Layered Memory Automata: Recognizers for Quasi-Regular Languages with Unbounded Memory

Clément Bertrand, Hanna Klaudel, Frédéric Peschanski

▶ **To cite this version:**

Clément Bertrand, Hanna Klaudel, Frédéric Peschanski. Layered Memory Automata: Recognizers for Quasi-Regular Languages with Unbounded Memory. 43rd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2022), Jun 2022, Bergen, Norway. pp.43-63, 10.1007/978-3-031-06653-5_3 . hal-03713609

HAL Id: hal-03713609

https://univ-evry.hal.science/hal-03713609

Submitted on 10 Aug 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Layered Memory Automata: Recognizers for Quasi-Regular Languages with Unbounded Memory

Clément Bertrand[1(✉)], Hanna Klaudel[2], and Frédéric Peschanski[3]

[1] Scalian Digital Systems, Valbonne, France
clement.bertrand@scalian.com
[2] IBISC, Univ. Evry, Université Paris-Saclay, Évry, France
hanna.klaudel@univ-evry.fr
[3] LIP6, Sorbonne Université, CNRS UMR7606, Paris, France
frederic.peschanski@lip6.fr

Abstract. This paper presents the model of Layered Memory Automata (LaMA) to deal with languages involving infinite alphabets, with practical applications in the analysis of datastreams, or modeling complex resource usages in concurrent systems. The LaMA can be seen as an extension of the Finite Memory Automata (FMA) with memory layers and the capacity of dealing with an unbounded amount of memory. Despite the increased expressiveness, the LaMA preserve most of the "good" properties of the FMA, in particular the closure properties for the so-called quasi-regular constructions. Moreover, the layering of the memory enables particularly economical constructions, which is an important focus of our study. The capacity of dealing with an unbounded amount of memory brings the LaMA closer to more powerful automata models such as the history register automata (HRA), thus occupying an interesting position at the crossroad between the operational and the more abstract points of view over data-languages.

Keywords: data languages · memory automata · register automata · unbounded memory · quasi-regular languages

1 Introduction

Automata on *datawords*, involving infinite alphabets, represent an influential foundation for the analysis of datastreams [13]. Resource analysis frameworks for concurrent systems have also been investigated based on similar automata-theoretic foundations, e.g. in [1] or our own previous work [7]. Quoting [10]:

> *Actions of concurrent processes, when concurrency and communication are restricted to very simple patterns, are another possible interpretation of infinite alphabets.*

The classification of automata models for datawords can be roughly decomposed in two major families. The first family, pioneered by the *finite memory*

automata (FMA) of [10] (colloquially known as *register automata*), adopts a mostly *operational* point of view similar to the classic *finite state automata* (FA). With FMA, letters (ranging over an infinite alphabet) can be temporarily or permanently stored in a *finite* amount of dedicated memory cells (or *registers*). They can then be compared with letters read at a later time during the recognition process. These models characterize an important notion of *freshness*: the property of a recorded letter to be unique among the ones already stored. In the FMA, this is obtained thanks to an *injectivity* constraint: the fact that the registers must hold distinct letters at any given time. The languages recognized by FMA are called *quasi-regular*, emphasizing their "classical" roots. In particular, they enjoy important closure properties, especially for the regular operators with the notable exception of *complementation*. Moreover, several important decision problems (e.g. emptiness checking) for FA remain decidable in FMA and related models. At the other end of the spectrum, the family related to *data automata* (DA) [5] adopts a more *abstract* point of view. They adopt principles, such as *guessing* that are very high level in comparison. Unsurprisingly, the decision problems are much harder for these models.

An important distinction can be made between these two *operational* vs. *abstract* families regarding the nature of the memory store. In the FMA and related models, the memory is finitely *bounded*. The automata cannot store more letters, in a given configuration, than the number of available registers. As a consequence, it is impossible expressing a language, which needs an unbounded number of different letters such as the language of words where each letter occurs at most once. This is a particularly strong constraint that one would like to lift in order to take more advantage of the infinite alphabets.

In this paper, we introduce an extension of the FMA, namely the model of *Layered Memory Automata* (LaMA), with both practical and theoretical benefits. Essentially, the intent is to establish a link between the *abstract* and *operational* families of automata. On the one side, the LaMA possess a strong operational nature in that they are a (conservative) extension of the FMA with the extra ability to handle an unbounded amount of memory. LaMA are nondeteministic finite state automata that have a finite number of variables, each of them able to store a finite set of letters. Upon reading a letter, a transition can test if the letter is already stored in a variable, can store the letter in a variable, or can reset a variable to emptyset. Like FMA, the variables of LaMA are under an injectivity constraint, which means that two variables cannot stores the same letter. This constraint is partially relaxed with the introduction of a finite set of *memory layers*. Variables are grouped into layers, and the injectivity constraint is only required between variables of a same layer.

Other works designed models extending FMA to manage an unbounded number of different letters. The FRA (*Fresh-Register Automata*) introduce the history, a memory cell able to store a set of letters from the infinite alphabet not restricted by injectivity constraint. It is used as well as the registers of FMA to express the notion of globally fresh letter, a letter never stored in a register before. However, this extension only slightly increases the expressivity of FMA as the set of languages recognized by FRA is not close to concatenation

and Kleene star. The HRA (*History-Register Automata*) uses multiple of this histories instead of registers to store letter of the infinite alphabet. Without injectivity constraints, HRA uses a similar transition as M-FMA (*M-Automata* from [10]), the transition guard is satisfied when the input letter is stored in the exact set of histories annotating the transition. HRA transitions can clear a history and transfer the inputted letter among histories. Our model LaMA is similar to HRA, but without the possibility of performing *transfers* among its variables (histories) and preserving the injectivity constraint from FMA.

As a primary contribution, we argue that the LaMA provide a kind of a *sweet spot* between the "good" operational properties of the FMA, and (at least some of) the expressiveness of higher-level models with unbounded memory capabilities. A second contribution we defend in this paper is the *economical* nature of the proposed model regarding the (quasi-)regular constructions. The regular constructions proposed for FMA in [10,15] or [8] all yield automata of exponential sizes. Despite the fact that the LaMA strictly subsume the FMA (with unbounded memory), the constructions we propose for concatenation, disjunction and conjunction[1] of (the language recognized by) LaMA remain polynomial. Despite its simplicity, the idea of the memory layers plays here a crucial role. Unfortunately, the construction for the *Kleene star* remains exponential for LaMA. In [2] we introduce a variant of the LaMA with *transfer* capabilities that allows to obtain a polynomial construction. However, this variant only preserves the membership problem, and most other "good" properties are lost. Because of this and of space constraints, this variant will not be presented in detail in this paper.

The outline of the presentation is as follows. The LaMA model is presented in Sect. 2 with a discussion of related work, and its main closure properties are discussed in Sect. 3. Important language inclusion links between LaMA and other automata models are presented in Sect. 4. Finally, in Sect. 5 we discuss the important aspect of the sizes of the regular constructions in LaMA and related models.

2 Layered Memory Automata

We present in this section the model of layered memory automata (LaMA), an extension and improvement of the ν-automata presented in previous works [3]. The principle is to recognize datawords based on a countably infinite alphabet of letters, that we denote by $\mathcal{U}$. During the recognition process, the LaMA use *variables* to identify memory cells that can store sets of letters read as input.

The main specificity of this memory model is its structuring in *layers*. A memory context M corresponds to a memory divided in distinct layers. Each layer, identified by a natural number, can store a finite set of letters (over the infinite alphabet). Thus, for example, we can say that, in M, a variable X contains (is associated with) the finite set $E \subseteq \mathcal{U}$ at layer l, which will be denoted by $M(X^l) = E$. By a slight abuse of terminology, we will often write

[1] Without complementation, the conjunction operator becomes primitive in FMA and related models.

"the variable X^l" to in fact designate "the variable X at layer l". The formal definition is given below.

Definition 1 (Memory context). *Given a finite set of variables V, a finite set of layers L and an infinite alphabet $\mathcal{U}$, we define a memory context M as an association function whose signature is as follows: $M : V \times L \to 2^{\mathcal{U}}$ where $M(X^l) \subset \mathcal{U}$ is the finite set of letters* associated *with variable X^l.*

The most important feature of memory contexts is the following *injectivity* constraint.

Definition 2 (Injectivity of layers). *Let a memory context M be defined on the finite sets of variables V and layers L, the injectivity constraint is:*

$$\forall (X, Y) \in V \times V, \forall l \in L, X \neq Y \implies M(X^l) \cap M(Y^l) = \emptyset$$

In more informal terms: it is forbidden for a given letter to be stored in the memory corresponding to distinct variables at the same layer. If compared to FMA, we can say that each layer resembles the memory context of a FMA, but that distinct layers remain independent. The second, and fundamental difference with FMA is that the memory of LaMA is *unbounded*: each memory cell X^l can store an arbitrary number of letters.

Thanks to the injectivity constraint, we can define the notion of a *fresh letter at layer l*, i.e., a letter that is associated with no variable of the layer l. This subsumes the usual notion of a fresh letter, i.e., a letter being fresh at all layers.

We now explain the composition of a LaMA as *state-transition machines*.

Definition 3 (Layered Memory Automata). *Layered Memory Automata are defined with respect to an infinite alphabet $\mathcal{U}$ and are represented as tuples of the form $A = (Q, q_0, F, \Delta, V, L, M_0)$ where:*

- *Q is a finite set of states*[2]*,*
- *$q_0 \in Q$ and $F \subseteq Q$ are respectively the initial state and the set of accepting states,*
- *Δ is a finite set of transitions*[3]*, described below in Definition 4 and 5,*
- *V and L are respectively the finite set of variables and the finite set of layers, and*
- *$M_0 : V \times L \mapsto 2^{\mathcal{U}}$ is the initial memory context.*

The initial memory context M_0 indicates the letters initially associated with each variable. This makes it possible to define a finite alphabet of *constants*,

[2] The term *state* is rather connoted, being also used in e.g. "state-space" to designate "runtime" artifacts. We will use the term *configuration* to designate the notion of a "running state".

[3] For transitions, we will make the distinction between the transition itself, and its *firing*, i.e., the fact of effecting the transition on a previous configuration, to construct a next configuration, at "runtime".

similarly to FMA, with thus the possibility to simulate classical FA (a feature we will not take advantage of in this paper).

The set Δ of transitions encompasses two kinds of transitions: (1) the *observable transitions* that are fired when a letter is read in input, and which *consume* the letter, and (2) the *ε-transitions*, which are non-observable and thus can be fired at any time (without consuming the input).

Definition 4 (Observable transition). *The observable transitions are tuples of the form:* $\delta = (q, \nu, \alpha, \overline{\nu}, q') \in \Delta$ *where:*

- $q, q' \in Q$ *are the source and destination states of the transition,*
- $\nu \subseteq 2^{V \times L}$ *is the set of variables modifiable by the transition,*
- $\alpha : L \to V \cup \{\sharp\}$ *indicates for each layer the variable consulted by the transition,*
- $\overline{\nu} \subseteq 2^{V \times L}$ *is the set of variables which are reset by the transition.*

Input letters can only be consumed when firing such observable transitions. The precise definition of a transition firing is given below (cf. Definitions 7 and 8) but we summarize the informal intent now. The α function indicates the variables consulted by the transition, with the constraint that at most one variable can be consulted for each layer. The special symbol $\sharp$ is used to indicate that no variable is to be consulted for this layer when firing the transition. When reading a letter u, the transition may be fired if, for each variable X^l such that $\alpha(l) = X$:

- either X^l is not modifiable ($X^l \notin \nu$) and u is already associated with X^l;
- or, if X^l is modifiable ($X^l \in \nu$), then u is fresh for layer l (i.e., associated with no variable of layer l).

Because of the injectivity constraint, only a *fresh* letter can be associated with a variable X^l. That is, upon reading, the letter must not be associated with any variable in layer l, not even X^l.

Remark 1 (Universal transition). If no variable is consulted by a transition (i.e., $\forall l \in L, \alpha(l) = \sharp$), then the transition can be fired when reading any letter.

The set $\overline{\nu}$ is the set of variables that must be reset by the transition. No letter is associated with the variables of $\overline{\nu}$ in the configuration reached by the transition.

Definition 5 (ε-transition). *The non-observable ε-transitions are tuples of the form* $\delta_\varepsilon = (q, \overline{\nu}, q') \in \Delta$ *where:*

- $q, q' \in Q$ *are the source and destination states of* δ_ε*,*
- $\overline{\nu} \subseteq 2^{V \times L}$ *is the set of variables reset by the transition.*

We now turn to the dynamics of the model, describing the behavior of LaMA as language *recognizers*. We begin with the definition of a configuration, i.e., a "running state".

Definition 6 (configuration). *A configuration of a LaMA is a pair (q, M) of a state q and a memory context M. Given an automaton $A = (Q, q_0, F, \Delta, V, L, M_0)$, the initial configuration is (q_0, M_0) and an accepting configuration is a pair (q_f, M), for a reachable memory context M and an accepting, final state $q_f \in F$.*

A dataword belongs to the language of a LaMA if there is a (finite) sequence of firings of transitions going from the initial configuration to an accepting one. For observable transitions the question is the following: given a source configuration (q, M) and an input letter $u \in \mathcal{U}$, is there an observable transition $\delta \in \Delta$ which is *enabled* such that, as an *effect*, we can construct a destination configuration (q', M')? In such a case, the actual firing is denoted by $(q, M) \xrightarrow[\delta]{u} (q', M')$ (for observable transitions), or alternatively $(q, M) \xrightarrow[\delta]{\varepsilon} (q', M')$ (for ε-transitions).

Observable transitions, to be fired, must be *enabled*, under the following conditions.

Definition 7 (Enabling of an observable-transition). *For a configuration (q, M) and an input letter $u \in \mathcal{U}$, an observable transition $(q, \nu, \alpha, \overline{\nu}, q') \in \Delta$ is enabled if and only if for each layer $l \in L$ and variable $X \in V$ such that $\alpha(l) = X \in V$:*

- *if X^l is modifiable ($X^l \in \nu$) then no variable must be already associated with u in layer l, i.e., $\nexists Y \in V, u \in M(Y^l)$;*
- *otherwise ($X^l \notin \nu$), u must be associated with X in layer l, i.e., $u \in M(X^l)$.*

Informally, the role of the enabling conditions is: (1) to preserve the injectivity of each layer, and (2) to check the capability of consuming the input and store it in the required memory cells. The ε-transitions are enabled independently from the inputted letter. Once a transition is enabled, it can be non-deterministically *fired*, which produces as an *effect* a resulting configuration, as explained by the following definition.

Definition 8 (Effect of a transition firing). *For a source configuration (q, M), an input letter $u \in \mathcal{U}$ and an enabled transition $\delta \in \Delta$, the firing $(q, M) \xrightarrow[\delta]{u} (q', M')$ produces the configuration (q', M') constructed as follows:*

- *if $\delta = (q, \overline{\nu}, q')$, then M' consists of M where the variables in $\overline{\nu}$ are reset, $M' = M[\overline{\nu} \to \emptyset]$,*
- *if $\delta = (q, \nu, \alpha, \overline{\nu}, q')$, then in M' the modifiable variables are associated with u and the variables of $\overline{\nu}$ are reset,*

i.e., for each X^l, $M'(X^l) = \begin{cases} \emptyset & \text{if } X^l \in \overline{\nu} \quad (1) \\ M(X^l) \cup \{u\} & \text{if } \alpha(l) = X \wedge X^l \in \nu \quad (2) \\ M(X^l) & \text{if } X^l \notin \nu \vee \alpha(l) \neq X \quad (3) \end{cases}$

The memory context M' produced by a transition firing is the result of a combination of three different cases of effects, denoted by (1) - (3) in the definition above. Case (1) corresponds to the reset of the memory cell X^l, which

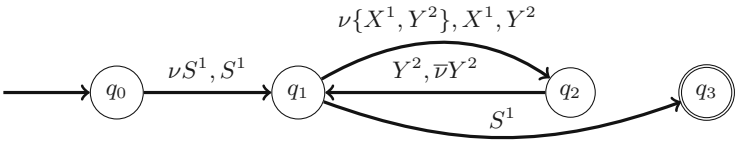

Fig. 1. A layered memory automaton recognizing words of the form $abbccdd \dots a$

is thus emptied. Case (2) corresponds to the actual consumption of the letter u, which is placed in all the required memory cells. Finally Case (3) aims at preserving the unchanged parts of the memory.

The language recognized by a LaMA is now naturally defined by sequences of firings from the initial configuration to accepting ones. To simplify the definition, we first introduce the notion of a *weak firing* that encompasses the firing of a single observable transition, surrounded by (possibly empty) sequences of ε-transitions.

We denote by $(q, M) \overset{u}{\underset{\delta}{\Rightarrow}} (q'', M'')$ a weak transition firing, corresponding to any firing sequence of the form: $(q, M) \overset{\varepsilon}{\underset{\gamma}{\rightarrow}}{}^* \cdots \overset{u}{\underset{\delta}{\rightarrow}} (q', M') \overset{\varepsilon}{\underset{\eta}{\rightarrow}}{}^* (q'', M'')$.

Definition 9 (Language of a LaMA). *Let A be a LaMA and $\mathbb{L}(A)$ the language it recognizes. A word $w = u_1\ u_2\ \dots\ u_n \in \mathcal{U}^*$ belongs to $\mathbb{L}(A)$ iff there exists a sequence of weak transition firings: $(q_0, M_0) \overset{u_1}{\underset{\delta_1}{\Rightarrow}} (q_1, M_1) \overset{u_2}{\underset{\delta_2}{\Rightarrow}} \cdots \overset{u_n}{\underset{\delta_n}{\Rightarrow}} (q_n, M_n)$ such that $q_n \in F$.*

We depict in Fig. 1 an example of a LaMA with 4 states and 5 transitions. The memory structure of the automaton involves the variables X, Y, S with two distinct layers 1 and 2.

For the sake of readability, we use a slightly simplified notation for the transition label. A transition labeled $\nu\{X^1, \dots\} Y^1 Z^2 \dots \overline{\nu}\{U^1, \dots\}$ in a diagram, from a state labeled q to a state labeled q', corresponds more formally to a transition $\delta = (q, \nu, \alpha, \overline{\nu}, q')$ such that $\nu = \{X^1, \dots\}$, $\alpha = \{1 \mapsto Y, 2 \mapsto Z, \dots\}$ and $\overline{\nu} = \{U^1, \dots\}$. Also, we omit the brackets for singleton sets, and we also omit the empty sets and the epsilons. For example, in the diagram of Fig. 1, for the transition labeled $Y^2, \overline{\nu} Y^2$ we in fact mean $\nu = \emptyset$, $\alpha = \{1 \mapsto \sharp, 2 \mapsto Y\}$ and $\overline{\nu} = \{Y^2\}$. Thus, the transition labeled S^1 actually means $\nu = \overline{\nu} = \emptyset$ and $\alpha = \{1 \mapsto S, 2 \mapsto \sharp\}$.

Now that the simplified notation is in place, we can explain the behavior of the depicted automaton. The language it recognizes is the following one:

$$\{s x_0 x_0 x_2 x_2 \dots x_n x_n s \mid \forall i, j \in \mathbb{N}, s, x_i \in \mathcal{U}, i \neq j \implies x_i \neq x_j\}$$

This is an example inspired from [3] where we study the pattern recognition in dynamic graphs, with datawords representing sequences of edges established dynamically (so-called *link streams*). In this representation, the automaton characterizes a Hamiltonian circuit as a pattern.

We assume the initial memory context to be empty, i.e., no letter is initially associated with the variables. The role of variable S is to identify and memorize the first letter of the word (here a node of a graph) through the transition from q_0 to q_1. This is stored in S at layer 1, denoted S^1. The cycling transitions between q_1 and q_2 allow to read intermediate letters of the word. The variable X^1 memorizes these intermediate letters when ensuring that letters in even positions are all different from each other. Since X^1 belongs to the same layer as S^1, the injectivity constraint ensures that all letters are different from the first one. Then, the variable Y^2 ensures that the letters in odd positions are identical to the ones which immediately precede them. In the transition going from q_1 to q_2 the letter in even position is associated with Y^2. The only letter enabling the transition from q_2 to q_1 is the one previously associated with Y^2. Then, Y^2 is reset in order to track the next letter, and not confuse it with the one previously stored. Eventually, the last letter is read, which has to be in even position and to be the same letter as the one stored in S^1 to enable the transition from q_1 to the accepting state q_3.

3 Regular Constructions and Closure Properties

One of the most important properties of FMA, beyond their extended expressiveness, is the fact that they preserve most of the "good" properties of FA, especially closure properties for all the *regular constructions*, except for complement. This aspect is emphasized by the authors of [10] by defining the class of languages recognized by FMA as *quasi-regular*.

The LaMA we introduce in this paper correspond to a strict extension of the FMA (and in fact an extension of both the FRA and the GRA, as discussed in Sect. 4). But most importantly, we aim with the LaMA to an extension that is as *conservative* as possible, wrt. the "good" properties of FMA. In particular, the LaMA ensure the same closure properties as the FMA wrt. the regular constructions. In fact, most regular constructions are greatly facilitated by the availability of layers that allow to compose memory contexts without interference (e.g. composing two LaMA for concatenation). With the notable exception of the Kleene star, the proof schemes thus resemble the ones of FA. As such, we will only present proof sketches, the details being available in [2]. Note, also, that Sect. 5 discusses quantitative aspects related to these constructions.

Theorem 1 (Closure properties of basic operators). *Let the two LaMA* $A_1 = (Q_1, q_1, F_1, \Delta_1, V_1, L_1, M_1)$ *and* $A_2 = (Q_2, q_2, F_2, \Delta_2, V_2, L_2, M_2)$*, such that* $L_1 \cap L_2 = \emptyset$*, then:*

- *(Concatenation) there is a LaMA* $A_{1\cdot 2}$ *such that* $\mathbb{L}(A_{1\cdot 2}) = \mathbb{L}(A_1) \cdot \mathbb{L}(A_2)$.
- *(Union) there is a LaMA* $A_{1\cup 2}$ *such that* $\mathbb{L}(A_{1\cup 2}) = \mathbb{L}(A_1) \cup \mathbb{L}(A_2)$.
- *(Intersection) there is a LaMA* $A_{1\cap 2}$ *such that* $\mathbb{L}(A_{1\cap 2}) = \mathbb{L}(A_1) \cap \mathbb{L}(A_2)$.

Proof (Proof sketches). The assumption $L_1 \cap L_2 = \emptyset$ is without loss of generality because a trivial fact is that the injective renaming of the set of layers of a LaMA

(with *fresh* layer identities) does not change the language it recognizes. Now, we consider the basic operators in turn.

Concatenation. It is possible to construct automaton $A_{1\cdot 2}$ following the classical construction of finite state automata, which consists in adding ε-transitions allowing to access the initial state of A_2 from each accepting state of A_1. As the layers of A_1 and A_2 are disjoint, their memories are actually put side by side and the variables of both automata do not interact together. Thus, we ensure that there is no side effect of A_1 on A_2 and the initial values of the variables of A_2 do not change when firing transitions in A_1.

Union. Similarly as above, the classical construction of FA applies here, which consists in adding a new initial state connected to the former initial states of A_1 and A_2 with ε-transitions (without reset). As for concatenation, the variables of both automata do not interact thus the initial context of the A_1 has no impact on the recognized language of A_2.

Intersection. As the memories of both automata are disjoint, it is possible to use the classical construction of a synchronized product of automata. The synchronization of two non-ε-transitions consists forming a transition labeled with the union of the sets ν, α and $\overline{\nu}$ of both transitions. Formally, the synchronization of observable transitions $(q_1, \nu_1, \alpha_1, \overline{\nu}_1, q_1') \in \Delta_1$ with $(q_2, \nu_2, \alpha_2, \overline{\nu}_2, q_2') \in \Delta_2$ is the transition : $((q_1, q_2), \nu_1 \cup \nu_2, \alpha_{1\cap 2}, \overline{\nu}_1 \cup \overline{\nu}_2, (q_1', q_2'))$ where $\forall i \in \{1, 2\}, l \in L_i, \alpha_{1\cap 2}(l) = \alpha_i(l)$. This construction is illustrated in Appendix A.

The case of iteration, or Kleene star, is a little bit less straightforward because during an iteration the memory context of the automaton may change, however such effect should be "canceled" for further iterations. Indeed, each (regular) iteration has to recognize exactly the same language, and not a language changed due to memory effects of previous iterations.

Theorem 2 (Closure property of Kleene star). *Let $A = (Q, q, F, \Delta, V, L, M)$ be a LaMA, then there is a LaMA A^* such that $\mathbb{L}(A^*) = \mathbb{L}(A)^*$.*

Proof (Proof sketch). The proposed construction is based on the classical one for FA which requires adding "ε-loops" from accepting states to the initial, thus allowing to iterate on the content of automaton A. As with all kinds of register automata, one difficulty with LaMA is that the language recognizable from a configuration depends on its memory context. And the latter can change at each iteration. In a way similar to what is done in the case of M-automata [10], the required "cancelling" of memory effects is realized thanks to a mechanism simulating a reset of the memory context to its initial value M_0. To do so, the principle is to duplicate the set of variables of layers in L on a set of "shadow" layers L_s. The variables of L are used to memorize the initial values of M_0 while the variables of L_s are used to store the fresh values recognized during the iterations. This way, in order to retrieve the initial values of the memory context, it is enough to remove at the end of each iteration all the letters stored in the variables of L_s.

If a transition is enabled in A when the letter read is associated with variable X^l, then this transition has to be duplicated in A^* such that it is possible to

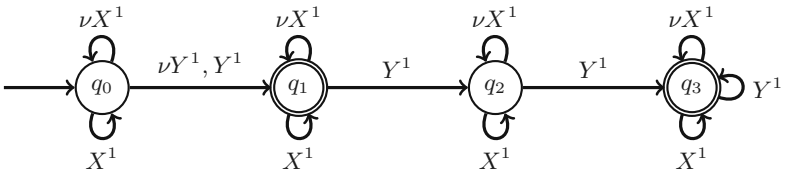

Fig. 2. LaMA accepting the language $\mathbb{L}_{\neq 2}$.

access either X^l (the initial values) or X^{l_s} (the possibly updated ones), with l_s the "shadow" layer corresponding to l. Moreover, if a transition in A has a guard referencing several variables, it is necessary to duplicate this transition in A^*. For example, a transition accessing the variables X^l, Y^k will be duplicated 4 times, once for each pair of : $(X^l, Y^k), (X^{l_s}, Y^k), (X^l, Y^{k_s}), (X^{l_s}, Y^{k_s})$. This duplication is required, in the absence of e.g. a transfer mechanism (cf. Sect. 5), because it is not effective to consult the variables in the layers of L and L_s simultaneously. Indeed, their sets of values are disjoint (e.g. initially the layer L_s is empty). In consequence, this construction leads to an exponential growth in terms of the number of transitions of the resulting automaton A^*. Moreover, it is also necessary to know which variables have been reset during each iteration, which is realized by duplicating states, implying also an exponential growth in terms of constructed states. These exponential growth phenomena are discussed further in Sect. 5.

The infinite nature of the alphabet manipulated by all the classes of memory automata (at least all the classes discussed in this paper) is in contradiction with the principle of complementation and determinism. Thus, unsurprisingly the following negative result also applies to LaMA.

Proposition 1 (Complement). *The set of languages recognized by LaMA is not closed under complement.*

Proof. The LaMA represented in Fig. 2 recognizes the language $\mathbb{L}_{\neq 2}$ of words containing at least one letter not appearing twice in all words. It does so by non-deterministicaly selecting a letter when it occurs for the first time, associating it to variable Y^1 and accepting the word only if this letter does not occur in the word exactly twice. The variable X^1 is used to store all the other letters and to never forget them, which ensures that the selection of a letter may only happen at its first occurrence.

The complement of $\mathbb{L}_{\neq 2}$ is the language $\mathbb{L}_{=2}$ containing *only* words with all their letters occurring exactly twice. In order to encode $\mathbb{L}_{=2}$, it is necessary to enumerate the occurrences of all the letters of words recognized by this language. An automaton recognizing this language would have to count an arbitrary number of occurrences of distinct letters. With a finite number of variables and states, such a construction is not possible with LaMA.

A deterministic LaMA is an automaton such that for all configurations, when reading any letter of $\mathcal{U}$, at most one transition can be fired. This restriction

implies that when reading a globally fresh letter there is at each step only one way to identify it (associate it with a variable).

Proposition 2 (Determinism). *The set of languages recognized by deterministic LaMA is strictly included in the set of languages recognized by non-deterministic LaMA.*

Proof. The language $\mathbb{L}_{\neq 2}$ recognized by the non-deterministic LaMA from Fig. 2 cannot be recognized by a deterministic LaMA. To recognize this language, the automaton would have to "find" a letter that will not occur exactly twice. However, the words from this language are finite but may contain an arbitrary amount of different letters. Thus, it is not possible to track the number of occurrences of each of them with a finite amount of variables and layers.

4 A Classification of LaMA (Related Work)

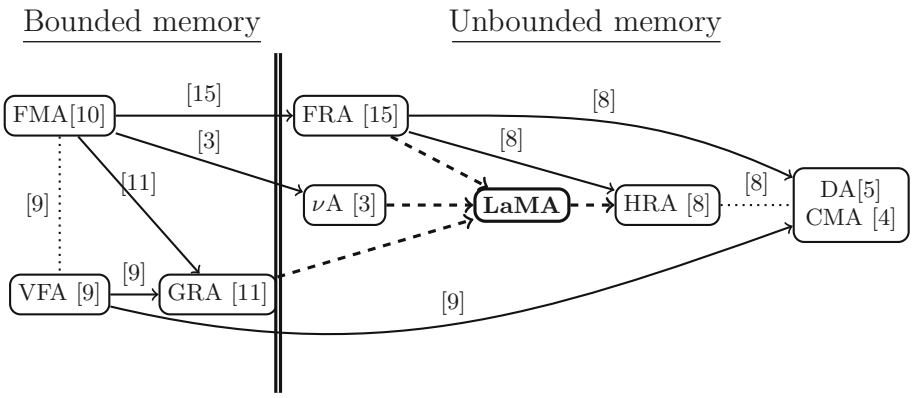

Fig. 3. A classification of automata over datawords, based on [12]. The arrows represent (strict) language inclusions, the dashed arrows are presented in Sect. 4, and dotted lines denote language incomparability.

Figure 3 represents most of the automata models we investigated while developing our proposition. The arrows on the figure are (strict) language inclusions. In this discussion, we denote by $A \sqsubset B$ the fact that the languages recognized by automata of model A strictly includes those of model B. For example, we know from [15] that the FRA (*fresh register automata*) can simulate the FMA, and thus FMA $\sqsubset$ FRA. The models related by dotted lines are knowingly *incomparable*. In this section we discuss the positioning of the LaMA in the family of data language recognizers. More precisely, we present the language inclusions depicted by dashed arrows on the figure. Since we cannot describe the related automata models with enough details in this paper, the discussion remains mostly informal, with the complete proof available in [2].

The LaMA were designed, broadly speaking, as a variant of FMA with unbounded memory capabilities. It is thus expected that LaMA are able to simulate FMA. Since the LaMA with one layer correspond exactly to ν-automata, we can reuse the result of [6] to show that LaMA are able to simulate the FMA.

Proposition 3. *FMA* $\sqsubset$ *LaMA*

However, in technical terms, it is interesting to compare the LaMA with other models proposed as extensions or variations of the FMA. FRA (*fresh register automata*) is a conservative extension of FMA capable of dealing with (a restricted kind of) unbounded memory. It is possible to simulate a FRA with a 2-layer LaMA, and thus to simulate a FMA by transitivity.

Proposition 4. *FRA* $\sqsubset$ *LaMA*

Proof (Proof sketch). The FRA model is based on a memory composed of a set of registers capable to memorize a unique letter, and constrained by injectivity. The model is thus quite similar to the FMA, however with a little but important "twist". An FRA also provides a "special" variable capable of recording all the letters read since the beginning of the recognition. The transitions of FRA are found in three categories that can be enabled in three different ways:

1. when reading a letter already present in some register;
2. when reading a letter which is locally fresh, i.e., not present in any register currently;
3. when reading a letter which is globally fresh, i.e., not encountered since the beginning of the recognition.

It is not difficult to provide these mechanisms with a LaMA. The required memory context contains two layers. Each variable of the first layer corresponds to a register of the simulated FRA. The second layer, independent, only concerns the "special" variable to simulate its content. Since the memory cells of LaMA are not bounded, we can say that all the variables of LaMA are "special", in the FRA understanding of the term. Put in other terms, the FRA can be seen as a special cases of LaMA with a FMA-like layer of bounded memory, and a unique variable of unbounded memory in a second layer.

The LaMA are also strictly more expressive than the FRA. One may observe, indeed, that FRA are not closed under concatenation. For example, the language $\mathbb{L}_{\neq}$ of words composed of all-distinct letters, may be recognized by both FRA or LaMA. But the language $\mathbb{L}_{\neq} \cdot \mathbb{L}_{\neq}$ is only recognized by LaMA.

The GRA (*guessing register automata*) model is an interesting variant of FMA using a non-deterministic assignation (*guessing*) principle. By proving, below, that LaMA are able to simulate GRA it emphasizes the fact that the LaMA are also capable of simulating its *guessing* principle, and not only the operational principles of the FMA. This establishes an interesting connection with the "logical" family that also rely on guessing features (note the inclusion link between VFA and DA in Fig. 3).

Proposition 5. *GRA* $\sqsubset$ *LaMA*

Proof. The GRA model is a variant of FMA with a modified variable assignment method. The memory of a GRA is composed of a finite set of registers, each containing at most one letter, together with an injectivity constraint. The transitions of GRA are found in two categories:

- the observable ones are annotated by the register containing the letter that has to be consumed to fire the transition;
- the ε-transitions are annotated with a register which is reassigned to a non-deterministically *guessed* letter.

The assigned letter will be decided when firing the next observable transition annotated with this register. However, if other registers are reassigned in the meantime, they cannot be assigned the same letter due to the injectivity constraint.

Given a GRA, it is possible to construct a LaMA which recognizes the same language. After the reassignment of a register r, an arbitrary letter of the infinite alphabet is non-deterministically assigned to it. To find out which letter was assigned to r, it is necessary to memorize all letters currently assigned to the other registers and those that will be assigned to them until an observable transition labeled with r is fired. This transition will be enabled by any letter not recorded since the reassignation.

Hence, for each register of a GRA, the simulating LaMA will use as many variables as necessary to memorize all the values stored by every other registers between its reassignment and the transition that will determine the guessed value. This way, when an observable transition allowing to determine the value of the input letter is enabled, the injectivity constraint ensures that the letter is different from those already associated with other registers. The actual construction is in consequence quite intricate, and we delegate to [2] for the formal details.

The inclusion is strict since it is known (from [11]) that there is no GRA that can recognize the language of words of any length with all letters occurring only once.

Perhaps the most interesting inclusion link is the one connecting the LaMA to the more expressive HRA (*history-register Automata*).

Proposition 6. *LaMA* $\sqsubset$ *HRA*

Proof (Proof sketch). The HRA memory is constituted of variables associated with histories that can store an unbounded amount of letters. This is very much like the ν-automata and thus the LaMA with a single memory layer. However, a very important difference is that the HRA histories are not restricted by an injectivity constraint. There are thus quite similar to the M-automata of [10], but with unbounded memory. The observable transitions are annotated with two sets of histories: R (read) and W (write). A transition is enabled when the input letter is exactly associated with all histories of R. After the firing, in the

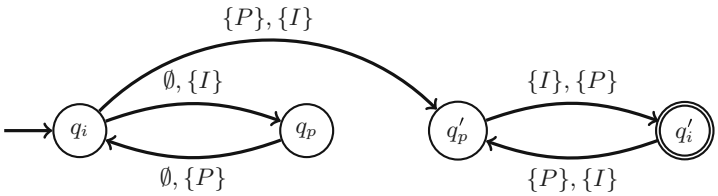

Fig. 4. HRA recognizing a language which is not recognized by a LaMA

resulting configuration, the letter is associated exactly to all histories of W. Thus, the letter can be transferred among the histories, or erased from them, in the resulting configuration. The ε-transitions are annotated with a set of histories C containing histories cleared (reset) in the resulting configuration.

It is possible to simulate a LaMA with a HRA by encoding the memory layers and the injectivity constraint. The simulating HRA has the same set of states, as well as a history for each variable of the original LaMA. Since the observable transitions of HRA cannot reset variables, they are split in two parts: (1) a transition for the enabling and firing, and (2) a transition for the reset. To simulate the enabling and firing of a LaMA transition, multiple observable transitions are needed in the HRA:

- for each variable X^l consulted in the LaMA transition, $\alpha(l) = X, X^l \notin \nu$, the matching history is part of both R and W;
- for each variable X^l modified in the LaMA transition, $\alpha(l) = X, X^l \in \nu$, the matching history is only part of W.

As R needs to encompass the histories containing the input letter in order to be enabled, when no variable is consulted for some layer, $\exists l, \alpha(l) = \sharp$, then the transition needs to be duplicated in the HRA to search if the value is present in one of the histories of this layer. If multiple layers are not consulted, then the transition is duplicated to search the letter in each combination of histories for those layers. To enforce the layer injectivity constraint, the construction is designed so that the transitions are never annotated by histories that simulate variables of the same layer. This way, during the recognition, it is not possible to reach a configuration in which the histories corresponding to the same layer contain a common letter.

The observable transitions can remove the input letter from the histories it is annotated with, when $R \backslash W \neq \emptyset$. It will thus be possible to delete a particular letter from a history, which is impossible for LaMA. Thus, it is rather easy to come up with a language recognizable by a HRA, and not recoginizable by a LaMA. For example, no LaMA can recognize the language of the HRA in Fig. 4, which is the language of words of the form $w = uv$ where:

- the prefix u is a word whose length is **even** and in which all letters are different;

– the suffix $v = v_1v_2v_3 \dots v_n$ is a word in which each letter v_i satisfies that if i is odd then the occurrence of v_i is in an even position in w, and if i is even then the previous occurrence of v_i is in an odd position in w.

It is known, from [8], that the HRA recognize languages that are incomparable with those of the CMA and DA (class memory automata and data automata). This is due to the capability of resetting histories in HRA, which cannot be simulated by a CMA/DA. We have not studied the problem finely, but, for the same reason, we expect the incomparability of LaMA vs. CMA/DA, although it is for now only a conjecture.

The connections we established with related automata models allow us to give some insight about the complexity (and decidability) of some decision problems concerning LaMA. First, the strict inclusion of FMA induces the undecidability of the same problems as FMA, in particular the *language inclusion* and the *universal language* problems (cf. [14]). The inclusion links discussed previously allow to establish the following:

Fact 1. *The emptiness checking and membership problems for LaMA are both NP-hard.*

Proof. The *emptiness checking* problem consists in detecting if the language of an automaton is empty. The problem is known to be NP-complete in the case of FMA [10,14]. Moreover, the same problem is known to be Ackermann-complete for HRA [8], thus trivially decidable for LaMA. The situation is in fact exactly the same for the *membership problem*: NP-complete for FMA and "at-most" Ackermann-complete for HRA. Indeed, the membership problem can be solved through emptiness, although for some automata model the membership problem can be solved by better, dedicated ways (starting with FA). It is unlikely that this would be the case for LaMA since it is already not the case for FMA (cf. [10,14]).

As a future work, we intend to study more finely the complexity of these two problems for LaMA. It would be interesting to see if the use of unbounded memory *without* a transfer mechanism simplifies the emptiness problem (put in other terms, do we reach the Ackermann bound?).

5 A Quantitative Point of View on Regular Constructions

Expressiveness is not the only important aspect to consider when comparing classes of automata. For example, many "regular"-expression packages (e.g. PCRE[4]) adopt the non-deterministic finite state automata (NFA) rather than the theoretically "more efficient" and equivalent determistic ones (DFA), because of the exponential growth when translating the former to the latter. In the same spirit, the prototype analysis tool we develop[5] requires the construction of an

[4] Perl compatible regular expressions, cf. https://www.pcre.org/.
[5] PaMaTina, cf. https://github.com/clementber/MaTiNA.

Table 1. Translation between FMA and M-FMA

	♯states	♯transitions	♯registers
FMA → M-FMA	$\|Q\| * (\|M\|!)$	$\|\Delta\| * (\|M\|!)$	$\|M\| + 1$
M-FMA → FMA	$\|Q\| * \|M\|^{\|M\|}$	$\|\Delta\| * \|M\|^{\|M\|}$	$\|M\|$

automaton, akin to a (timed variant of the) LaMA, from an extension of regular expressions (cf. [2,3]). In this compilation step, the size of the resulting automaton plays a significant role.

In this section we compare the sizes of the regular constructions for three models of automata: the LaMA, the FMA (taking the constructions proposed in [10]) and the HRA (taking those of [8]). Note that these sizes are not given in the aforementioned papers, and we established them while learning about those constructions. As a consequence, all encountered errors about these computations would be ours, not those of the original authors. We evaluated the sizes of the constructions of the FMA presented in the proof of Theorem 3 of [10]. For the HRA, we evaluated the sizes of the constructions presented in Sect. 3 of [8]. The GRA [11] and FRA [15] constructions are not studied here as they are based on the ones presented for FMA and HRA. For the sake of concision, we only consider the (most intricate) cases of concatenation and Kleene star in this paper (the other constructions being also detailed in [8,10]).

Most importantly, our intent is *not* to say that the construction we propose are "better", in any sense of the word, but instead: (1) to motivate the fact that reasoning about the size of the constructions is important, and (2) trying to find ways to make such construction as *compact* as possible. A positive point of view is that if we find compact constructions for LaMA, then they can also be used almost directly as compact constructions for FMA (by first translating FMA to LaMA, which is both straightforward and economical), and similarly for FRA, GRA and VFA.

To compare the constructions, the sizes we consider are the worst-case estimates of the automata, with respect to:

- the number of states in the automata, denoted by $|Q|$,
- the number of transitions, denoted by $|\Delta|$,
- and the number of memory identifiers, denoted by $|M|$.

What we call memory identifiers here are the registers in the FMA, the histories in the HRA and the variables $X^l \in V \times L$ in the LaMA. This quantification on the identifiers does not take into account the number of letters that may be stored in memory, simply because there is no bound in the case of LaMA and HRA. In the following tables we denote by $|\Sigma|$ the number of letters initially stored in the memory of an automaton and by $|L|$ the number of layers of the LaMA.

In [10], the regular constructions are not established directly for FMA but rather rely on the equivalent model of *M-Automata* (M-FMA). Thus, the FMA are first converted to M-FMA, which in fact already causes an exponential

growth, as described on Table 1. The M-FMA resulting from the translations use approximately the same number of registers. However, the loss of the injectivity constraint in M-FMA causes an explosion in the number of states required to simulate the correct (i.e., injective) use of registers. The duplication of transitions follows from the duplication of states. Perhaps surprisingly the exponential growth is also present when translating back to FMA (which could perhaps be avoided by keeping a little bit more structural information in M-FMA). But as it is, none of the regular constructions proposed for FMA has polynomial size.

Concatenation. Table 2 represents the sizes of the automata constructed for concatenation. The constructions for the three models try to duplicate that of the finite state automata by keeping the structures of the two automata and by adding transitions allowing access to the initial state of the suffix automaton at the end of the prefix automaton path.

Table 2. Sizes of constructions for concatenation $\mathbb{L}(A_1) \cdot \mathbb{L}(A_2)$

	♯states	♯transitions
M-FMA	$\lvert Q_1\rvert + \lvert Q_2\rvert$	$\lvert \Delta_1\rvert * 2^{\lvert M_2\rvert} + \lvert \Delta_2\rvert * 2^{\lvert M_1\rvert}$
HRA	$(\lvert Q_1\rvert + \lvert Q_2\rvert) * 2^{\lvert \Sigma_2\rvert * \lvert M_2\rvert}$	$(\lvert \Delta_1\rvert + \lvert \Delta_2\rvert) * (\lvert \Sigma_2\rvert + 1) * 2^{\lvert \Sigma_2\rvert * \lvert M_2\rvert}$
LaMA	$\lvert Q_1\rvert + \lvert Q_2\rvert$	$2 * \lvert \Delta_1\rvert + \lvert \Delta_2\rvert$

	♯registers
M-FMA	$\lvert M_1\rvert + \lvert M_2\rvert$
HRA	$\mathsf{max}(\lvert M_1\rvert, \lvert M_2\rvert) + \lvert \Sigma_2\rvert$
LaMA	$\lvert M_1\rvert + \lvert M_2\rvert$

In M-FMAs, the constructed automaton uses all the registers of the two concatenated automata, as well as their initial valuations. However, due to the nature of the transitions, similar to that of the HRA, it becomes necessary to duplicate all the transitions for each subset of registers of the other automaton. Thus, this leads to a combinatorial explosion in the number of transitions in the automaton resulting from the construction.

In the HRA, before the construction is carried out, all the letters initially associated with the histories of the suffix automaton are extracted from the two automata. These letters are each associated with a new history. This preserves the initial value of the suffix automaton memory when transiting the prefix one. However, when these values are extracted, it is necessary to add transitions in order to preserve the language of the automaton. Thus, in the resulting automaton, the transitions leading to the initial position of the suffix automaton reset all the histories except those containing the extracted letters.

Kleene Star. Table 3 presents the sizes of constructions for Kleene star. The construction used in the LaMA is inspired from that of M-FMA. Thus, the sizes are of the same order.

Table 3. Sizes of constructions for the Kleene star $\mathbb{L}(A)^*$.

	♯states	♯transitions	♯registers
M-FMA	$\|Q\| * 2^{\|M\|}$	$\|\Delta\| * 2^{2*\|M\|}$	$2 * \|M\|$
HRA	$\|Q\| * 2^{\|\Sigma\|*\|M\|}$	$\|\Delta\| * 2^{\|\Sigma\|*\|M\|} * (\|\Sigma\| + 1)$	$\|M\| + \|\Sigma\|$
LaMA	$\|Q\| * 2^{\|M\|}$	$\|\Delta\| * 2^{2*\|M\|}$	$(2 * \|M\|) + \|L\|$

For the Kleene star, the construction in the HRA consists first of all in extracting all the letters initially stored in the histories and in storing them in new dedicated histories, as in the construction for concatenation. It is again necessary to duplicate the transitions and the states so that the automaton always recognizes the same language. So at the end of each iteration it suffices to reset all the other histories in order to reset the memory to its initial value.

As a summary, the constructions proposed for LaMA are in most cases more compact than the ones proposed for FMA (and M-FMA) and HRA. This is not shown here but the situation is the same for all the regular operators. In fact, all constructions are polynomial for LaMA with the notable exception of the Kleene star. To address this issue, we propose in [2] a variant of LaMA with a transfer mechanism that allows to copy all the letters associated with a variable from one layer to another layer. This allows to "dump" the memory from the layers in L to the layers in L_s in the final transitions of an iteration, enabling an exponential reduction in the number of required states and transitions. However this new mechanism is quite "powerfull", causing a loss of several "good" properties of the model (if only the closure properties). However, it is shown in [2] that this alternative model is conservative wrt. the membership problem, which explains why we use it in practice.

6 Conclusion

In this paper we introduced the model of LaMA, characterized by the layered structure of their memory, and the fact that this memory is not bounded. We mostly discussed the quasi-regular constructions (insisting on quantitative aspects) and language inclusion links with related models. Beyond such (important) theoretical considerations, we find important to emphasize the fact that the LaMA were also designed with practical applications in mind. This is the main reason why we emphasized so much the "compactness" of the quasi-regular constructions, the layered architecture playing a significant role here.

For future works, we intend to study two more aspects of the model. First, we know that the class of *deterministic* LaMA is strictly less expressive than the non-deterministic ones. However, this class is still worth studying given the fact that the membership problem becomes much easier in this case. Second, we would also like to investigate the relationship between subclasses of MSO and language classes recognizable by LaMA, or a restricted version (without reset for example) as it is done for DA wrt. ∃MSO.

A Examples of Regular Constructions (Complement to Sect. 3)

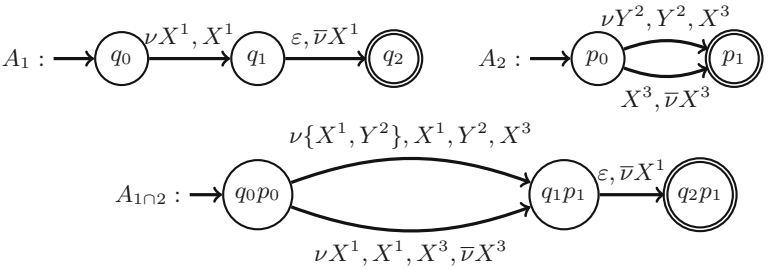

Fig. 5. Intersection construction

In this appendix, we give some more details about the constructions corresponding to the intersection and iteration of LaMa, as a complement to Sect. 3.

Intersection. Figure 5 illustrates the synchronized product of two LaMA on the left, A_1 and A_2, used to produce the LaMA recognizing the intersection of the languages of A_1 and A_2. The resulting LaMA $A_{1\cap 2}$, on the right of the Figure, contains only the states reachable by transitions from the initial states. The construction is thus quite similar to the usual construction for finite automata. One notable difference relates to then handling of observable transitions. In fact, only observable transitions are synchronized together, while non-observable ones are not. The reason is the firing of non-observable transition does not consume letters, and are thus "transparent" wrt. language intersection.

Iteration. Figure 6 illustrates the Kleene star construction, with on top a LaMA A, recognizing language $\mathbb{L}(A)$, and below the LaMA A^* constructed such that $\mathbb{L}(A^*) = \mathbb{L}(A)^*$.

The construction is in principle close to the equivalent construction for finite automata. However, the handling of memory layers requires some care. To illustrate this, the automaton A in the figure uses two layers, 1 and 2. To simulate the reset, two so-called "shadow layers", resp. 3 and 4, are added in A^*. A variable Ω is added on the layers 1 and 2 (even if not used on 2) to check the layer freshness without altering the values initially associated with their other variables.

The states of A are duplicated in A^* where they are annotated with the variables that were reset since the beginning of an iteration. These annotations are used in the construction to create the outgoing transitions. When the variable X^1 is consulted by a transition in A, the matching transitions in A^* are going to consult both X^1 and X^3 if X^1 was never reset before. However, if X^1 was reset, then only X^3 is consulted as the values associated with X^1 should have been deleted. When the variable X^1 is modified by a transition of A, the matching

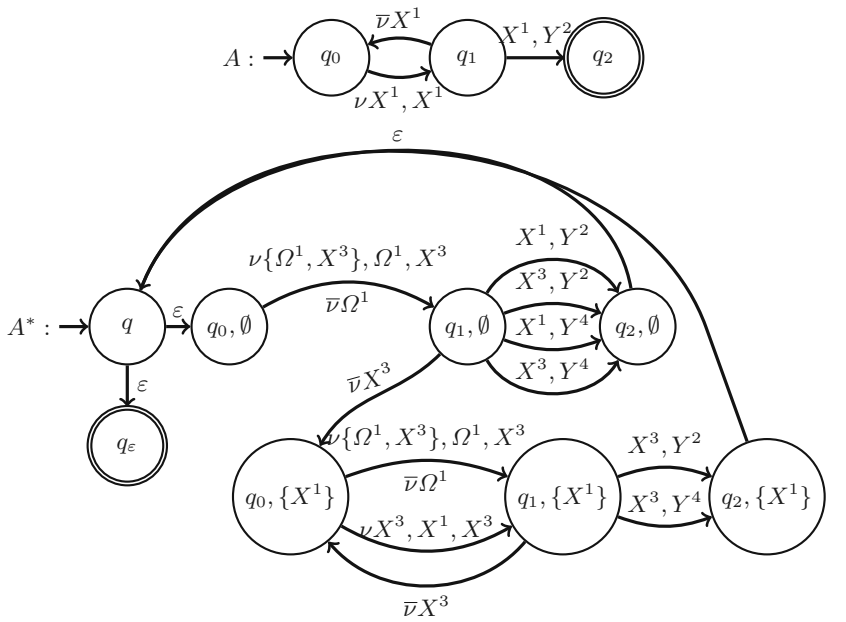

Fig. 6. Kleene star construction

transition in A^* will modify X^3 and it will also check if the value is fresh on layer 1, using Ω^1. However, if X^1 is supposed to have been reset earlier in the iteration, then the transition is duplicated to check if the letter is associated with X^1 instead, as the values it is associated with are supposed to be fresh.

References

1. Bartoletti, M.: Usage automata. In: Degano, P., Viganò, L. (eds.) ARSPA-WITS 2009. LNCS, vol. 5511, pp. 52–69. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03459-6_4
2. Bertrand, C.: Reconnaissance de motifs dynamiques par automates temporisés à mémoire. (Matching of dynamic patterns with timed memory automata). Ph.D. thesis, University of Paris-Saclay, France (2020). https://tel.archives-ouvertes.fr/tel-03172600
3. Bertrand, C., Peschanski, F., Klaudel, H., Latapy, M.: Pattern matching in link streams: timed-automata with finite memory. Sci. Ann. Comput. Sci. **28**(2), 161–198 (2018). http://www.info.uaic.ro/bin/Annals/Article?v=XXVIII2&a=1
4. Björklund, H., Schwentick, T.: On notions of regularity for data languages. Theor. Comput. Sci. **411**(4), 702–715 (2010). https://doi.org/10.1016/j.tcs.2009.10.009. https://www.sciencedirect.com/science/article/pii/S0304397509007518. Fundamentals of Computation Theory
5. Bojanczyk, M., David, C., Muscholl, A., Schwentick, T., Segoufin, L.: Two-variable logic on data words. ACM Trans. Comput. Log. **12**(4), 27:1–27:26 (2011). https://doi.org/10.1145/1970398.1970403

6. Deharbe, A.: Analyse de ressources pour les systèmes concurrents dynamiques. Ph.D. thesis, Université Pierre et Marie Curie, France, September 2016. https://tel.archives-ouvertes.fr/tel-01523979
7. Deharbe, A., Peschanski, F.: The omniscient garbage collector: a resource analysis framework. In: 14th International Conference on Application of Concurrency to System Design, ACSD 2014, Tunis La Marsa, Tunisia, 23–27 June 2014, pp. 102–111. IEEE Computer Society (2014). https://doi.org/10.1109/ACSD.2014.18
8. Grigore, R., Tzevelekos, N.: History-register automata. Log. Methods Comput. Sci. **12**(1) (2016). https://doi.org/10.2168/LMCS-12(1:7)2016
9. Grumberg, O., Kupferman, O., Sheinvald, S.: Variable automata over infinite alphabets. In: Dediu, A.-H., Fernau, H., Martín-Vide, C. (eds.) LATA 2010. LNCS, vol. 6031, pp. 561–572. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13089-2_47
10. Kaminski, M., Francez, N.: Finite-memory automata. Theor. Comput. Sci. **134**(2), 329–363 (1994). https://doi.org/10.1016/0304-3975(94)90242-9
11. Kaminski, M., Zeitlin, D.: Finite-memory automata with non-deterministic reassignment. Int. J. Found. Comput. Sci. **21**(5), 741–760 (2010). https://doi.org/10.1142/S0129054110007532
12. Kara, A.: Logics on data words: expressivity, satisfiability, model checking. Ph.D. thesis, Technical University of Dortmund, Germany (2016). http://hdl.handle.net/2003/35216
13. Libkin, L., Tan, T., Vrgoč, D.: Regular expressions for data words. J. Comput. Syst. Sci. **81**(7), 1278–1297 (2015). https://doi.org/10.1016/j.jcss.2015.03.005
14. Neven, F., Schwentick, T., Vianu, V.: Towards regular languages over infinite alphabets. In: Sgall, J., Pultr, A., Kolman, P. (eds.) MFCS 2001. LNCS, vol. 2136, pp. 560–572. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44683-4_49
15. Tzevelekos, N.: Fresh-register automata. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, 26–28 January 2011, pp. 295–306. ACM (2011). https://doi.org/10.1145/1926385.1926420