Pattern Matching in Link Streams: Timed-Automata with Finite Memory

Clément BERTRAND[1], Frédéric PESCHANSKI[2],
Hanna KLAUDEL[1], Matthieu LATAPY[2]

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, difference bound matrices, 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 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

[1] IBISC, Univ Evry, Université Paris-Saclay, 91025, Evry, France, Email: {clement.bertrand,hanna.klaudel}@univ-evry.fr

[2] LIP6 – Sorbonne Université, Paris, France, Email: {matthieu latapy,frederic.peschanski}@lip6.fr

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 [14, 19] 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 sequences. 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 [15]. 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, 23]. These generic works do not handle the specificity of the input streams. For example, the real-time and graph related properties are of particular interest in 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 abstraction under study, namely the link streams.

Our starting point is the regular expressions and finite state automata for the recognition of patterns in texts. The main idea is to interpret link streams as (finite) words and develop a pattern language involving both structural and temporal features. This leads to a new kind of hybrid automata, the *timed ν-automata*, as recognizers for this pattern language. They are built upon finite state automata (FSA) with both timed [2, 3] and finite-memory [13, 8, 9] 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. [17, 18, 21, 22]. While our model bears some resemblance with these propositions, we adopt a generalized approach to temporal patterns based on the *difference bound matrix* (DBM) abstraction [10]. Moreover, we study pattern matching in the presence of real-time constraints *together with* finite memory. To our knowledge this has not been addressed in the literature.

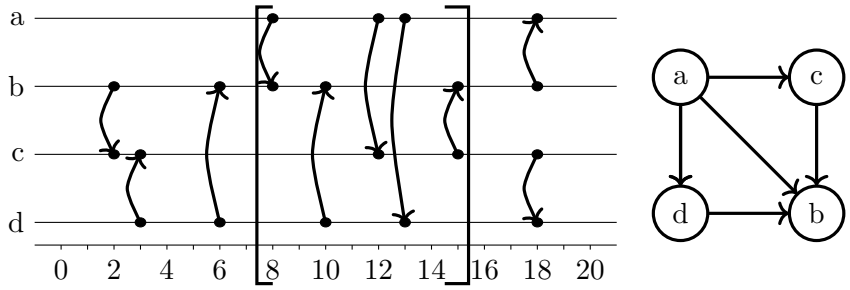


Figure 1: A link stream (left) and its graph projection in time interval $[8, 15]$ (right).

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 encouraging. 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 present paper is an extended version of [5] with a generalization of the timed model using the DBM abstraction. 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 recognition principles are formalized in Section 3. The Section 4 is entirely new and describes a timed pattern matched based on difference bound matrices. The pattern languages, the prototype tool we develop and a few experiments are then discussed in Section 5.

2 Patterns in link streams

We consider *link streams* [14] 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 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 an 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 [15].

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":

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

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:

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

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.

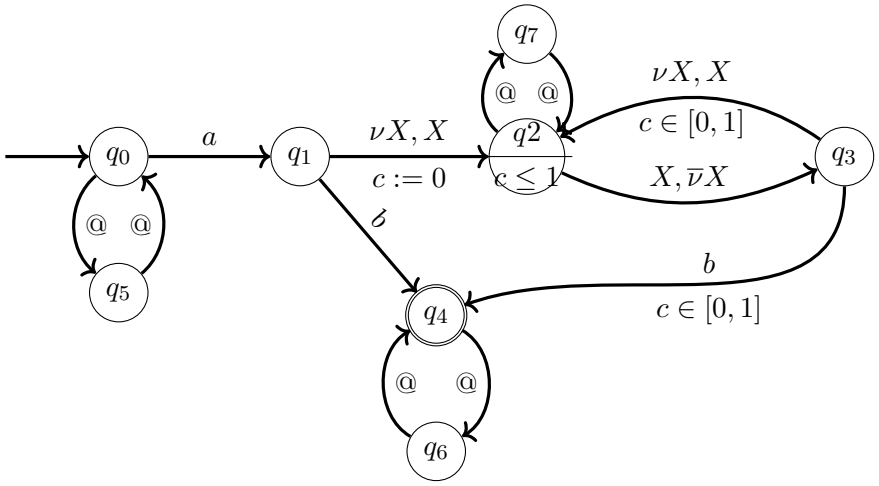


Figure 2: Automaton for $(@ \to @)^* \otimes \Big((a \to b) \vee a \to \sharp X \cdot \langle (X! \to \sharp X)^* \cdot (X! \to b)\Big)\rangle_{[0,1]}$

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) [13]. In this paper, we build upon the model of ν-automata that we developed in a previous work [8, 9]. 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* [16].

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 [9]). 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 *timed constraint*. A transition can also be annotated with time constraints to restrict its firing. The grammar of timed constraints, identical to [2], is as follows:

Definition 2 (time constraints grammar)

$$\gamma ::= \gamma \wedge \gamma \mid c_1 \sim n \mid c_1 - c_2 \sim n$$

where c_1 and c_2 are clocks, n is a constant in $\mathbb{Q}$ and $\sim \in \{=, <, >, \leq, \geq\}$.

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

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

[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 [13]), i.e., an element of the state-space: a location together with a memory content and clock values.

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 transition timed constraint is γ. 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 The state notion: tokens

States in timed ν-automata are formed by a distribution of *tokens*[4], i.e., combinations of memory and timed valuations, over given locations. Because the recognition principles we develop exploit the intrinsic non-determinism of ν-automata, each location can be associated to multiple tokens, each token corresponding to a particular reachable state[5].

Definition 4 (Token) *A* token *is a pair* $k = \langle \mathbf{D}, \mathcal{M} \rangle$ *with* $\mathbf{D}$ *a time zone representing a set of possible clock values, and* $\mathcal{M}$ *a memory valuation being a mapping from variables to sets of* allocations.

The timed valuation of a token is represented by a *timezone* $\mathbf{D}$ that encodes a set of clocks $c_1, \ldots, c_n$ (together with a special clock c_0 representing the time 0) associated to the constraints about their possible values. Following [10] we technically represent timezones as *difference bound matrices* (DBMs). Because it is a rather complex aspect, in this section we only discuss the high-level point of view of timezones, the DBM representation is detailed in Section 4.

The memory valuation $\mathcal{M}$ of a token is represented as a set $\mathcal{M}$ of *variable allocations*.

Definition 5 (Variable allocation) *For a variable* $X \in V$*, an allocation is a finite subset of unknown symbols* $\mathbb{A}_X \subset \mathcal{U}$*, together with a flag. The flag may be* $\mathbb{A}_X^\bullet$ *(read mode, default) or* $\mathbb{A}_X^\circ$ *(write mode). In read mode, the only available operation is to check if an input symbol is already present in* $\mathbb{A}_X$*. In write mode, the only available operation is to add to* $\mathbb{A}_X$ *a fresh symbol* $\alpha \notin \bigcup_{Y \in V} \mathbb{A}_Y$*.*

[4] The notion of token we use is very similar to, an in fact inspired by the corresponding notion of high-level Petri nets.

[5] In fact each state is itself a (potentially infinite) set of possible clock values corresponding to the token's timezone.

Property 1 (memory injectivity) *For any pair of distinct variables X, Y we have $\mathbb{A}_X \cap \mathbb{A}_Y = \emptyset$.*

Although most memory models do not work like this, this injectivity property is an important feature of finite-memory automata models because it allows a compact representation of memory constraints (cf. [13]). This property, from ν-automata, strengthens the memory constraint without reducing the expressibility of the model. It allows a less ambiguous description of patterns.

A *configuration* of a hybrid ν-automaton is a kind of a global state that encompasses a set of proper reachable states, thus expressing some non-determinism. Technically, the definition is as follows.

Definition 6 (Configuration) *A configuration of an automaton is a mapping S from the locations in Q to sets of tokens. We denote by $S(q)$ the set of tokens associated to location q.*

The initial configuration of a timed ν-automaton contains a single token in the initial location. The content of this token is $\langle \mathbf{D_0}, \{X \to \emptyset^\bullet | \forall X \in V\}\rangle$ where $\mathbf{D_0}$ is the initial timezone with all the clocks initialized at time 0. The memory is empty, i.e., each variable is associated to an empty set with the *read mode* flag.

Each time an input is read a new configuration is computed from the previous one. The whole input sequence is accepted if after being consumed entirely there is at least one token in some final location of the automaton. This *token game* is explained in the next section.

3.3 Token game

Given a global configuration S and an input α – either a time delay or an event (a known or an unknown symbol) – the objective is to build a next configuration S' corresponding to all the reachable states of the automaton after consuming the input. The formal definition is as follows.

Definition 7 *(global update)*

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

In the case of a time delay the tokens should be propagated through the ε-transitions, which is handled by the enabled $\sigma_{\mathsf{closure}}$ function presented below. In the case of an event, the next tokens will be produced by the transitions that are enabled for the input symbol. This is formalized by the function σ_{step} defined below. We also use the $\sigma_{\mathsf{closure}}$ function to handle the ε-transitions. The "trick" is to consider that the event is recognized together with a time delay of 0. By the non-deterministic nature of the automata model, if a token enables multiple transitions then a new token will be generated in each location reachable by all those transitions.

3.3.1 Event handling

We first consider the case of events. The time delays, a little bit more involved, will follow.

Definition 8 *(event handling)*

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

with $\delta_{step}(t,\langle \mathbf{D},\mathcal{M}\rangle,\alpha) = \langle \delta_{time}(t,\mathbf{D}), \delta_{mem}(t,\mathcal{M},\alpha)\rangle$ *when defined.*

The σ_{step} function simply consists in applying the local update function δ_{step} at all locations for all non ε-transitions. This function is partial, only defined if the subfunction δ_{time} returns a non-empty timezone, and δ_{mem} yields a value distinct from $\perp$.

The function δ_{time} computes the new timezone after crossing the considered transition. It is defined as follows.

Definition 9 *(Time constraint)*

$$\delta_{time}(q \xrightarrow[\gamma,\rho]{\nu,e,\overline{\nu}} q', \mathbf{D}) = (tz(\gamma) \cap \mathbf{D})[\rho \leftarrow 0]$$

The notation $\mathsf{tz}(\gamma)$ denotes the conversion of the time constraint γ to a corresponding timezone. We then compute the intersection of the later with the timezone $\mathbf{D}$. In the final timezone, the clock identified by ρ are reset. These operations are formalized precisely in Section 4.

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: (1) the allocation of the variables in set ν is performed, then (2) the consistency between the input and transition label is checked, and finally (3) the variables in the set $\overline{\nu}$ are released.

Definition 10 *(memory update) Let V be a set of variables, and $\mathcal{U}$ an infinite set of unknown symbols.*

$$\delta_{\mathsf{mem}}(t,\mathcal{M},\alpha) = \begin{cases} \bot \text{ if } & e \notin V \wedge \alpha \neq e & \text{(c.1.1)} \\ & \vee e \in V \wedge \alpha \notin \mathcal{U} & \text{(c.1.2)} \\ & \vee e \in V \setminus \nu \wedge \mathcal{M}(e) = \mathbb{A}_e^\bullet \wedge \alpha \notin \mathbb{A}_e & \text{(c.1.3)} \\ & \vee (e \in \nu \vee \mathcal{M}(e) = \mathbb{A}_e^\circ) \wedge \exists Y,\ \alpha \in \mathcal{M}(Y) & \text{(c.1.4)} \\ \text{otherwise } & \{X \mapsto k'_X \mid X \in V\} & \\ & \text{with } k'_X = \begin{cases} \emptyset^\bullet, \text{ if } X \in \overline{\nu} & \text{(c.2.1)} \\ (\mathbb{A}_X \cup \{\alpha\})^\bullet, \text{ if } X = e & \text{(c.2.2)} \\ \mathbb{A}_X^\circ, \text{ if } X \in \nu & \text{(c.2.3)} \\ \mathcal{M}(X), \text{ otherwise} & \text{(c.2.4)} \end{cases} & \end{cases}$$

where $e \in V \cup \Sigma \cup \{\varepsilon\}$ denotes 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 otherwise 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. Moreover, 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 $\mathbb{A}_X$ 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.

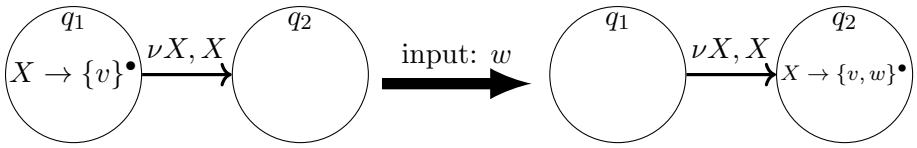


Figure 3: Passing a transition of the automaton from Figure 2 with input w

Example 1 *Figure 3 illustrates the generation of a new token taking as an example the transition $d = q_1 \xrightarrow{\{X\},X,\{\}} q_2$ in the automaton from Figure 2.*

We are focusing here only on its memory component to illustrate δ_{mem}. Based on the initial memory valuation $\{X \to \{v\}^\bullet\}$ in location q_1, the input w enables the transition d producing a new token in q_2, computed as follows:

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 $\mathbb{A}_X$ in the newly generated token. However, it is only possible if the input is fresh. Since $\mathbb{A}_X = \{v\}$ and X is the only variable, this freshness constraint is satisfied. Hence, the new token associates the memory $\{v, w\}^\bullet$ to X. ◇

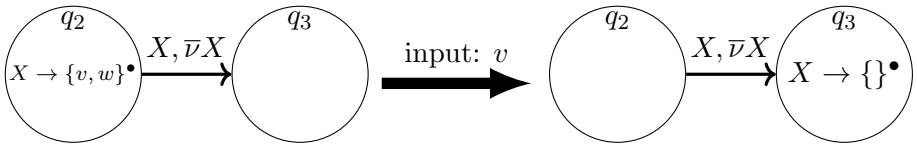


Figure 4: Passing a transition of the automaton from Figure 2 with input v

Example 2 *Figure 4 presents another example of memory transition with another transition in the same automaton from Figure 2. Here again we ignore the temporal component of the automaton to focus on its memory component. 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 $\mathbb{A}_X$. The input is actually the unknown symbol $v \notin \Sigma = \{a, b\}$. Furthermore, $v \in \{v, w\} = \mathbb{A}_X$, 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 satisfying the Property 1, and suppose $k' = \delta(t, k, \alpha) \neq \perp$ for some transition t and input α. Then the token k' will satisfy Property 1.*

Proof: In the token $k = \langle \mathbf{D}, \mathcal{M} \rangle$ only the memory component $\mathcal{M}$ is impacted by the injectivity property. The main hypothesis is that k satisfy Property 1, it means that $\forall X, Y \ (X \neq Y),\ \mathcal{M}(X) \cap \mathcal{M}(Y) = \emptyset$.

Suppose that the transition $t = q \xrightarrow[\gamma,\rho]{\nu,e,\overline{\nu}} q'$ produces token $k' = \delta(t, k, \alpha) = (\mathbf{D}', \mathcal{M}')$. We have to show that k' satisfy the Property 1. In the definition of δ_{mem} (Definition 10) we are 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 that k satisfies Property 1 and the fact that $\emptyset$ is the zero element of intersection, trivially k' satisfies Property 1 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 $\alpha \in \mathcal{M}(e)$ then the set $\mathbb{A}_e$ is not modified, so the Property 1 is trivially satisfied;
  - or $\alpha \notin \mathcal{M}(e)$ then case c.1.4 ensures that α is absent in all the domains of the other variables. Thus, Property 1 is satisfied as well.

□

3.3.2 Time delay and ε-closure

We now explain the propagation of tokens for ε-transitions and a given time delay x (a positive real value, potentially 0 in the case of an event), which is handled by the $\sigma_{\mathsf{closure}}$ function defined below. Note that it is a closure function, in that whole paths of successive ε-transitions must be considered. The rather non-trivial definition is as follows.

Definition 11 (ε-closure)

$$\sigma_{closure}(S, x) = \{q \mapsto K \mid q \in Q\}$$

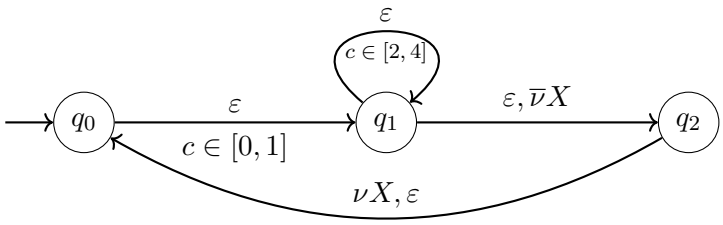
Such that

$$K = \left\{ (\mathbf{D}', \mathcal{M}') \;\middle|\; \begin{array}{l} \exists q_0 \underbrace{\xrightarrow[\gamma_1,\rho_1]{\nu_1,\varepsilon,\overline{\nu_1}}}_{t_1} q_1 \underbrace{\xrightarrow[\gamma_2,\rho_2]{\nu_2,\varepsilon,\overline{\nu_2}}}_{t_2} \cdots q_{n-1} \underbrace{\xrightarrow[\gamma_n,\rho_n]{\nu_n,\varepsilon,\overline{\nu_n}}}_{t_n} q \in \Delta, \\ \exists k \in S(q_0),\ \mathsf{walk}(k, [t_1, t_2, \ldots, t_n], x) = \langle \mathbf{D}', \mathcal{M}' \rangle \\ \wedge \neg \mathsf{empty}(\mathbf{D}') \wedge \mathcal{M}' \neq \bot \end{array} \right\}$$

For each ε-path from a location q_0 to a location q, and for each token present in q_0 in the previous configuration S, we try to generate a new token using the walk function. This new token consists of a timezone $\mathbf{D}'$ and an updated memory valuation $\mathcal{M}'$. An important requirement is that the delay x has been fully consumed at the end of the path. The function is partial, in particular it fails if x is consumed "too early" along the path. Because it involves rather complex DBM computations, the formal definition of the function walk is given in Section 4.

Example 3 *Figure 5 illustrates the dynamics of tokens in an ε-closure expressed by the function $\sigma_{\mathsf{closure}}$ with the delay $\alpha = 4$.*

The tokens in 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



#	tokens in q_0	tokens in q_1	tokens in q_2
0	$k_0{:}\langle X \to \{u, v\}^\bullet, c \to 0\rangle$		
1	$k_0{:}\langle X \to \{u, v\}^\bullet, c \to 0\rangle$	$k_1{:}\langle X \to \{u, v\}^\bullet, c \to 0\rangle$	
2	$k_0{:}\langle X \to \{u, v\}^\bullet, c \to 0\rangle$	$k_1{:}\langle X \to \{u, v\}^\bullet, c \to 0\rangle$ $k'_1{:}\langle X \to \{u, v\}^\bullet, c \to 2\rangle$	
3	$k_0{:}\langle X \to \{u, v\}^\bullet, c \to 0\rangle$	$k_1{:}\langle X \to \{u, v\}^\bullet, c \to 0\rangle$ $k'_1{:}\langle X \to \{u, v\}^\bullet, c \to 2\rangle$	$k_2{:}\langle X \to \{\}^\bullet, c \to 0\rangle$ $k'_2{:}\langle X \to \{\}^\bullet, c \to 2\rangle$
4	$k_0{:}\langle X \to \{u, v\}^\bullet, c \to 0\rangle$ $k'_0{:}\langle X \to \{\}^\circ, c \to 0\rangle$ $k''_0{:}\langle X \to \{\}^\circ, c \to 2\rangle$	$k_1{:}\langle X \to \{u, v\}^\bullet, c \to 0\rangle$ $k'_1{:}\langle X \to \{u, v\}^\bullet, c \to 2\rangle$	$k_2{:}\langle X \to \{\}^\bullet, c \to 0\rangle$ $k'_2{:}\langle X \to \{\}^\bullet, c \to 2\rangle$
5	$k_0{:}\langle X \to \{u, v\}^\bullet, c \to 0\rangle$ $k'_0{:}\langle X \to \{\}^\circ, c \to 0\rangle$ $k''_0{:}\langle X \to \{\}^\circ, c \to 2\rangle$	$k_1{:}\langle X \to \{u, v\}^\bullet, c \to 0\rangle$ $k'_1{:}\langle X \to \{u, v\}^\bullet, c \to 2\rangle$ $k''_1{:}\langle X \to \{\}^\circ, c \to 0\rangle$ $k'''_1{:}\langle X \to \{\}^\circ, c \to 2\rangle$	$k_2{:}\langle X \to \{\}^\bullet, c \to 0\rangle$ $k'_2{:}\langle X \to \{\}^\bullet, c \to 2\rangle$
6	$k_0{:}\langle X \to \{u, v\}^\bullet, c \to 4\rangle$ $k'_0{:}\langle X \to \{\}^\circ, c \to 4\rangle$	$k_1{:}\langle X \to \{u, v\}^\bullet, c \to 4\rangle$ $k''_1{:}\langle X \to \{\}^\circ, c \to 4\rangle$	$k_2{:}\langle X \to \{\}^\bullet, c \to 4\rangle$

Figure 5: Example of ε-closure with an input delay 4

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 4 to consume all the input delay.

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 $\mathbb{A}_X^a$, where $\mathbb{A}_X$ is the set associated to X and a the initial mode of X. Its reachable values in the ε-closure are :

- $\mathbb{A}_X^a$ in all ε-paths with no operations on X,
- $\mathbb{A}_X^\circ$ 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.

It is well known that the number of timed zones computable from an ε-closure is finite when the clocks have an upper bound [4]. In the case of pattern matching this bound is the sum of all delay inputted. As the number of memory states and clocks states are both finite, the number of combinations between them is finite too. □

4 Timed pattern matching with DBM

In this section we detail the core of the timed aspect of the pattern matcher. As explained in the previous section, it is based on non-trivial computation of timezones. Our approach is based on the classical model described in [10], which represents timezones as *difference bound matrices* (DBM). Unlike [10] our objective is not to develop a model-checking procedure but a recognition algorithm, hence there are many differences in the details.

4.1 The clocks representation

As explained in the previous section, a timezone corresponds to the set of all clock valuations satisfying both the possible intervals and the time constraints in a given state of the automaton. We represent a timezone as a difference bound matrix (DBM), owing to the fact that the timezones together with the time constraints expressed in the grammar of Definition 2 can be represented precisely by bounds on individual clocks and on the differences between pairs of clocks.

A DBM to represent the timezone of the set of clocks $\{c_1, c_2, ..., c_n\}$ is a matrix $\mathbf{D} = \{d_{ij}\}_{0 \leq i,j \leq n}$ where each d_{ij} is a *time bound*, i.e., an element of the set $(\mathbb{Q} \times \{<, \leq\}) \cup \{(\infty, <), (-\infty, <)\}$. Each time bound $d_{ij} = (x, \sim)$ expresses the constraint $c_i - c_j \sim x$. A DBM also requires a special clock c_0 with constant value 0 used to represent the minimal and maximal values of the other clocks. The bounds are ordered such that $d_1 < d_2$ with $d_1 = (x_1, \sim_1)$ and $d_2 = (x_2, \sim_2)$ iff $x_1 < x_2 \vee (x_1 = x_2 \wedge \sim_1 = < \wedge \sim_2 = \leq)$. We will also often use the minimum of two bounds, denoted by $\mathsf{min}(d_1, d_2)$. The ordering relation on bounds naturally extends to an inclusion ordering on DBMs.

We denote by $v \in \mathbf{D}$ the fact that a valuation $v = (v_1, v_2, ..., v_n) \in \mathbb{Q}^n$ (where each v_i is a value for clock c_i) is within the timezone represented by the matrix $\mathbf{D}$. Let $\mathbf{D} = \{(x_{ij}, \sim_{ij})\}_{0 \leq i,j \leq n}$, then $v \in \mathbf{D}$ iff $v_i - v_j \sim_{ij} x_{ij}$, $\forall i, j \in [0, n]$ (assuming $v_0 = 0$).

Three operations on DBMs defined in [10] are required by the pattern matcher. We denote by $\mathsf{empty}(\mathbf{D})$ the *emptiness predicate*, i.e., the Boolean function returning True if and only if no valuation exists in the timezone represented by $\mathbf{D}$. The *intersection* of DBMs $\mathbf{D}_1$ and $\mathbf{D}_2$, i.e., the DBM representing the intersection of the corresponding timezones, is denoted by $\mathbf{D}_1 \cap \mathbf{D}_2$. The *canonical form* $[\![\mathbf{D}]\!]$ of a DBM $\mathbf{D}$ is the strongest set of bounds defining the same timezone as $\mathbf{D}$.

The precise definitions of the operators and notations discussed above can be found in [10]. For our approach, we also need a few specific operators. First, we define the *extension* of the maximal bounds of a DBM $\mathbf{D}^1 = \{d^1_{ij}\}_{0 \leq i,j \leq n}$ to those of another DBM $\mathbf{D}^2 = \{d^2_{ij}\}_{0 \leq i,j \leq n}$. Formally we have:

$$\mathsf{ext}(\mathbf{D}^1, \mathbf{D}^2) = \left\{ d_{ij} = \left[\begin{array}{l} d^2_{i0} \text{ if } j = 0 \\ d^1_{0j} \text{ if } i = 0 \\ \text{otherwise } \mathsf{min}(d^1_{ij}, d^2_{ij}) \end{array} \right\}_{0 \leq i,j \leq n} \right.$$

One way of interpreting the definition is that $\mathsf{ext}(\mathbf{D}^1, \mathbf{D}^2)$ constructs a

path of valuations from $\mathbf{D}^1$ towards $\mathbf{D}^2$.

A second operator is the *reset*, which sets a clock value to zero and updates the DBM so that the constraints are consistent w.r.t. the new value. Formally, we write, for a DBM $\mathbf{D} = \{d_{ij}\}_{0 \leq i,j \leq n}$ and c_k a clock:

$$\mathbf{D}[c_k \leftarrow 0] = \left\{ d'_{ij} = \left[\begin{array}{l} (0, \leq) \text{ if } (i = k \wedge j = 0) \vee (i = 0 \wedge j = k) \\ d_{0j} \text{ if } i = k \wedge j \neq 0 \\ d_{i0} \text{ if } j = k \wedge i \neq 0 \\ \text{otherwise } d_{ij} \end{array} \right] \right\}_{0 \leq i,j \leq n}$$

To reset a set of clocks $\{c_1, c_2 ...\}$ we can write $\mathbf{D}^1[c_1, c_2, \ldots \leftarrow 0]$ which is equivalent to $\mathbf{D}^1[c_1 \leftarrow 0][c_2 \leftarrow 0] \ldots$

The shift operator translates a timezone according to a time delay x.

$$\mathsf{shift}(\mathbf{D}, x) = \left\{ d'_{ij} = \left[\begin{array}{l} d_{i0} + x \text{ if } i > 0, j = 0 \\ d_{0j} - x \text{ if } i = 0, j > 0 \\ \text{otherwise } d_{ij} \end{array} \right] \right\}$$

Finally, the *release* operator is used to remove the constraints concerning a clock. For a DBM $\mathbf{D} = \{d_{ij}\}_{0 \leq i,j \leq n}$ and a clock c_k:

$$\mathbf{D} \setminus c_k = \left\{ d'_{ij} = \left[\begin{array}{l} (\infty, <) \text{ if } i = k \\ (0, \leq) \text{ if } j = k \wedge i = 0 \\ d_{i0} \text{ if } j = k \wedge i \neq 0 \\ \text{otherwise } d_{ij} \end{array} \right] \right\}_{0 \leq i,j \leq n}$$

As for the reset operator, we let $\mathbf{D} \setminus \{c_1, c_2, \ldots\}$ be equivalent to $\mathbf{D} \setminus c_1 \setminus c_2 \ldots$

4.2 Handling of epsilon-paths

The function $\sigma_{\mathsf{closure}}$ of Definition 11 in Section 3 computes a next token while traversing a path of ε-transitions given a time delay x as input (with $x \geq 0$). We now explain, in terms of DBMs, the details of this computation.

Definition 12 (walking an ε-path)

$\mathit{walk}(\langle \mathbf{D}, \mathcal{M}^0 \rangle, [t_1, t_2, \ldots, t_n], x) = \langle (\mathbf{F}^n \cap \mathit{dbm}(\Gamma_{q^n})), \mathcal{M}^n \rangle$

where q^n is the arrival location of t_n, and $\mathbf{F^n}, \mathcal{M}^n$ are obtained thanks to the following iterative procedure:

$$\left[\begin{array}{ll} (\mathbf{P^0}, \mathbf{W^0}, \mathbf{F^0}) & = (\mathbf{D}, \mathbf{D}, \mathit{shift}(\mathbf{D}, x)) \\ (\mathbf{P^i}, \mathbf{W^i}, \mathbf{F^i}, \mathcal{M}^i) & = \delta_{\mathit{closure}}(t_i, \mathbf{P^{i-1}}, \mathbf{W^{i-1}}, \mathbf{F^{i-1}}, \mathcal{M}^{i-1}), i \in [1, n] \end{array} \right.$$

The function walk is based on an iteration procedure, which consists in updating a set of three distinct DBMs: **P** (past),**W** (now) and **F** (future). When a time delay x is inputted, **P** represents the initial clocks valuation, and **F** represents the expected clocks valuation after the delay x, independently of the location of the token. The third timezone **W** is used to represent the successive clock valuations between possibly several ε-transitions.

The core of the walk function is the function $\delta_{\mathsf{closure}}$ that takes as parameters a transition, the three DBMs and a memory valuation. It produces the updated DBMs and memory valuation.

Definition 13 (update) *Consider the transition* $t = q \xrightarrow[\gamma,\rho]{\nu,e,\overline{\nu}} q'$*, the three DBMs* **P**,**W** *and* **F***, and* $\mathcal{M}$ *a memory valuation, then:*

$$\delta_{closure}(t, (\mathbf{P}, \mathbf{W}, \mathbf{F}, \mathcal{M})) = (\delta_{delay}(\mathbf{P}, \mathbf{W}, \mathbf{F}), \delta_{mem}(\mathcal{M}'))$$

The memory update δ_{mem} is defined in Section 3 (cf. Definition 10). In the following, we focus on the DBM computations performed by δ_{delay}.

4.3 The time delay dataflow

Definition 14 (time update) *Let* $t = q \xrightarrow[\gamma,\rho]{\nu,e,\overline{\nu}} q'$ *be a transition, and the DBMs* $\mathbf{P}, \mathbf{W}, \mathbf{F}$*, then the function* $\delta_{delay}(t, \mathbf{P}, \mathbf{W}, \mathbf{F})$ *is computed according to the dataflow of Figure 6. This function is defined only if the whole dataflow procedure is executed.*

The partial function δ_{delay} computes, from the three inputted DBMs, the clock valuation enabling the transition and the outgoing time valuation after

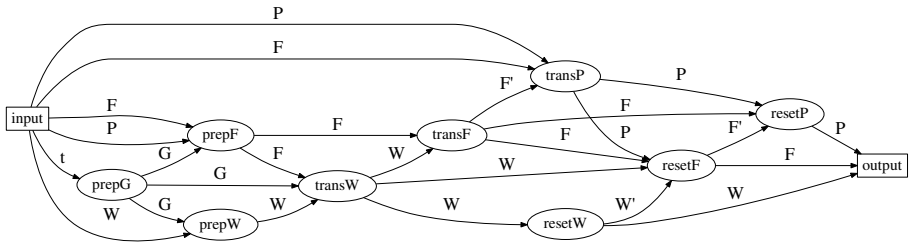


Figure 6: The dataflow for computing δ_{delay}.

the possible resets. It is structured according to the dataflow of Figure 6. From the source *input* on the left to the *output* on the right, the three DBMs $\mathbf{P}, \mathbf{W}, \mathbf{F}$ flow through a certain number of nodes. There is also a fourth DBM $\mathbf{G}$ that interprets the time constraints of the transition. This DBM is only used internally by the dataflow. Each node in the dataflow corresponds to a function that may sometimes fail to compute a value, which means that δ_{delay} is in fact not defined for the given input. Put in other terms, the considered transition is not enabled. There are three main phases in the computation. First, the *preparation step* (nodes with prefix `prep`) takes into account the coarsest constraints to reduce the input DBMs. Then, the *transition step* (suffix `trans`) takes into account the time constraints of the transition. Finally, the *reset step* (suffix `reset`) computes the output values by applying the clock resets of the transition. As shown by the diagram arrows, there are quite intricate flow dependencies between the nodes. In the remaining of the section, we will present each node function in details. But first we introduce our running example for illustrating the definitions.

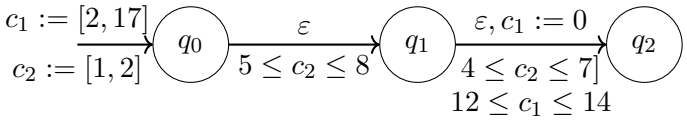


Figure 7: A simple timed automaton with two clocks and two ε-transitions.

Example 4 *We follow the crossing of the first transition on the automaton of Figure 7, without going into the details of the dataflow computation. The automaton has two clocks c_1 and c_2, and the initial timezone is represented by the following DBM:*

$$\mathbf{D} = \begin{pmatrix} (0,\leq) & (-2,\leq) & (-1,\leq) \\ (17,\leq) & (0,\leq) & (16,\leq) \\ (2,\leq) & (0,\leq) & (0,\leq) \end{pmatrix}$$

We assume that the inputted time delay is 9 seconds, hence the iterations of walk *begin with the following DBMs:*

$$\mathbf{P^{(0)}} = \mathbf{D};\ \ \mathbf{W^{(0)}} = \mathbf{D};\ \ \mathbf{F^{(0)}} = \mathit{shift}(\mathbf{D}, 9)$$

Now, we let $(\mathbf{P^{(1)}}, \mathbf{W^{(1)}}, \mathbf{F^{(1)}}, \mathcal{M}^{(1)}) = \delta_{\mathit{delay}}(t_1, (\mathbf{P^{(0)}}, \mathbf{W^{(0)}}, \mathbf{F^{(0)}}))$ for the first transition $t_1 = q_0 \xrightarrow[5\leq c_2\leq 8]{\varepsilon} q_1$. The updated timezones are the

following : $\mathbf{P^{(1)}} = \mathbf{P^{(0)}}$ *and* $\mathbf{F^{(1)}} = \mathbf{F^{(0)}}$ *and*

$$\mathbf{W^{(1)}} = \begin{pmatrix} (0,\leq) & (-4,\leq) & (-4,\leq) \\ (22,\leq) & (0,\leq) & (16,\leq) \\ (6,\leq) & (0,\leq) & (0,\leq) \end{pmatrix}$$

The further examples will detail the crossing of the second transition from this first application of δ_{delay}.

Preparation phase The node prep_G of the dataflow computes the DBM $\mathbf{G}$ corresponding to the global time constraint. It is defined as follows:

$$\mathsf{prep}_G(q \xrightarrow[\gamma,\rho]{\nu,e,\overline{\nu}} q') = [\![\mathsf{tz}(\gamma) \cap \mathsf{tz}(\Gamma_q) \cap (\mathsf{tz}(\Gamma_{q'}) \setminus \rho)]\!]$$

The DBM $\mathbf{G}$ is the time zone corresponding to all the constraints the clocks have to satisfy in order to enable the transition from q to q'. The constraints from the timed constraint of q' concerning the reset clocks are released because the value enabling the transition are not the ones entering q' for this clocks.

In the preparation phase, we also need to filter out the valuations that would contradict the global constraint. We remove from $\mathbf{P}$ and $\mathbf{W}$ the valuations with at least one clock value above the maximal value of the same clock in $\mathbf{G}$. Symmetrically, the invalid valuations of $\mathbf{F}$ are those with at least one clock below the minimal value in $\mathbf{G}$. More formally, we have:

$$\mathsf{prep}_W(\mathbf{W},\mathbf{G}) = \left[\!\!\left[\left\{ w'_{ij} = \begin{bmatrix} \mathsf{min}(w_{i0}, g_{i0}) \text{ if } j = 0 \\ \text{otherwise } w_{ij} \end{bmatrix} \right\}_{0\leq i,j\leq n} \right]\!\!\right]$$

$$\mathsf{prep}_F(\mathbf{F},\mathbf{G},\mathbf{P}) = \left[\!\!\left[\left\{ f'_{ij} = \begin{bmatrix} \mathsf{min}(f_{ij}, g_{ij}) \text{ if } j \neq 0 \\ f_{i0} - p_{i0} + g_{i0} \text{ if } j = 0, g_{i0} < p_{i0} \\ \text{otherwise } f_{ij} \end{bmatrix} \right\}_{0\leq i,j\leq n} \right]\!\!\right]$$

Note that the timezone $\mathbf{P}$ and $\mathbf{F}$ are linked as they represent respectively the initial and final valuation of the clocks. If one is changed then the other must also be updated. However, we do not need to add a preparation node for $\mathbf{P}$ because it is not needed for the next phase, moreover the computation would be redundant. However in the example below we will show how $\mathbf{P}$ and $\mathbf{F}$ are synchronized for illustration purpose.

All the DBMs computed in this phase and most of those computed in the following phases are canonicalized (using operator $[\![\cdot]\!]$). This ensures that

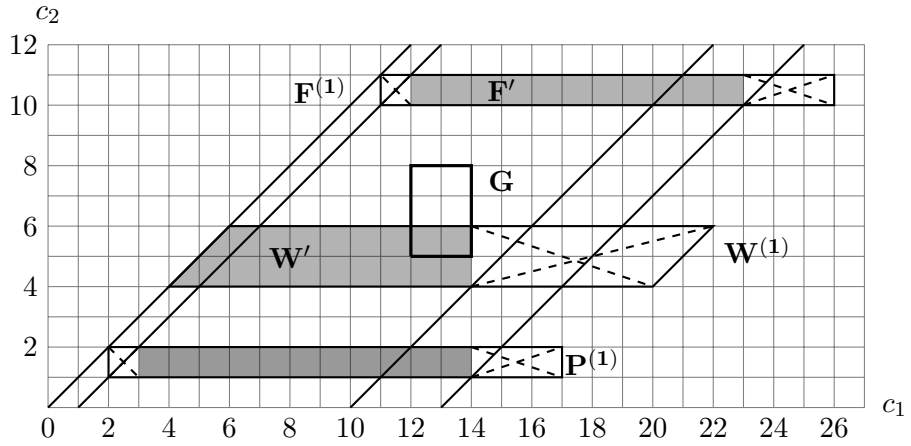


Figure 8: Illustrating the preparation phase of δ_{delay} applied on t_2 from the automaton of Figure 7.

no precision loss may be propagated during the computation. Only $\mathbf{P}$ is not canonicalized as it is synchronized on $\mathbf{F}$, which is already in canonical form. Furthermore, the only values used in $\mathbf{P}$ are the individual clocks' intervals (p_{i0} and p_{0j} for all i, j).

Example 5 *Figure 8 illustrates the preparation phase of* $\delta_{delay}(t_2, \mathbf{P^{(1)}}, \mathbf{W^{(1)}}, \mathbf{F^{(1)}})$. *Since the automaton only uses two clocks, the DBMs can be represented as polyhedra in a 2-dimensional space. The global constraint* $\mathbf{G} = prep_G(t_2)$ *is depicted as a rectangle in the middle of the figure. The initial DBMs* $\mathbf{P^{(1)}}, \mathbf{W^{(1)}}, \mathbf{F^{(1)}}$ *are depicted as polyhedra outlines. The filled zones correspond to results of the preparation functions.*

First, we define $\mathbf{W'} = prep_W(\mathbf{W^{(1)}}, \mathbf{G})$, *which removes the right part of* $\mathbf{W^{(1)}}$ *(shown as a barred area). Indeed, each valuation of* $\mathbf{W^{(1)}}$ *where* $c_1 > 14$ *must be filtered out as* 14 *is the upper bound of* c_1 *in* $\mathbf{G}$. *In the case of the "future" DBM we define* $\mathbf{F'} = prep_F(\mathbf{F^{(1)}}, \mathbf{G}, \mathbf{P^{(1)}})$. *The valuation of* $\mathbf{F^{(1)}}$ *where* $c_1 < 11$ *must be removed because the lowest value of* c_1 *in* $\mathbf{G}$ *is* 11. *This corresponds to the barred area on the left of* $\mathbf{F'}$ *on the figure. If we actually computed the current update for* $\mathbf{P}$ *(the filled zone at the bottom of the picture), then we would have to remove the corresponding barred area. Symmetrically, the barred area on the right of the updated* $\mathbf{P}$ *is also removed on* $\mathbf{F'}$ *side*[6].

[6]As we already explained we do not have to actually compute the update timezone of

The canonical form of both $\mathbf{W}'$ *and* $\mathbf{F}'$ *is computed to get the most precise values for the two-clocks constraints, i.e., the constraints corresponding to pairs of clocks (the diagonals in Figure 8). For* $\mathbf{F}'$ *we get* $-1 \leq c_1 - c_2 \leq 13$. *And for* $\mathbf{W}'$ *we get* $0 \leq c_1 - c_2 \leq 10$. *At the end of the first step we have:*

$$\mathbf{W}' = \begin{pmatrix} (0,\leq) & (-4,\leq) & (-4,\leq) \\ (14,\leq) & (0,\leq) & (10,\leq) \\ (6,\leq) & (0,\leq) & (0,\leq) \end{pmatrix} \text{ and}$$

$$\mathbf{F}' = \begin{pmatrix} (0,\leq) & (-12,\leq) & (-10,\leq) \\ (23,\leq) & (0,\leq) & (13,\leq) \\ (11,\leq) & (-1,\leq) & (0,\leq) \end{pmatrix}$$

Transition phase In the next step, we actually "enter" the transition by first updating the timezones so that it is enabled. Moreover, the values unreachable after the transition are filtered out. We first consider the update of $\mathbf{W}$, as follows:

$$\mathsf{trans}_W(\mathbf{W},\mathbf{F},\mathbf{G}) = [\![(\mathsf{ext}(\mathbf{W},\mathbf{F}) \cap \mathbf{G})]\!]$$

This function computes the reachable clock valuations consisting in extending the elements of $\mathbf{W}$ toward their final position in $\mathbf{F}$. In the extension only the elements satisfying the global time constraints $\mathbf{G}$ are preserved. If the result of trans_W is an empty DBM, this means a time constraint is not satisfied, thus the transition is not enabled (and the whole dataflow execution fails).

$$\mathsf{trans}_F(\mathbf{F},\mathbf{W}') = \left[\!\!\left[f'_{ij} = \left\{ \begin{array}{l} \mathsf{min}(w'_{ij}, f_{ij}) \text{ if } i,j \neq 0 \\ \text{otherwise } f_{ij} \end{array} \right\}_{0 \leq i,j \leq n} \right]\!\!\right]$$

$$\mathsf{trans}_P(\mathbf{P},\mathbf{F},\mathbf{F}') = \left\{ p'_{ij} = \left[\begin{array}{l} p_{ij} - f_{ij} + f'_{ij} \\ \qquad \text{if } (i = 0 \vee j = 0) \wedge p_{i0} \neq (0,\leq) \\ \text{otherwise } p_{ij} \end{array} \right. \right\}_{0 \leq i,j \leq n}$$

The two-clocks time constraints of the DBM returned by trans_W are applied on $\mathbf{F}$ to remove its unreachable values. The result is put in canonical form because the actual maximal and minimal values of each clock must be known before the next phase. The function trans_P is used to synchronize

$\mathbf{P}$.

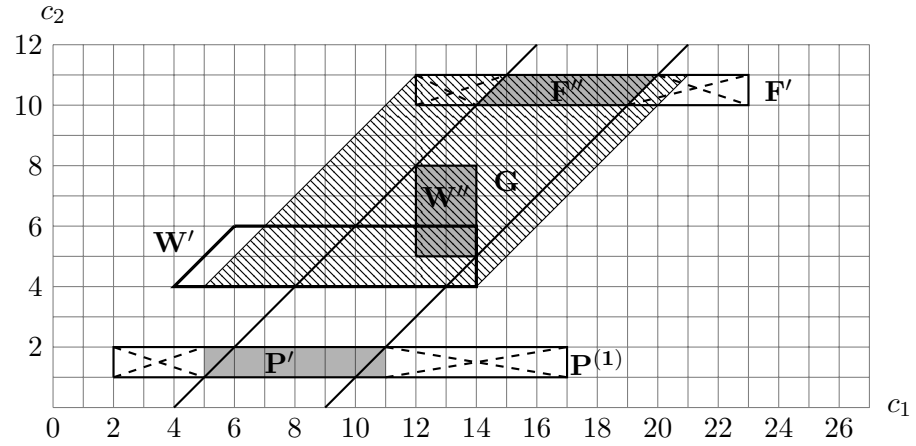


Figure 9: Illustrating the transition phase of δ_{delay}.

$\mathbf{P}$ with the DBM returned by trans_F. As depicted on the dataflow (cf. 6), the parameter $\mathbf{F}$ is the initial DBM whereas $\mathbf{F'}$ is the output of trans_F. The function removes from $\mathbf{P}$ the area of $\mathbf{F}$ absent in $\mathbf{F'}$, which we illustrate in the next step of the example.

Example 6 *Figure 9 represents the transition phase of δ_{delay}. At the beginning, the DBMs $\mathbf{W'}$ and $\mathbf{F'}$ resulting from the preparation phase, together with $\mathbf{P}^{(1)}$ are represented by their outline. The DBM $\mathbf{W''} = \mathsf{trans}_W(\mathbf{W'}, \mathbf{F'}, \mathbf{G})$ is represented as a filled area, which corresponds to the extension of $\mathbf{W'}$ towards $\mathbf{F'}$ (the dotted area on the figure), intersected with the global constraint $\mathbf{G}$.*

$$\begin{array}{ccccc} ext(\mathbf{W'}, \mathbf{F'}) & \cap & \mathbf{G} & = & \mathbf{W''} \\ \begin{pmatrix} (0,\leq),(-4,\leq),(-4,\leq) \\ (23,\leq),(0,\leq),(10,\leq) \\ (11,\leq),(-1,\leq),(0,\leq) \end{pmatrix} & \cap & \mathbf{G} & = & \begin{pmatrix} (0,\leq),(-12,\leq),(-5,\leq) \\ (14,\leq),(0,\leq),(9,\leq) \\ (8,\leq),(-4,\leq),(0,\leq) \end{pmatrix} \end{array}$$

Once $\mathbf{W''}$ is computed, we can define $\mathbf{F''} = \mathsf{trans}_F(\mathbf{F'}, \mathbf{W''})$, which consists in removing the valuations of $\mathbf{F'}$ that are not reachable from $\mathbf{W''}$. Finally, the computation of $\mathbf{P'} = trans_P(\mathbf{P}^{(1)}, \mathbf{F}^{(1)}, \mathbf{F''})$ consists in comparing the definition interval of each clock in $\mathbf{F}$ and $\mathbf{F''}$, and subtracting the difference to the bounds on $\mathbf{P}^{(1)}$.

$$\mathbf{F''} = \begin{pmatrix} (0,\leq),(14,\leq),(-10,\leq) \\ (20,\leq),(0,\leq),(9,\leq) \\ (11,\leq),(-4,\leq),(0,\leq) \end{pmatrix}; \mathbf{P'} = \begin{pmatrix} (0,\leq),(-5,\leq),(-1,\leq) \\ (11,\leq),(0,\leq),(16,\leq) \\ (2,\leq),(0,\leq),(0,\leq) \end{pmatrix}$$

Reset phase The final step consists in resetting the clocks present in ρ. The three nodes of this part of the dataflow produce the outputs of δ_{delay}. For the timezone $\mathbf{W}$ the computation is straightforward:

$$\mathsf{reset}_W(\mathbf{W}, \rho, q') = [\![\mathbf{W}[\rho \leftarrow 0] \cap \mathsf{tz}(\Gamma_{q'})]\!]$$

The resulting timezone corresponds to the clock valuations entering the arrival location q'. These are the valuations of $\mathbf{W}$ where the clocks of ρ are reset, and such that the invariant $\Gamma_{q'}$ is satisfied.

The computation performed by the node reset_F is a little bit more involved. The definition is as follows:

$$\mathsf{reset}_F(\mathbf{F}, \rho, \mathbf{W}, \mathbf{W}', \mathbf{P}) =$$
$$\left[\!\!\left[\left\{ f'_{ij} = \begin{cases} \mathsf{min}(\{f_{k0} + w'_{0k} | 1 \le k \le n\} \cup \{f_{k0} - p_{k0} | 1 \le k \le n\}) \\ \quad \text{if } c_i \in \rho, c_j = c_0 \\ \mathsf{min}(\{(0, \le)\} \cup \{f_{0k} + w'_{k0} | 1 \le k \le n\}) \\ \quad \text{if } c_i = c_0, c_j \in \rho \\ w'_{ij} \text{ if } i, j \ne 0 \\ \text{otherwise } f_{ij} \end{cases} \right\}_{0 \le i,j \le n} \right]\!\!\right]$$

The function needs both the DBM $\mathbf{W}$ outputted by trans_W and $\mathbf{W}'$ as returned by reset_W. It returns the timezone representing all the valuations reachable in the destination location q' independently of the timed constraint of q'. To compute it from $\mathbf{F}$ (result of trans_F) we have to find the intervals of definition for all the reset clocks, and then restrict the timezone with the time constraint of $\mathbf{W}'$. All the reset clocks have the same maximal and minimal valuations: $\forall i, j \in \rho, f'_{i0} = f'_{j0} \wedge f'_{0i} = f'_{0j}$. This value is the maximum (resp. minimum) distance between the values in $\mathbf{W}$ and their corresponding final position in $\mathbf{F}$. We have to make sure that this distance is not greater than the maximal distance between a point of $\mathbf{P}$ (from trans_P) and the corresponding point in $\mathbf{F}$. The non-reset clocks keep their previous maximal and minimal values. In case the resulting DBM is empty, the dataflow execution is considered failed.

Finally, reset_P will generate the timezone corresponding to $\mathbf{P}$ after the resets.

$$\mathsf{reset}_P(\mathbf{P}, \rho, \mathbf{F}, \mathbf{F}') =$$
$$\left\{ p'_{ij} = \begin{cases} (0, \le) \text{ if } (i = 0, j \in \rho) \vee (i \in \rho, j = 0) \\ p_{ij} - f_{ij} + f'_{ij} \text{ if } i = 0 \vee j = 0, p_{i0} \ne (0, \le) \\ \text{otherwise } p_{ij} \end{cases} \right\}_{0 \le i,j \le n}$$

The DBM $\mathbf{F}$ is the result of trans_F and the DBM $\mathbf{F}'$ is the result of reset_F.

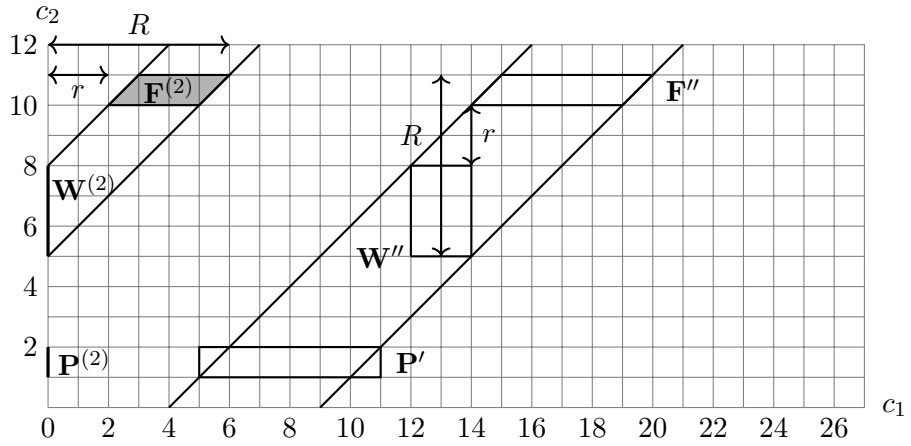


Figure 10: Illustrating the reset phase of δ_{delay}.

All the clocks of ρ have their maximal and minimal values set to zero. A clock of $\mathbf{P}$ of value zero is only used by reset_F to get the upper bound on the maximal value of the reset clocks. For the non-reset clocks, their maximal and minimal values are restricted in order to keep the corresponding bound with the ones returned by reset_F, as required by the preparation phase of the (potential) next transition. This is done by comparing them with the bound of $\mathbf{F}$ (from trans_F).

Example 7 *Figure 10 depicts the final reset phase, the computation of the zone outputted by δ_{delay}. The zones resulting from the transition phase are depicted by their respective outline. The timezones $\mathbf{W}^{(\mathbf{2})} = \mathsf{reset}_W(\mathbf{W}'', \rho, q')$ and $\mathbf{P}^{(\mathbf{2})} = \mathsf{reset}_P(\mathbf{P}, \rho, \mathbf{F}, \mathbf{F}')$ are flattened on the c_2 axis because the clock c_1 is reset. The timezone $\mathbf{F}^{(\mathbf{2})} = \mathsf{reset}_F(\mathbf{F}'', \rho, \mathbf{W}'', \mathbf{W}^{(2)}, \mathbf{P}')$ is depicted by the filled area on the picture. We first compute the maximal and minimal allowed values for each clock. The non-reset clocks keep the same bounds as the ones from $\mathbf{F}''$. For the reset clocks, we have to compute the remaining time interval represented on the picture with the arrows $[r, R]$. This correspond to the minimal, resp. maximal, distance separating a valuation of $\mathbf{W}''$ to its corresponding "future" in $\mathbf{F}''$. However, the maximal value cannot be greater than the distance separating an element of $\mathbf{P}'$ of its corresponding element in $\mathbf{F}''$ to avoid a situation in which the remaining time is greater than the initial delay. The two-clock relations are the ones from $\mathbf{W}^{(\mathbf{2})}$. Finally, its canonical form is computed to have the exact bound of values for each clock.*

$$\mathbf{P}^{(\mathbf{2})} = \begin{pmatrix} (0,\leq),(0,\leq),(-1,\leq) \\ (0,\leq),(0,\leq),(16,\leq) \\ (2,\leq),(0,\leq),(0,\leq) \end{pmatrix}; \mathbf{W}^{(\mathbf{2})} = \begin{pmatrix} (0,\leq),(0,\leq),(-5,\leq) \\ (0,\leq),(0,\leq),(-5,\leq) \\ (8,\leq),(8,\leq),(0,\leq) \end{pmatrix}$$

$$\mathbf{F}^{(\mathbf{2})} = \begin{pmatrix} (0,\leq),(-2,\leq),(-10,\leq) \\ (6,\leq),(0,\leq),(-5,\leq) \\ (11,\leq),(8,\leq),(0,\leq) \end{pmatrix}$$

5 Pattern language and experiments

5.1 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)			$e_1 \cdot e_2$	(concatenation)
			$e_1 \mid e_2$	(disjunction)
			$e_1 \otimes e_2$	(shuffle)
			e^*	(iteration)
(time)			$\langle e \rangle_{[x,y]}$	(delay[7])
(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, unknown or arbitrary nodes. The link construct $n_1 \to n_2$ describes

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

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. A special case is the link expression $n_1 \to n_2$ that corresponds to a basic automaton with three locations and two transitions in a row, 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). Note that the translation is relatively straightforward. The translation rules for the regular expression constructs are the classical ones. The function $\mathsf{aut}: \mathit{expression} \to \mathit{automaton}$ translates a timed ν-expression to the corresponding timed ν-automaton.

Figure 11 illustrates the translation for some notable operators of the language from [3]: delay and concatenation, which are impacted by the time component.

To translate the delay operator $\langle e \rangle_I$, we first need to generate the automaton of the constrained sub-expression e. Then we create a new clock c dedicated to measure the time for the new constraint. Finally, all transitions to a final location of the automaton have their timed constraints strengthened with the constraint $c \in I$.

The timed aspect of the concatenation operator $e_1 \cdot e_2$ consists in resetting all the clocks in order to initialize the checking of the timed constraints. Its translation is mostly the same as for regular expressions: each of the sub-expressions, e_1 and e_2, is translated to an automaton (resp. $\mathcal{A}_1$ and $\mathcal{A}_2$) and new automaton $\mathcal{A}$ is created containing all the locations of $\mathcal{A}_1$ and $\mathcal{A}_2$, all their transitions and clocks. The initial location of $\mathcal{A}$ is the initial location of $\mathcal{A}_1$, and its final locations are those of $\mathcal{A}_2$. Moreover, for each transition of $\mathcal{A}_1$ going to one of its final locations, the equivalent transition but with the initial location of $\mathcal{A}_2$ as destination and resetting all the clocks of C_2 is added to $\mathcal{A}$.

Figure 12 illustrates how the allocation and release operators are translated. 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

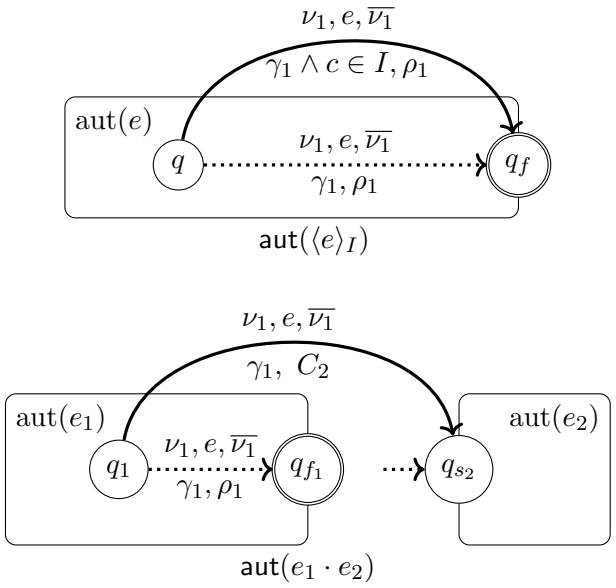


Figure 11: Automata for the delay and concatenation operations.

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 $\sharp X! \stackrel{\text{def}}{=} \sharp\{X\}X\{X\}!$

5.2 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[8]. In this section we present early experiments with this prototype to real-world link streams.

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

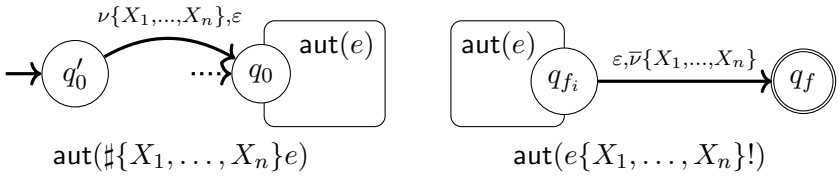


Figure 12: Automata for memory operators

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 13, 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.

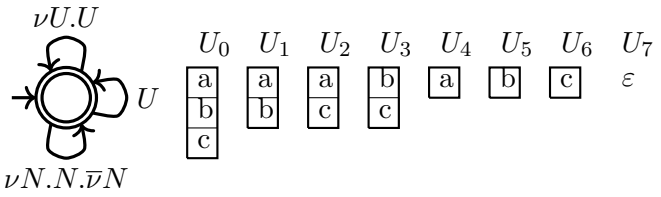


Figure 13: A subset automaton after input a b c.

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 [6, 7].

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 [11], 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 [20] identified as meaningful to this regard. Since there is approximately one link every

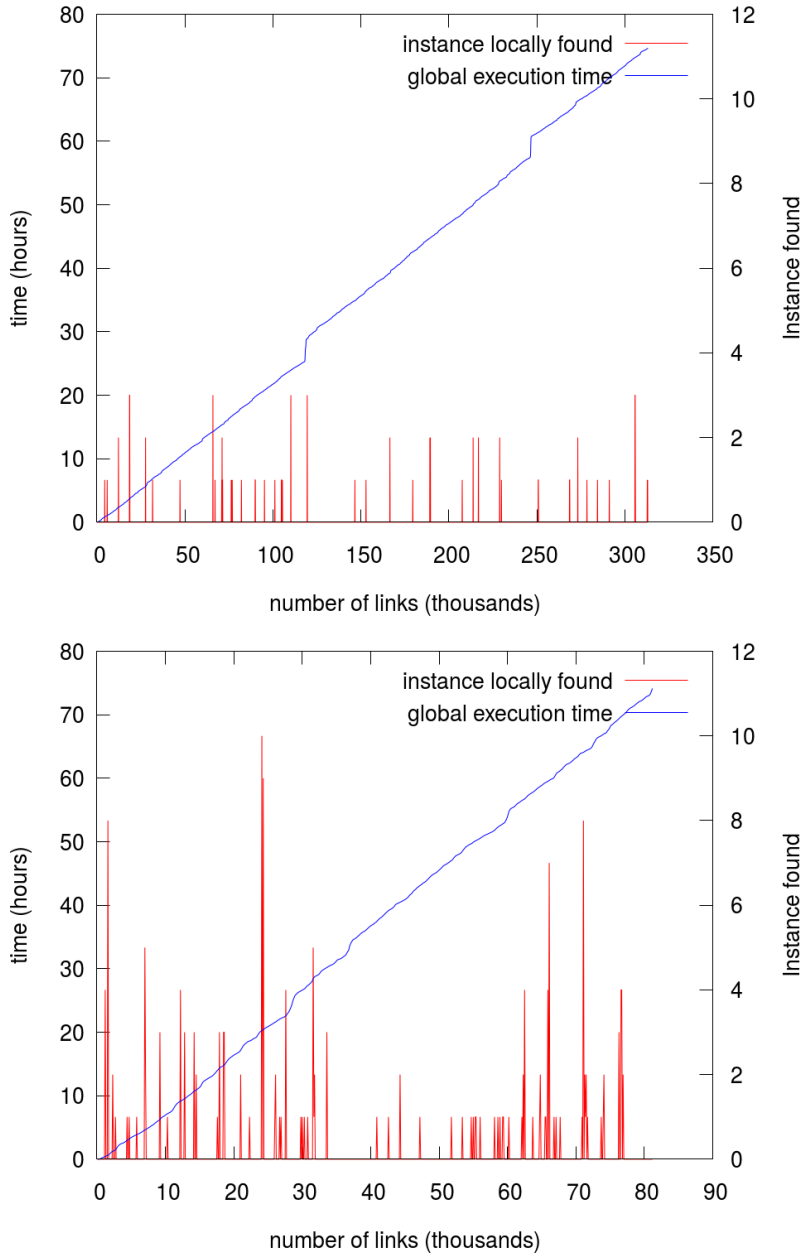


Figure 14: DDoS pattern recognition with time frames $\delta = 0.01$ (top) and $\delta = 0.02$ (bottom).

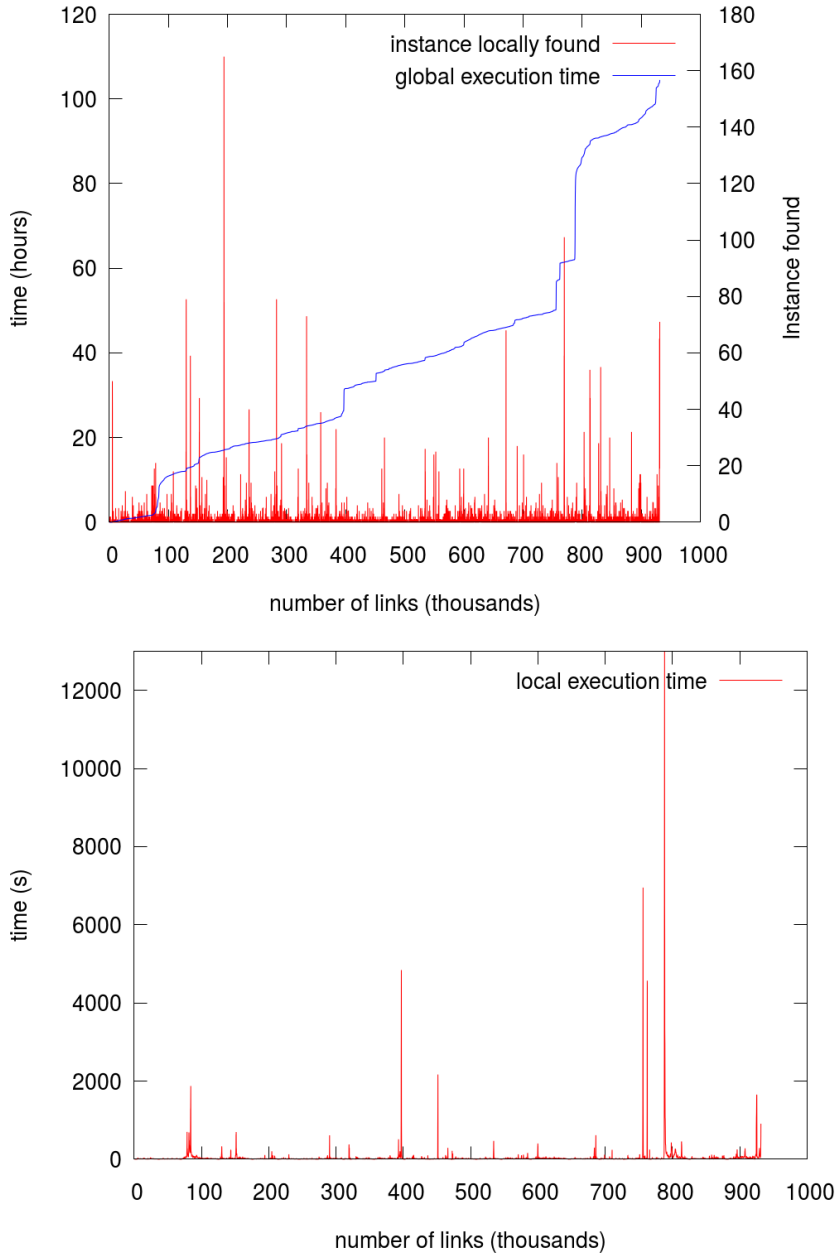


Figure 15: Triangle detection in Twitter exchanges with running time and number of detected instances (top) and local running time (bottom).

$2\mu s$ 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 14. 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 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 [9]. 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 15 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.

6 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. Compared to the conference paper, this extended version presents a generalized version of the time component of timed ν-automata. The clocks are now represented using timezones, modeled as Difference Bound Matrices. Using them, we have formalized the timed

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

pattern matching dynamics. This extension allows in particular unrestricted resets on ε-transitions and enhances the language expressiveness. However, in order to fully exploit in practice this semantical extension it would be necessary to add some dedicated operators in the syntax of the language.

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 closer 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 [12]. 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.

References

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 147–160.

[2] R. Alur and D. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[3] E. Asarin, P. Caspi, and O. Maler. Timed regular expressions. *Journal of the ACM*, 49(2):172–206, 2002.

[4] J. Bengtsson and W. Yi. *Timed Automata: Semantics, Algorithms and Tools*, pages 87–124. Springer Berlin Heidelberg, Berlin, Heidelberg.

[5] C. Bertrand, H. Klaudel, F. Peschanski, and M. Latapy. Pattern matching in link streams: a token-based approach. In *Petri Nets 2018*, 2018. To appear.

[6] C. Câmpeanu, K. Salomaa, and S. Yu. A formal study of practical regular expressions. *Int. J. Found. Comput. Sci.*, 14(6):1007–1018, 2003.

[7] B. Carle and P. Narendran. On extended regular expressions. In *LATA 2009*, volume 5457 of *LNCS*, pages 279–289. Springer, 2009.

[8] A. Deharbe and F. Peschanski. The omniscient garbage collector: A resource analysis framework. In *ACSD 2014*. IEEE Computer Society, 2014.

[9] A. Deharbe and F. Peschanski. The Omniscient Garbage Collector: a Resource Analysis Framework. Research report, LIP6 UPMC Sorbonne Universités, France, 2014.

[10] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, pages 197–212. Springer Berlin Heidelberg, 1990.

[11] R. Fontugne, P. Borgnat, P. Abry, and K. Fukuda. MAWILab: Combining Diverse Anomaly Detectors for Automated Anomaly Labeling and Performance Benchmarking. In *ACM CoNEXT '10*, 2010.

[12] V. K. Garg and M. T. Ragunath. Concurrent regular expressions and their relationship to petri nets. *Theor. Comput. Sci.*, 96(2):285–304, 1992.

[13] M. Kaminski and N. Francez. Finite-memory automata. *Theoritical Computer Science*, 134:329–363, 1994.

[14] M. Latapy, T. Viard, and C. Magnien. Stream graphs and link streams for the modeling of interactions over time. *CoRR*, abs/1710.04073, 2017.

[15] A. Paranjape, A. R. Benson, and J. 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.

[16] N. Tzevelekos. Fresh-register automata. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT*, POPL '11, pages 295–306. ACM, 2011.

[17] D. Ulus, T. Ferrère, E. Asarin, and O. Maler. Timed Pattern Matching. *Formal Modeling and Analysis of Timed Systems*, 8711, 2014.

[18] D. Ulus, O. Maler, E. Asarin, and T. Ferrère. Online Timed Pattern Matching Using Derivatives. *LNCS*, 9636:7–8, 2016.

[19] T. Viard, R. Fournier, C. Magnien, and M. Latapy. Discovering Patterns of Interest in IP Traffic Using Cliques in Bipartite Link Streams. In *(CompleNet'18) International Conference on Complex Networks*, pages 233–241, Mar. 2018.

[20] T. Viard, R. Fournier-S'niehotta, C. Magnien, and M. Latapy. Discovering patterns of interest in IP traffic using cliques in bipartite link streams. *CoRR*, abs/1710.07107, 2017.

[21] M. Waga, T. Akazaki, and I. Hasuo. A boyer-moore type algorithm for timed pattern matching. In *Formal Modeling and Analysis of Timed Systems*, 2016.

[22] M. Waga, I. Hasuo, and K. Suenaga. Efficient online timed pattern matching by automata-based skipping. In *Formal Modeling and Analysis of Timed Systems*, 2017.

[23] H. Zhang, Y. Diao, and N. Immerman. On complexity and optimization of expensive queries in complex event processing. 2014.