1). If it is not released and the variable is to be read (i.e. $X = e$) then α is added to the memory content (c.2.2). In (c.2.3) the variable is not read ($X \neq e$) but it is allocated (in set ν). In this situation the memory content is put in write mode. Otherwise (c.2.4) the memory is left unchanged for variable X.

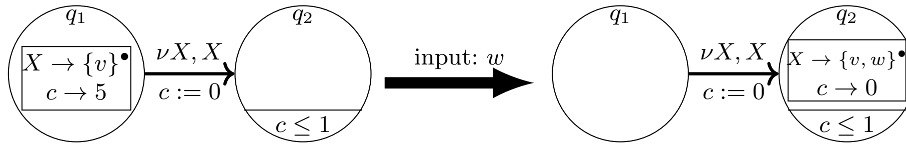


Fig. 3. Passing a transition of the automaton from Figure 2 with input w

Example 1. Figure 3 illustrates the generation of a new token taking as an example of the transition $d = q_1 \xrightarrow[\rho=\{c\}]{\{X\},X,\{\}} q_2$ from the automaton in Figure 2. Based on the token $(\{c \to 5\}, \{X \to \{v\}^{\bullet}\})$ in location q_1, the input w enables the transition d producing a new token in q_2, computed as follows:

- for the time component, the value of clock c verifies the transition constraints. In the arrival location q_2 the clock is reseted and the clock value 0 verifies

the time invariant in q_2. The transition would be disabled without the reset because clock value 5 does not satisfy the invariant. Of course, the clock value in the arrival token is $c \to 0$.
- for the memory component, the transition d is only enabled when the input is an unknown symbol, because transition d is labeled with a variable. Since the alphabet Σ of known symbols is $\{a, b\}$, the symbol w is considered as unknown, i.e., $w \in \mathcal{U}$. Because the allocations are applied before checking the input, the variable X is allocated and then used to enable the transition. So the symbol w should be added to M_X in the newly generated token. However, it is only possible if the input is fresh. Since $M_X = \{v\}$ and X is the only variable, this freshness constraint is satisfied. Hence, the new token associates the memory $\{v, w\}^\bullet$ to X.

As both δ_{time} and δ_{mem} can compute a new value, a new token is generated for the location q_2. ◇

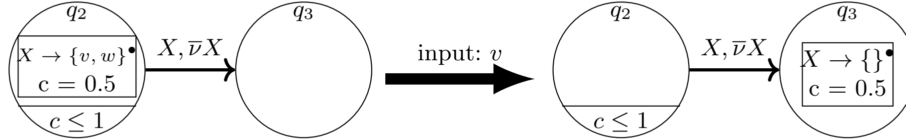


Fig. 4. Passing a transition of the automaton from Figure 2 with input v

Example 2. Figure 4 presents another example of transition in the automaton of the Figure 2. This example illustrates a case of memory evolution with δ_{mem}. Here the variable X is used as the trigger and then freed. The variable's freeing occurs simultaneously to reset of the clocks, after checking of guards. As X is not allocated during the transition and was neither allocated before, the transition is enabled only if the input is an unknown symbol and belongs to M_X. The input is actually the unknown symbol $v \notin \Sigma = \{a, b\}$. Furthermore, $v \in \{v, w\} = M_X$, so the transition may be passed and the variable X is cleared in the newly generated token. ◇

The important property of memory injectivity must be preserved through δ_{mem} to fulfill the freshness constraints.

Proposition 1. *(preservation of injectivity) Let k be a token such that $\mathsf{INJ}(k)$, and suppose $k' = \delta(t, k, \alpha) \neq \bot$ for some transition t and input α. Then we have $\mathsf{INJ}(k')$.*

Proof. In the token $k = (k_{\mathsf{time}}, k_{\mathsf{mem}})$ only the memory component k_{mem} is affected by the injectivity property. The main hypothesis $\mathsf{INJ}(k)$ means that $\forall X, Y\ (X \neq Y),\ k_{\mathsf{mem}}(X) \cap k_{\mathsf{mem}}(Y) = \emptyset$.

Suppose that the transition $t = q \xrightarrow[\gamma,\rho]{\nu,e,\overline{\nu}} q'$ produces token $k' = \delta(t, k, \alpha) = (k'_{\mathsf{time}}, k'_{\mathsf{mem}})$. We have to show $\mathsf{INJ}(k')$, i.e. $k'_{\mathsf{mem}}(X) \cap k'_{\mathsf{mem}}(Y) = \emptyset$ for the same

pairs of distinct variables. In the definition of δ_{mem} (Definition 7) we are mostly concerned with cases (c.2.1) to (c.2.4) because we expect a token as output. The memory update depends on the value of the transition trigger e and is as follows:

- If e is not a variable: $e \in \varepsilon \cup \Sigma$, then the case c.2.2 of δ_{mem} cannot occur. So, in the token k' the variable domains are either empty (case c.2.1), or the same as in k (case c.2.3 or c.2.4). Given the hypothesis $\mathsf{INK}(k)$ and the fact that $\emptyset$ is the zero element of intersection, we trivially have $\mathsf{INK}(k')$ as expected.
- If e is a variable: $e \in V$ then the case c.2.2 occurs for exactly one variable of the generated token. As presented above, the variable domains generated with the cases c.2.1, c.2.3 and c.2.4 have empty intersections with each other. Only the domains generated by case c.2.2 must be handled with care. We have to consider two situations:
  - if $\alpha \in k_{mem}(e)$ then the set M_e is not modified, so the Property 1 is trivially verified;
  - or $\alpha \notin k_{mem}(e)$ then case c.1.4 ensures that α is absent in all the domains of the other variables. Thus, Property 1 is verified as well. □

Given a global configuration S and an input α, a new configuration is computed with function σ. It consists in producing all new tokens by the enabled transitions and propagating them through all the ε-transitions.

Definition 8. *(global update)*

$$\sigma(S,\alpha) = \begin{cases} \sigma_{closure}(S,\alpha) \ \textit{if}\ \alpha \in \mathbb{R}^+ & \textit{(time delay)} \\ \sigma_{closure}(\sigma_{step}(S,\alpha),0) \ \textit{otherwise} & \textit{(symbol)} \end{cases}$$

The input may be either a known or unknown symbol, or a time delay. In case of a time delay the token should just be propagated through the ε-transitions, defined by $\sigma_{\mathsf{closure}}$ presented below. If the input is a symbol, then new tokens shall be produced through the non ε-transitions, which is expressed by σ_{step}, and then propagated through the ε-closure.

Definition 9. *(event handling)*

$$\sigma_{step}(S,\alpha) = \{q' \mapsto \{\delta(t,k,\alpha) \mid t = q \xrightarrow[\gamma,\rho]{\nu,e,\overline{\nu}} q' \wedge k \in S(q)\} \mid q' \in Q\}$$

The σ_{step} function simply consists in applying the local update function δ at all locations for all non ε-transitions. The tokens belonging to S are not kept in the new configuration.

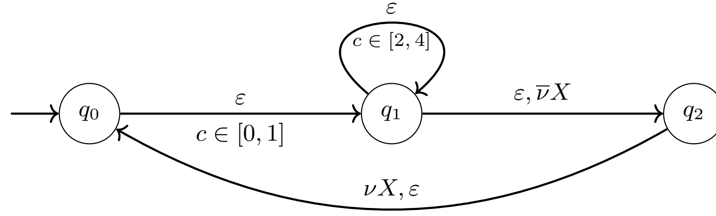
The resulting tokens are propagated for ε transitions with a time delay fixed to 0. In the case of a time delay, the ε-closure is applied with this fixed delay as argument. The function $\sigma_{\mathsf{closure}}$ thus handles the time propagation in the automaton. It produces all the possible next tokens in all locations reachable through an ε-path.

Definition 10. *(ε-closure)*

$$\begin{aligned}\sigma_{\mathit{closure}}(S,x) = \{\, q \mapsto \{k \mid\ & \exists q_0 \underbrace{\xrightarrow[\gamma_1,\rho_1]{\nu_1,\varepsilon,\overline{\nu_1}}}_{t_1} q_1 \underbrace{\xrightarrow[\gamma_2,\rho_2]{\nu_2,\varepsilon,\overline{\nu_2}}}_{t_2} \cdots q_{n-1} \underbrace{\xrightarrow[\gamma_n,\rho_n]{\nu_n,\varepsilon,\overline{\nu_n}}}_{t_n} q \in \Delta \\ & \wedge \exists k_0 \in S(q_0),\ \exists x_0 + x_1 + \cdots + x_n = x \\ & \wedge \forall i \in [1,n],\ \exists k_i = \delta(t_i, \mathit{shift}(k_{i-1}, x_{i-1}), \varepsilon) \\ & \wedge k = \mathit{shift}(k_n, x_n), \forall c \in C, k_{\mathit{time}}(c) \in \Gamma_q(c)\} \\ \mid q \in Q\} \end{aligned}$$

where $\mathsf{shift}((k_{\mathsf{time}}, k_{\mathsf{mem}}), x) = (\{c \to k_{\mathsf{time}}(c) + x | c \in C\}, k_{\mathsf{mem}})$.

The idea behind the above is to compute a decomposition of the given delay x as a sum $x_1 + \cdots + x_n$ corresponding to a specific way of waiting in the locations encountered on the ε-path. In fact, only the existence of the decomposition is required. Indeed, for two distinct decompositions the final value retained for a clock c in the next token is the same (either the delay added to the initial value, or 0 for a reseted clock).



step	tokens in q_0	tokens in q_1	tokens in q_2
0	$k_0 : (X \to \{u,v\}^\bullet, c \to 0)$		
1	$k_0 : (X \to \{u,v\}^\bullet, c \to 0)$	$k_1 : (X \to \{u,v\}^\bullet, c \to 0)$	
2	$k_0 : (X \to \{u,v\}^\bullet, c \to 0)$	$k_1 : (X \to \{u,v\}^\bullet, c \to 0)$ $k_1' : (X \to \{u,v\}^\bullet, c \to 2)$	
3	$k_0 : (X \to \{u,v\}^\bullet, c \to 0)$	$k_1 : (X \to \{u,v\}^\bullet, c \to 0)$ $k_1' : (X \to \{u,v\}^\bullet, c \to 2)$	$k_2 : (X \to \{\}^\bullet, c \to 0)$ $k_2' : (X \to \{\}^\bullet, c \to 2)$
4	$k_0 : (X \to \{u,v\}^\bullet, c \to 0)$ $k_0' : (X \to \{\}^\circ, c \to 0)$ $k_0'' : (X \to \{\}^\circ, c \to 2)$	$k_1 : (X \to \{u,v\}^\bullet, c \to 0)$ $k_1' : (X \to \{u,v\}^\bullet, c \to 2)$	$k_2 : (X \to \{\}^\bullet, c \to 0)$ $k_2' : (X \to \{\}^\bullet, c \to 2)$
5	$k_0 : (X \to \{u,v\}^\bullet, c \to 0)$ $k_0' : (X \to \{\}^\circ, c \to 0)$ $k_0'' : (X \to \{\}^\circ, c \to 2)$	$k_1 : (X \to \{u,v\}^\bullet, c \to 0)$ $k_1' : (X \to \{u,v\}^\bullet, c \to 2)$ $k_1'' : (X \to \{\}^\circ, c \to 0)$ $k_1''' : (X \to \{\}^\circ, c \to 2)$	$k_2 : (X \to \{\}^\bullet, c \to 0)$ $k_2' : (X \to \{\}^\bullet, c \to 2)$
6	$k_0 : (X \to \{u,v\}^\bullet, c \to 4)$ $k_0' : (X \to \{\}^\circ, c \to 4)$	$k_1 : (X \to \{u,v\}^\bullet, c \to 4)$ $k_1'' : (X \to \{\}^\circ, c \to 4)$	$k_2 : (X \to \{\}^\bullet, c \to 4)$

Fig. 5. Example of ε-closure with an input delay 4

Example 3. Figure 5 illustrates the propagation of tokens in an ε-closure, expressed by the function $\sigma_{\mathsf{closure}}$. The token will be propagated with the delay

$\alpha = 4$ as an input to exhibit its effect. In this example, we begin with the configuration of step 0 containing only one token in location q_0.

The tokens of this automaton are composed of a variable X and a clock c. The initial configuration, in step 0, contains only one token k_0 in q_0. This token is initialized with X in read mode and a set containing the unknown symbols u and v. There is only one clock c initialized to 0. In step 1, the token k_0 is propagated through the transition $t_{01} = q_0 \xrightarrow[c\in[0,1]]{\varepsilon} q_1$, which generates the token k_1 in location q_1. Since t_{01} has no side-effect (clock or memory update), k_1 is a copy of k_0. In step 2, the token k_1 is propagated through the transition $t_{11} = q_1 \xrightarrow[c\in[2,4]]{\varepsilon} q_1$ generating the token k_1' in location q_1. The transition has no side-effect so the memory of k_1' is the same as the memory of k_1. However, to fulfill the time constraint $c \in [2,4]$, the value of c has to be at least 2. To cross the transition, the clocks values should consume some amount of the input delay α. In step 3, both k_1 and k_1' can be propagated through $t_{12} = q_1 \xrightarrow{\varepsilon,\overline{\nu}X} q_2$. This transition has as a side-effect to clear the variable X. So both the tokens k_2 and k_2' generated respectively from k_1 and k_1' have for variable X the value $\{\}^{\bullet}$ (an empty set of symbols in read mode). Step 4 consists in the propagation of tokens k_2 and k_2' through the transition $t_{20} = q_2 \xrightarrow{\nu X,\varepsilon} q_0$. This transition has as a side effect to allocate X. However, as t_{20} is an ε-transition, the set associated to X will not be modified and X will be in write mode on the generated tokens. In step 5 two tokens are generated in location q_1, but both come from the token k_0'. As k_0'' has $c \to 2$, it cannot enable t_{01} because the clock constraint $c \in [0,1]$ is not respected. The token with $c \to 0$ crosses t_{01} and the transition t_{11} (as in step 2) generating two tokens, k_1'' and k_1''', in q_1 with different clock values. After step 5 it is not possible to generate any new token in a location with a different value than the tokens already present in it. In step 6 the propagation is over and all the clocks are increased to $k_{0_{time}}(c) + \alpha = 0 + 4$.

However, only one token is kept at a location if several are generated with identical clock and memory valuations. The step 6 corresponds to the configuration returned by $\sigma_{\mathsf{closure}}$. ◇

Since there may be an infinite number of ε-paths from a given starting location q, the following is an important Property wrt. decidability.

Proposition 2. *For a given configuration S and time delay x, the function $\sigma_{\mathsf{closure}}$ can only produce a finite amount of tokens.*

Proof. To prove the proposition, we show that both the possible memory and clocks states are finite over the propagation through the ε-closure.

First, we prove that the number of memory states is finite. An ε-transition does not read any symbol. So, the only memory operations present in an ε-closure are the allocation ν and the freeing $\overline{\nu}$. Let X be a variable of initial valuation M_X^a, where M_X is the set associated to X and a the initial mode of X. Its reachable values in the ε-closure are :

- M_X^a in all ε-paths with no operations on X,

- M_X° in all ε-paths where X is only allocated,
- $\emptyset^\bullet$ in all ε-paths where the last memory operation used on X is a freeing $\overline{\nu}$,
- $\emptyset^\circ$ in all ε-paths where X was freed at least once and the last memory operation on X is an allocation ν.

As a consequence, if the tokens are composed of n variables, after the propagation in an ε-closure at most 4^n variations of each initial memory valuation can be generated.

To prove that the number of clock valuations is finite we recall that our time model is based on *timed pattern matching* [3], which implies that clock resets can only be on non ε-transitions. Thus, the final clock values after the propagation over an ε-closure are the initial values increased by a possible delay given as an input. The number of clock valuations is constant through the propagation.

As the number of memory states and clocks states are both finite, the number of combinations between them is finite too. An upper bound for the number of tokens generated in an ε-closure is $nb_{\text{tokens}} \cdot 4^{nb_{\text{vars}}} \cdot nb_{\text{states}}$, where nb_{tokens} is the number of tokens composing the initial configuration (as an upper bound for the number of memory valuations), nb_{vars} is the number of variables composing a token, and nb_{states} is the number of states composing the ε-closure. □

3.3 Pattern language

The description of non-trivial patterns in link streams can become tedious if specified directly as automata. Indeed, even simple patterns can yield very large automata. We are looking for a more concise way to describe the patterns, in the spirit of regular expressions. We propose the language of *timed ν-expressions* to specify patterns for link streams.

Node	$n, n_1, n_2 \ldots ::=$	k	(known node)
		X	(variable, unknown node)
		@	(arbitrary node)
Expression	$e, e_1, e_2, \ldots ::=$	n	(node)
		$n_1 \to n_2$	(link)
(regular)		$e_1 \,.\, e_2$	(concatenation)
		$e_1 \mid e_2$	(disjunction)
		$e_1 \otimes e_2$	(shuffle)
		e^*	(iteration)
(time)		$\langle e \rangle_{[x,y]}$	(delay[5])
(memory)		$\sharp\{X_1, \ldots, X_n\}e$	(allocation)
		$e\{X_1, \ldots, X_n\}!$	(release)

Table 1. The (core) pattern language

The syntax of the core constructs is given in Table 1. The basic constructs are those of traditional regular expressions. The symbols are referring to known,

[5] Following [3] the expression inside a delay should not be empty.

unknown or arbitrary nodes. The link construct $n_1 \rightarrow n_2$ symbolizes a *non-breaking* connection between two nodes. The delay construct for time constraints is the same as in [3]. The constructs for memory management are based on variable occurrences (for unknown nodes), allocations and releases. The notation $\sharp\{X_1, \ldots, X_n\}e$ (resp. $e\{X_1, \ldots, X_n\}!$) means that the variables $X_1, \ldots, X_n$ are allocated (resp. released) before (resp. after) recognizing the subexpression e. The shuffle operator $\otimes$ is present in the language to ease the description of patterns with independent parts.

The semantics of the pattern language is given in terms of a generated timed ν-automaton. By lack of space, we do not describe the translation formally in this paper but only describe it informally. We intend to investigate the formal properties of the language (e.g. language equivalence) in a future work. Note that the translation is relatively straightforward. The translation rules for the regular expression constructs are the classical ones. A special case is the link expression $n_1 \rightarrow n_2$ that corresponds to a basic automaton with three states and two transitions, one for n_1 and the second for n_2. One important property is that this construction is non-breaking (e.g. it is atomic for the shuffle). For the delay construct a thorough explanation is given in [3]. It mostly remains to explain the translation of the memory operations.

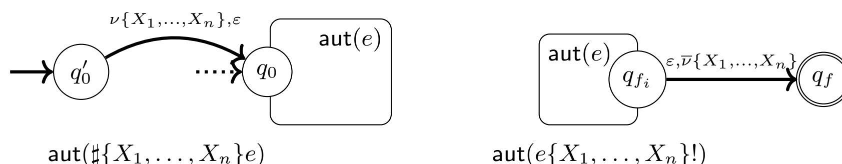


Table 2. Automata for memory operators

Table 2 illustrates how the allocation and release operators are translated:

- The function aut: *expression* $\rightarrow$ *automaton* translates a timed ν-expression to the corresponding timed ν-automaton.
- The translation of $\sharp\{X_1, \ldots, X_n\}e$ gives rise to a new initial location q_0' and a ε-transition between q_0' and the initial location of the automaton generated from e, which allocates the variables $X_1, \ldots, X_n$. The new initial location of the automaton is q_0'. The translation of $e\{X_1, \ldots, X_n\}!$ leads to the creation of a new final location q_f and a new transition from each final location of the automaton generated for e to q_f, each of them releasing the variables $X_1, \ldots, X_n$. The new unique final location is q_f.

In the experiments we often used the following derived constructs:

- allocation and use: $\sharp X \stackrel{\text{def}}{=} \sharp\{X\}X$
- use and release: $X! \stackrel{\text{def}}{=} X\{X\}!$
- allocation, use and release $\sharp X! \stackrel{\text{def}}{=} \sharp\{X\}X\{X\}!$

The whole translation has been implemented in a prototype tool described in the next section.

4 Experiments

Our main objective is to develop a practical pattern matching tool for link stream analysis. An early implementation of the tool is available online[6]. In this section we present early experiments with this prototype to real-world link streams.

For starters, the worst-case complexity of our pattern matching algorithm is exponential on the size of the link stream (the number of links). This complexity is reached for instance in the case depicted in Figure 6, which is a "memory-only" scenario. If the input is a sequence of distinct symbols then the number of tokens associated to the unique location of the automaton will double each time a symbol is consumed. For instance, in the Figure, the 8 tokens are associated to distinct versions of the variable U (the U_i's) after consuming the input $a\ b\ c$: one for each subset of the alphabet.

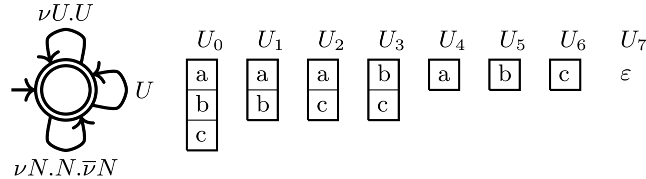


Fig. 6. A subset automaton after input $a\ b\ c$.

However, timed constraints most often improve the situation by removing expired tokens. Thus, in practice there are ways to avoid the worst-case scenarios. This is similar to the practical "regex" tools, which in general go well beyond regular expressions, also leading to exponential blowups in the worst case [4,5].

This makes experimental evaluation of our method particularly appealing to estimate its practical performances and applicability. In order to do so, we consider two link streams built from two different real-world datasets: (1) a recording of traffic routed by a large internet trans-Pacific router [8], and (2) a one month capture of tweets on Twitter France.

In the case of internet traffic, our motivation is to detect potential coordinated attacks. To do so, we define a variant of the triangle pattern discussed in section 2, namely 2x2 bicliques, i.e. squares, which [16] identified as meaningful to this regard. Since there is approximately one link every $2\mu s$ in the stream and the stream lasts for a whole day, it must be clear that we may not detect all untimed patterns in the stream. In this context, the time frame of an attack is in general quite sudden and precise, and so time is a crucial feature.

We present results for two different time frames in Figure 7. It displays the total running time as a function of the number of processed links, together with the number of found instances of the pattern. As expected, the number of instances of the pattern increases with the time frame. Also, the tool processes less links in a given amount of time (85 hours in this experiment). Although our

[6] The MaTiNa tool repository is at: https://github.com/clementber/MaTiNA

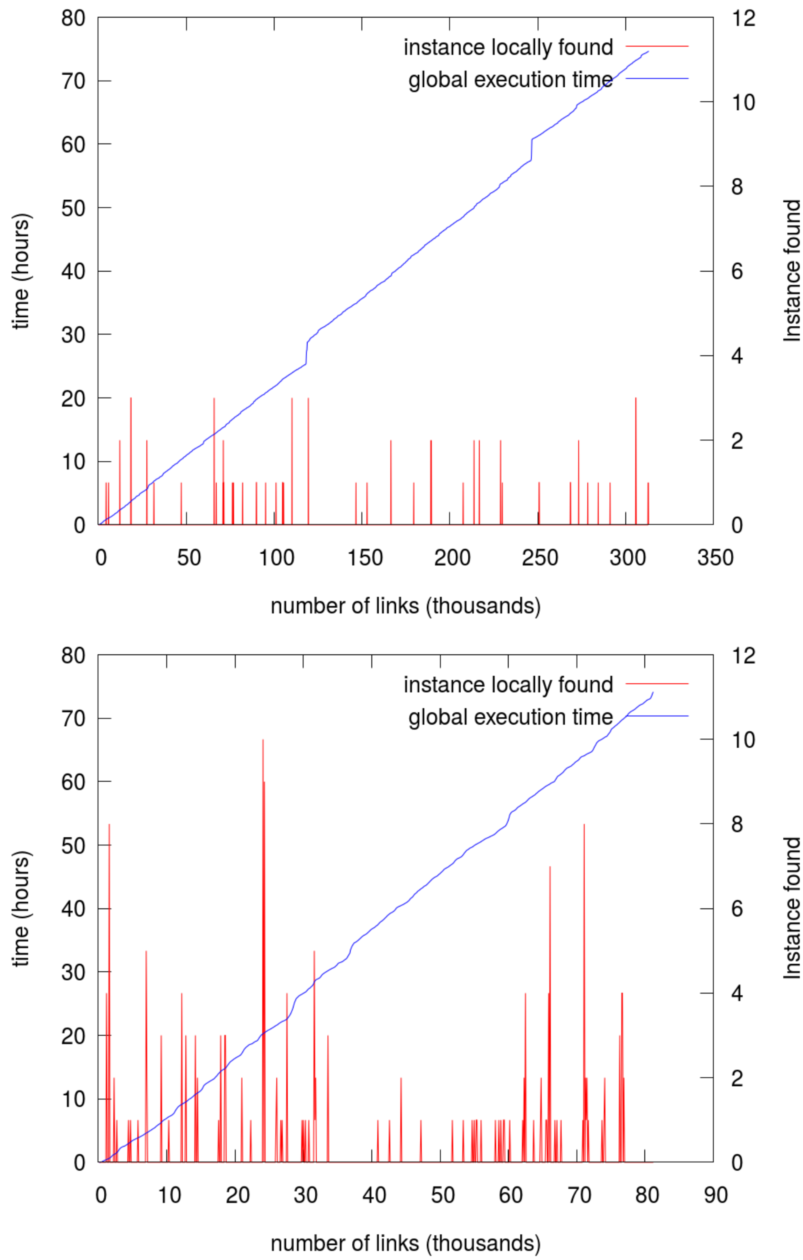


Fig. 7. DDoS pattern recognition with time frames $\delta = 0.01$ (top) and $\delta = 0.02$ (bottom).

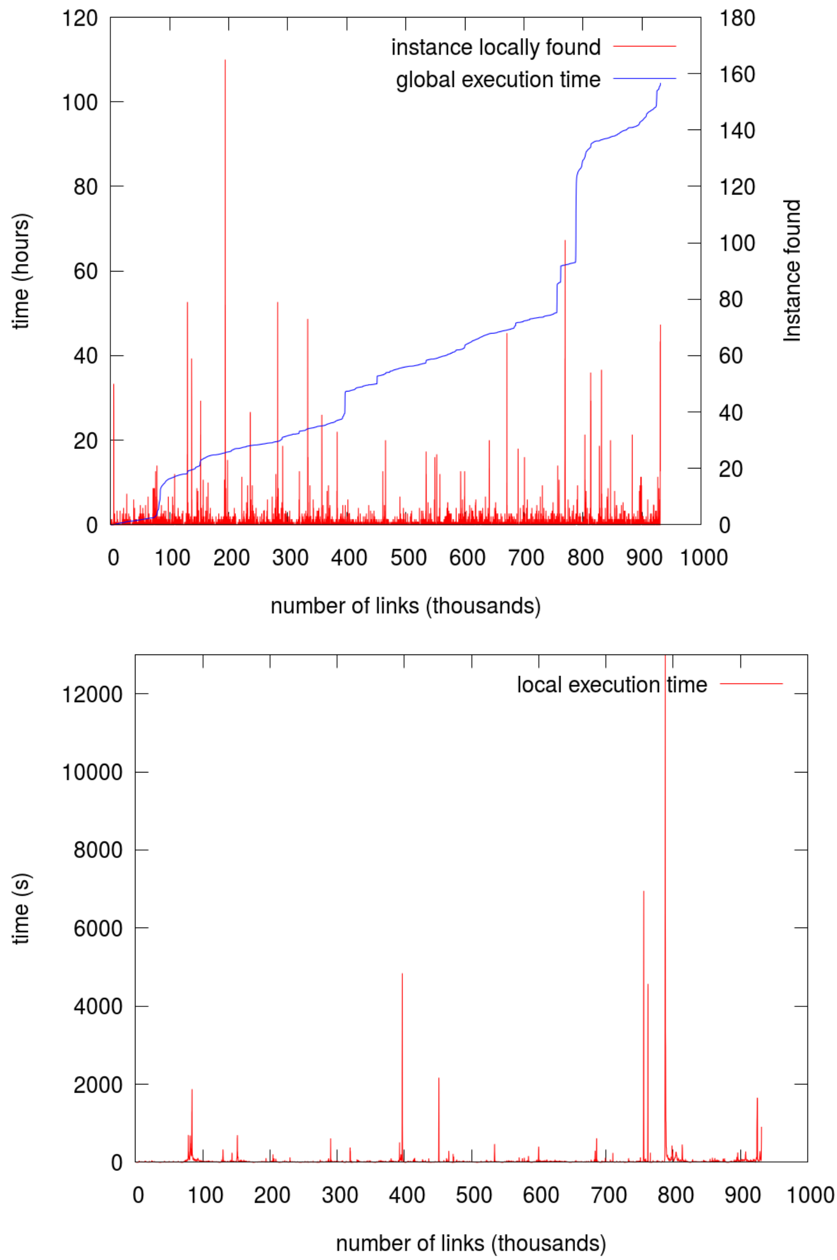


Fig. 8. Triangle detection in Twitter exchanges with running time and number of detected instances (top) and local running time (bottom).

implementation is not optimized at all, the linear time cost of the computations w.r.t. the number of processed links clearly appears.

Our second experiment targets communities of Twitter users. We consider tweets over a period of a month, leading to a stream of 1.3 million links [7]. The pattern we seek is an undirected complete graph between k users for a given k, i.e. cliques of size k occurring in a time frame of ten minutes. Figure 8 presents the results for $k = 3$, i.e. triangle detection. The running time experiences sharp increases at specific times, that correspond to peak periods in Twitter exchanges. This is confirmed by plotting the execution time at each step of the computation (right part of the Figure). During such peaks of tweets, the tool has to store more data than usual, leading to a more costly processing of links. One way to improve this issue would be to consider a variable time rate by e.g. decomposing the link stream in distinct sub-streams processed with different time frame.

5 Conclusion

The language of *timed ν-expressions* we propose to specify patterns in link streams is heavily inspired by regular expressions, but enriched with timed and memory features. The language is rather low-level but with well-chosen derived constructs we think it is usable (and has been used) by domain experts. The language has a straightforward translation to the core outcome of our research: the *timed ν-automata* formalism and the corresponding recognition principles. Beyond the formalities, we developed a functional, and freely available, prototype that we experimented in a realistic setting. Non-trivial patterns have been detected on real-world link streams, with decent performances for such an early prototype. These early experiments give us confidence regarding the relevance of our approach.

For future work, we plan both theoretical investigations and more practical work at the algorithmic and implementation level. We also expect to broaden the application domains. In particular, since our detection is performed *online*, one potential area of application is that of monitoring open systems at runtime for e.g. security or safety properties. At the theoretical level, we plan to study the pattern language and its more precise relation to the automata framework. Since the semantics are based on a token game, the formalism is in a way closed to the Petri nets than it is from classical automata. Hence, interesting extensions of the formalism could be developed based on a high-level Petri net formalism, e.g. in the spirit of [9]. Our prototype tool uses a relatively naive interpreter for pattern matching. We plan to improve its performances by first introducing a compilation step. Moreover, there is an important potential for parallelization of the underlying token game.

[7] The data come from the Politoscope project by the CNRS Institut des Systèmes Complexes Paris Ile-de-France (https://politoscope.org)

References

1. Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 147–160.
2. Rajeev Alur and D.L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.
3. Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *Journal of the ACM*, 49(2):172–206, 2002.
4. Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. A formal study of practical regular expressions. *Int. J. Found. Comput. Sci.*, 14(6):1007–1018, 2003.
5. Benjamin Carle and Paliath Narendran. On extended regular expressions. In *LATA 2009*, volume 5457 of *LNCS*, pages 279–289. Springer, 2009.
6. Aurelien Deharbe and Frédéric Peschanski. The omniscient garbage collector: A resource analysis framework. In *ACSD 2014*. IEEE Computer Society, 2014.
7. Aurélien Deharbe and Frédéric Peschanski. The Omniscient Garbage Collector: a Resource Analysis Framework. Research report, LIP6 UPMC Sorbonne Universités, France, 2014. URL: https://hal.archives-ouvertes.fr/hal-01626770.
8. Romain Fontugne, Pierre Borgnat, Patrice Abry, and Kensuke Fukuda. MAWILab: Combining Diverse Anomaly Detectors for Automated Anomaly Labeling and Performance Benchmarking. In *ACM CoNEXT '10*, 2010.
9. Vijay K. Garg and M. T. Ragunath. Concurrent regular expressions and their relationship to petri nets. *Theor. Comput. Sci.*, 96(2):285–304, 1992.
10. Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoritical Computer Science*, 134:329–363, 1994.
11. Matthieu Latapy, Tiphaine Viard, and Clémence Magnien. Stream graphs and link streams for the modeling of interactions over time. *CoRR*, abs/1710.04073, 2017. arXiv:1710.04073.
12. Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. Motifs in temporal networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, WSDM '17, pages 601–610. ACM, 2017.
13. Nikos Tzevelekos. Fresh-register automata. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT*, POPL '11, pages 295–306. ACM, 2011.
14. Dogan Ulus, Thomas Ferrère, Eugene Asarin, and Oded Maler. Timed Pattern Matching. *Formal Modeling and Analysis of Timed Systems*, 8711, 2014.
15. Dogan Ulus, Oded Maler, Eugene Asarin, and Thomas Ferrère. Online Timed Pattern Matching Using Derivatives. *LNCS*, 9636:7–8, 2016.
16. Tiphaine Viard, Raphaël Fournier-S'niehotta, Clémence Magnien, and Matthieu Latapy. Discovering patterns of interest in IP traffic using cliques in bipartite link streams. *CoRR*, abs/1710.07107, 2017.
17. Masaki Waga, Takumi Akazaki, and Ichiro Hasuo. A boyer-moore type algorithm for timed pattern matching. In *Formal Modeling and Analysis of Timed Systems*, 2016.
18. Masaki Waga, Ichiro Hasuo, and Kohei Suenaga. Efficient online timed pattern matching by automata-based skipping. In *Formal Modeling and Analysis of Timed Systems*, 2017.
19. Haopeng Zhang, Yanlei Diao, and Neil Immerman. On complexity and optimization of expensive queries in complex event processing. 2014.