Complexity of Membership and Non-Emptiness Problems in Unbounded Memory Automata

Clément Bertrand
Scalian Digital Systems, Valbonne, France

Cinzia Di Giusto[1]
Université Côte d'Azur, CNRS, I3S, France

Hanna Klaudel
IBISC, Univ. Evry, Université Paris-Saclay, France

Damien Regnault
IBISC, Univ. Evry, Université Paris-Saclay, France

Abstract

We study the complexity relationship between three models of unbounded memory automata: *nu*-automata (ν-A), Layered Memory Automata (LaMA)and History-Register Automata (HRA). These are all extensions of finite state automata with unbounded memory over infinite alphabets. We prove that the membership problem is NP-complete for all of them, while they fall into different classes for what concerns non-emptiness. The problem of non-emptiness is known to be Ackermann-complete for HRA, we prove that it is PSPACE-complete for ν-A.

2012 ACM Subject Classification Theory of computation $\rightarrow$ Automata over infinite objects; Theory of computation $\rightarrow$ Problems, reductions and completeness

Keywords and phrases memory automata, ν-automata, LaMA, HRA, complexity, non-emptiness, membership

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.33

Related Version *Full Version*: `https://hal.science/hal-04155339/file/main.pdf` [3]

Acknowledgements We want to thank the anonymous reviewers for the careful reviews and insightful comments.

1 Introduction

We study unbounded memory automata for words over an infinite alphabet, as introduced in [13, 17]. Such automata model essentially dynamic generative behaviours, i.e., programs that generate an unbounded number of distinct resources each with its own unique identifier (e.g. thread creation in Java, XML). For a detailed survey, we refer the reader to [4, 15]. We focus, in particular, on three formalisms, ν-automata (ν-A) [9, 5, 8], Layered Memory Automata (LaMA) [4] and HRA for History-Register Automata [11]. All these models are extensions of finite state automata with memory-storing letters. The memory for HRA is composed of registers (that can store only one letter) and histories (that can store an unbounded number of letters). Whereas the memory for the other two models consists of a finite set of variables. Among the distinctive features of HRA, they can reset registers and histories, and select, remove and transfer individual letters. In ν-A and LaMA, variables must satisfy an additional constraint, referred to as injectivity, meaning that they cannot store shared letters. Moreover, variables can be emptied (reset) but single letters cannot be removed. We know that LaMA are more expressive than ν-A as the former are closed under intersection while it is not the case for the latter ones.

[1] Corresponding author

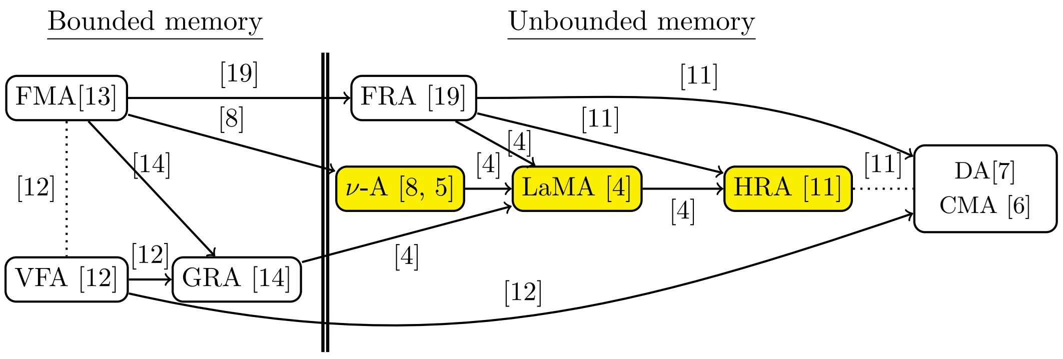


Figure 1 A classification of memory automata from [4]. Arrows represent strict language inclusion, while dotted lines denote language incomparability. The formalisms studied here are in yellow.

We tackle two problems: membership and non-emptiness. From a practical point of view, automata over infinite alphabets can be used to identify patterns in link-stream analysis [5]. In such a scenario, the alphabet is not known in advance (open systems) and runtime verification can help to recognize possible attacks on a network by looking for specific patterns. This problem corresponds to checking whether a pattern (word) belongs to a language (the membership problem). Concerning non-emptiness, this is the "standard" problem to address while considering automata in general.

Fig. 1 depicts the unbounded memory automata known in the literature (to the best of our knowledge). An implementation exists for ν-A and LaMA which includes an implementation of the membership algorithm, but we have not found anything neither for Data Automata (DA) nor Class Memory Automata (CMA). Both DA and CMA are incomparable classes wrt to HRA, hence we chose not to consider them. Fresh-Register automata (FRA), ν-A, LaMA and HRA are instead related from the expressiveness point of view. Given the similarities between those formalisms, the existence of implementations and the lack of complexity results we find it natural to consider these classes of automata.

Application-wise, ν-A, called resource graphs in [9], model the use of unbounded resources in the π-calculus, aiming at minimizing them. Then, runtime verification on link-streams was the initial motivation for the introduction of (timed)ν-A [5]. In subsequent works, LaMA have been introduced to be able to construct the synchronous product, hence being able to express the synchronization of resources. This entails the closure by intersection, which is interesting when one wants to define a language of expressions, an extension of (timed) regular expressions, which was proposed in the PhD thesis of Clément Bertrand [2].

For a precise discussion on the relations among these formalisms see [4], here we just recall the hierarchy: cfr. Fig. 1. Apart from expressiveness, a number of questions concerning complexity remains open. We know that checking whether the language recognized by an HRA is empty or not (referred to as the non-emptiness problem) is Ackermann-complete [11]. But the question has not been addressed for ν-A and LaMA. For finite-memory automata (FMA), it is known that membership (testing whether a word belongs to the language) and non-emptiness are NP-complete [18]. Knowing whether a language is included in another is undecidable for FMA when considering their non-deterministic version, but it is PSPACE-complete for deterministic ones [16]. In [12] it is shown that the non-emptiness problem for Variable Finite Automata (VFA) is NL-complete, while membership is NP-complete. For FRA and Guessing-Register Automata (as they are called in [15]) we only know that both problems are decidable but we do not know the accurate complexity class. Finally, for data-languages where data-words are sequences of pairs of a letter from a finite alphabet and an element from an infinite set and the latter can only be compared for equality, the non-emptiness problem for FMA is PSPACE-complete [10], for DA and CMA, membership and non-emptiness are only shown to be decidable, but no complexity is given [7, 6].

Contributions. In this paper, we close some open problems on the complexity of membership and non-emptiness. We first prove that testing membership for HRA, ν-A and LaMA is an equivalent problem. Then we address complexity and show that the problem is NP-complete with a reduction of 3-SAT to LaMA. Non-emptiness appears to be a much harder problem. We show that the non-emptiness problem is PSPACE-complete for ν-A by reducing TQBF (True Fully Quantified Boolean Formula) to ν-A.

The paper is organized as follows. The three formalisms are introduced and the main differences are recalled quickly in Section 2. Section 3 presents the complexity of the membership problem and Section 4 the non-emptiness one. Finally, Section 5 concludes with some remarks. Proofs and additional material can be found in [3].

2 Formalisms

All three formalisms are generalizations of finite state automata with memory over an infinite alphabet $\mathcal{U}$. For all of them, configurations (q, M) are pairs of a state of the automaton plus a memory context. A memory context assigns a set of letters to each identifier of the memory, variable or history depending on the formalism under consideration.

▶ **Definition 1** (Memory context). *Given a finite set of memory identifiers or variables V and an infinite alphabet $\mathcal{U}$, we define a memory context M as an assignment: $M : V \to 2^{\mathcal{U}}$ where $M(v) \subset \mathcal{U}$ is the finite set of letters* assigned *to v.*

The definition of accepted language common to the three formalisms is, as customary:

▶ **Definition 2** (Accepted language). *For an automaton A (LaMA, ν-A or HRA), the language of A is the set of words recognized by A: $\mathcal{L}(A) = \{w \in \mathcal{U}^* \mid (q_0, M_0) \overset{w}{\underset{A}{\Longrightarrow}} (q_f, M)$ s.t. $q_f \in F\}$, where $\overset{w}{\underset{A}{\Longrightarrow}}$ is the extension to sequences of transitions of $\overset{u}{\underset{A}{\longrightarrow}}$.*

2.1 n-Layered Memory Automata

We start with n-Layered Memory Automata (n-LaMA). The idea is that finite state automata are enriched with n layers each containing a finite number of variables. Variables on the same layer satisfy the *injectivity* constraint: variables on a given layer $l \in [1, n]$ (denoted v^l) do not share letters of the alphabet: $\forall v_1 \neq v_2 \in V, M(v_1^l) \cap M(v_2^l) = \emptyset$. Upon reading a letter, a transition can test if the letter is already stored in a set of variables and add a letter to a set of variables. The non-observable transition (ε-transition) empties a set of variables.

▶ **Definition 3** (n-LaMA). *An n-LaMA A is defined by the tuple $(Q, q_0, F, \Delta, V, n, M_0)$, where:*

- *Q is a finite set of states,*
- *$q_0 \in Q$ and $F \subseteq Q$ are respectively an initial state and a set of final states,*
- *Δ is a finite set of transitions,*
- *n is the number of layers, and V is a finite set of variables, denoted v^l with $l \in [1, n]$,*
- *$M_0 : V \mapsto 2^{\mathcal{U}}$ is an initial memory context.*

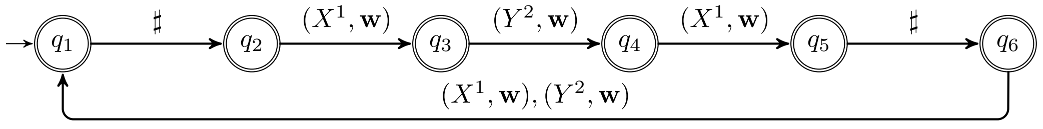


Figure 2 A 2-LaMA A_p recognizing $\mathcal{P}(2) \cap \mathcal{P}(3)$ from Example 8.

A *fresh letter at layer* l, is a letter that is associated with no variable of this layer. The set of transitions $\Delta = \Delta_o \cup \Delta_\varepsilon$ encompasses two kinds of transitions with Δ_o the set of *observable transitions* that consume a letter of the input and Δ_ε the set of *non-observable transitions*, that do not consume any letter of the input but can reset a set of variables.

▶ **Definition 4** (Observable transition). *An observable transition is a tuple of the form:* $\delta = (q, \alpha, q') \in \Delta_o$ *where:*

- $q, q' \in Q$ *are the source and destination states of the transition,*
- $\alpha : [1, n] \to (V \times \{\mathbf{r}, \mathbf{w}\}) \cup \{\sharp\}$*, such that* $\alpha(l) = (v^l, \mathbf{x})$ *for* $\mathbf{x} \in \{\mathbf{r}, \mathbf{w}\}$ *and for some* $v^l \in V$ *indicates for each layer* l *which variable is examined by the transition.*

Notice that only one variable per layer can be examined, and it is not possible to have $\alpha(l) = (v^k, \mathbf{x})$ with $l \neq k$. The special symbol $\sharp$ indicates that no variable is to be read or written for a specific layer.

▶ Remark 5 (Any-letter transition). If $\forall l \in [1, n], \alpha(l) = \sharp$ (i.e., no variable is examined) then the transition is executed consuming whatever letter is in input.

▶ **Definition 6** (Non-observable transition). *A non-observable transition is a tuple of the form* $\delta_\varepsilon = (q, \mathbf{reset}, q') \in \Delta_\varepsilon$ *where:*

- $q, q' \in Q$ *are the source and destination states of* δ_ε*,*
- $\mathbf{reset} \subseteq 2^V$ *is the set of variables reset (i.e., emptied) by the transition.*

▶ **Definition 7.** *The semantics of an n-LaMA* $A = (Q, q_0, F, \Delta, V, n, M_0)$ *is defined as:*

- *An observable transition* (q, α, q') *can be executed on an input letter* u *from memory context* M *leading to* M'*:* $(q, M) \underset{A}{\xrightarrow{u}} (q', M')$ *if for each* $\alpha(l) \neq \sharp$ *such that* $\alpha(l) = (v^l, \mathbf{x})$*:*
  - *if* $\mathbf{x} = \mathbf{r}$*, then* $u \in M(v^l)$ *and* $M'(v^l) = M(v^l)$ *;*
  - *if* $\mathbf{x} = \mathbf{w}$*, then* u *is fresh for layer* l *and* u *is added to* v^l *in the reached memory context:* $M'(v^l) = M(v^l) \cup \{u\}$*.*

  All variables v^l *not labeled through* α *remains associated to the same letters : if* $\alpha(l) = \sharp$ *or* $\alpha(l) = (v_1^l, x)$ *and* $v_1^l \neq v^l$ *then* $M'(v^l) = M(v^l)$*.*
- *A non-observable transition* $(q, \mathbf{reset}, q')$ *can be executed from memory context* M *without reading any input letter leading to* M'*:* $(q, M) \underset{A}{\xrightarrow{\varepsilon}} (q', M')$*, where* $\forall v^l \in \mathbf{reset} : M'(v^l) = \emptyset$ *and otherwise* $M'(v^l) = M(v^l)$*.*

▶ **Example 8.** Let $\mathcal{P}(p) = \{u_1 \dots u_s \mid \forall j, k > 0, j \neq k,\ u_{j \cdot p} \neq u_{k \cdot p}\}$, be the language recognizing words where the letters at positions, which are multiples of p are all different whereas the others are not constrained. Fig. 2 depicts a 2-LaMA for $\mathcal{P}(2) \cap \mathcal{P}(3)$.

2.2 ν-automata

ν-automata (ν-A) can be seen as a restricted version of LaMA with only one layer. Hence, each variable is constrained under the injectivity property, and no letter can be stored in more than one variable.

▶ **Definition 9** (ν-A). *A ν-A is defined as a tuple $(Q, q_0, F, \Delta, V, M_0)$, where*

- *Q is a finite set of states containing an initial state $q_0 \in Q$ and a set of final ones $F \subseteq Q$,*
- *V is a finite set of variables that may initially be storing a finite amount of letters from the infinite alphabet $\mathcal{U}$, as specified by the initial memory context M_0,*
- *and Δ is a finite set of transitions.*

As before, $\Delta = \Delta_o \cup \Delta_\varepsilon$ where Δ_o is the set of *observable transitions* and Δ_ε is the set of *non-observable* ones. Differently from LaMA, observable transitions are decoupled in read and write transitions.

▶ **Definition 10** (Observable transition). *An observable transition can be of two kinds: $(q, v, \mathbf{r}, q')$ and $(q, v, \mathbf{w}, q')$ ($\mathbf{r}$ for read and $\mathbf{w}$ for write) where $q, q' \in Q$ are the source and destination states and $v \in V$.*

▶ **Definition 11** (Non-observable transition). *A non-observable transition is a tuple of the form $\delta_\varepsilon = (q, \mathbf{reset}, q') \in \Delta_\varepsilon$ where: $q, q' \in Q$ are the source and destination states of δ_ε, $\mathbf{reset} \subseteq 2^V$ is the set of variables reset by the transition.*

▶ **Definition 12.** *The semantics of a ν-A $A = (Q, q_0, F, \Delta, V, M_0)$ is defined as:*

- *An observable transition $(q, v, \mathbf{x}, q')$ reading input letter u can be executed from memory context M leading to M': $(q, M) \xrightarrow[A]{u} (q', M')$ if for each v:*
  - *if $\mathbf{x} = \mathbf{r}$, then $u \in M(v)$ and $M'(v) = M(v)$;*
  - *if $\mathbf{x} = \mathbf{w}$, then u is fresh in M and u is added to v in the reached memory context: $M'(v) = M(v) \cup \{u\}$.*

  All other variables $v_1 \neq v$, remains associated to the same letters $M'(v_1) = M(v_1)$.
- *A non-observable transition $(q, \mathbf{reset}, q') \in \Delta_\varepsilon$ can be executed from memory context M leading to M': $(q, M) \xrightarrow[A]{\varepsilon} (q', M')$ without reading any input letter, where $\forall v \in \mathbf{reset}$: $M'(v) = \emptyset$ and otherwise $M'(v) = M(v)$.*

▶ Remark 13. Analogously to LaMA, we consider *any-letter transitions*, denoted by $(q, \sharp, q')$ with $\sharp \notin \mathcal{U}$, which are enabled whenever a letter is read and the memory context of the target configuration is the same as the origin's one.

Notice that any-letter transitions do not alter the expressive power of ν-A nor the complexity of its problems. Indeed, it is a sort of macro that can be encoded by a set of transitions searching for the presence of a letter or its freshness over the whole set V. To do so, one needs as many reading transitions as variables to allow the firing with any letter in memory. For fresh letters, one needs a transition writing in an extra variable, which is reset immediately after.

2.3 History-Register Automata

HRA are automata provided with a finite set H of histories, i.e., variables storing a finite subset of letters of the infinite alphabet $\mathcal{U}$. To simplify the presentation, we consider HRA defined only with histories and no registers. The latter does not provide additional expressiveness [11]. An essential distinction between HRA and LaMA or ν-A is that different histories are allowed to store the same letter (i.e., there is no injectivity constraint). Thus, an observable transition is annotated with the exact set of histories that should contain the letter read to enable it. This entails that for each observable transition the whole memory has to be explored while LaMA allow ignoring some layers using symbol $\sharp$ (this can be crucial while implementing the formalisms[2]).

[2] Implemented in tool available at `https://github.com/clementber/MaTiNA/tree/master/LaMA`

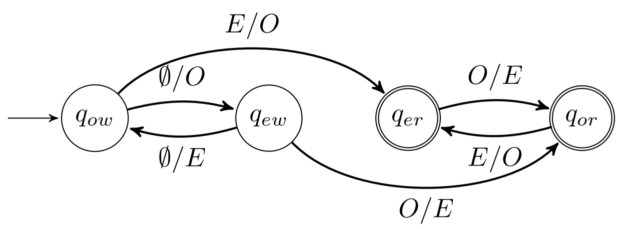


Figure 3 Example of an HRA A_h.

▶ **Definition 14** (HRA). *A History-Register Automata is defined as a tuple of the form $A = (Q, q_0, F, \Delta, H, M_0)$ where Q is the set of states, q_0 the initial one, F the set of final ones, Δ the set of transitions, H a finite set of histories and M_0 the initial memory context. The set of transitions $\Delta = \Delta_o \cup \Delta_\varepsilon$ are of the form:*

- *$(q, H_r, H_w, q') \in \Delta_o$ where $H_r, H_w \subseteq H$ (for read and write), which is an observable transition and*
- *$(q, H_\emptyset, q') \in \Delta_\varepsilon$ where $H_\emptyset \subseteq H$, which is a non-observable transition.*

An observable transition (q, H_r, H_w, q') is enabled if letter u is present in exactly all the histories in H_r and not present in $H \setminus H_r$. After the transition, u is present only in the histories in H_w. Notice that this allows moving an input letter from one set of histories to another, or even forgetting it if $H_w = \emptyset$. This is not possible in ν-A and LaMA. Finally, if $H_r = \emptyset$ then the input letter has to be fresh (absent from every history).

▶ **Definition 15.** *The semantics of an HRA $A = (Q, q_0, F, \Delta, H, M_0)$ is defined as:*

- *an observable transition (q, H_r, H_w, q') is enabled for memory context M when reading letter $u \in \mathcal{U}$: $(q, M) \xrightarrow[A]{u} (q', M')$ if $u \in M(h_r) \Leftrightarrow h_r \in H_r$ and $\forall h_w \in H_w : M'(h_w) = M(h) \cup \{u\}$ and $\forall h \notin H_w$: $M'(h) = M(h) \setminus \{u\}$;*
- *a non-observable transition $(q, H_\emptyset, q')$ is enabled for any memory context M and allows to move from configuration (q, M) to (q', M'): $(q, M) \xrightarrow[A]{\varepsilon} (q', M')$, where all the histories in $H_\emptyset$ have been reset in M'.*

▶ **Example 16.** Fig. 3 depicts an HRA that, with an initially empty memory context, recognizes the language

$$\begin{aligned}\{u_1u_2\dots u_n \mid\ & \exists k < n, \forall i, j \in [1, k], u_i = u_j \Leftrightarrow i = j,\\ & \forall m \in]k, n], \exists p < m, u_p = u_m, p \bmod 2 \neq m \bmod 2, \nexists q \in]p, m[, u_m = u_q\}\end{aligned}$$

The two transitions looping between states q_{ow} and q_{ew} allow us to recognize words where the first k letters are all different from each other. Letters are stored in histories O (odd) and E (even) to remember the parity of the position they are read at. The transitions between states q_{er} and q_{or} allow us to recognize words whose suffix is only composed of repetitions of the k first letters, with the additional constraint that they can only occur at a position with opposed parity wrt the previous occurrence. Thus, if a letter was read for the last time at an even position, it is stored in history E and can only be read in an odd position. Once it is read, it is transferred to the O history to remember it can only be read at an even position the next time.

3 Complexity of the membership problem

We know that each ν-A can be encoded into a LaMA and respectively each LaMA can be encoded into an HRA both recognizing the same language [4]. The encoding from LaMA to HRA is exponential in the number of layers, hence we know that the complexity of problems for HRA gives an upper bound to the complexity of the same problem for LaMA and ν-A. In this section, we show that the complexity of the membership problem (i.e., given an automaton A and a word w decide whether $w \in \mathcal{L}(A)$) falls in the same class for these three automata models. To do so, we show that the membership problem for LaMA can be simulated using ν-A, and the same can be done for HRA using LaMA.

Simulating the membership for LaMA in ν-automata. The idea is to represent an n-LaMA as a product of n ν-A, one for each layer. The main limitation is that having just one layer makes the injectivity constraint stronger. Indeed, it is not possible to trivially treat a same letter stored on different layers. To cope with this difficulty, we rename the word under consideration, replacing consistently each letter with a sequence of new ones - one per layer of the LaMA: i.e., for an n-LaMA the letter $u \in w$ is replaced by the letters $u^1, ..., u^n$ where all the u^i are different in order to have the letters belonging to different layers all distinct from each other. This renaming is always possible as the alphabet $\mathcal{U}$ is infinite. For example, for the word aba, a consistent renaming, for a 2-LaMA, could produce $a^1\, a^2\, b^1\, b^2\, a^1\, a^2$.

▶ **Definition 17** (Renaming). *$\xi^n : \mathcal{U} \to \mathcal{U}^n$ is a* renaming *function that given a letter $u \in \mathcal{U}$ generates a new sequence of n letters $u^1 \dots u^n$ with for all $i \neq j \in [1, n]$ $u^i \neq u^j$ and such that if $u_1 \neq u_2$ then for all $i, j \in [1, n]$, $u_1^i \neq u_2^j$. $\xi^n(u_1 \dots u_m) = u_1^1 \dots u_1^n \dots u_m^1 \dots u_m^n$ is its pointwise extension to words .*

Let $A = (Q, q_0, F, \Delta, V, n, M_0)$ be an n-LaMA and $w = u_1 \dots u_m \in \mathcal{U}^*$. We know that $w \in \mathcal{L}(A)$ if and only if there is a finite sequence of transitions such that for some M_f, $(q_0, M_0) \overset{w}{\underset{A}{\Longrightarrow}} (q_f, M_f)$ with $q_f \in F$. It is then possible to construct a ν-A that accepts $\xi^n(w)$, which simulates the recognition process of the n-LaMA over the word w. To do so, we encode every observable transition of A into a sequence of transitions successively simulating the constraints applied to variables of each layer. Moreover, we apply the renaming function ξ^n to the initial memory context. In order to simplify the notations, in the following, we denote by $x\lfloor_k$ the projection onto the k-th element of tuple x, e.g., $(a, b, c)\lfloor_2 = b$.

▶ **Definition 18** (Encoding of a memory context). *Let M be the memory context over the set of variables V over n layers, then $\forall v^l \in V$, its renaming through ξ^n, is defined as $[\![M]\!]_\xi(v^l) = \{\xi^n(u)\lfloor_l \mid u \in M(v^l)\}$.*

▶ **Definition 19** (Encoding of a LaMA). *Let $A = (Q, q_0, F, \Delta, V, n, M_0)$ be an n-LaMA, then the ν-A $[\![A]\!]_\xi = (Q', q_0', F', \Delta', V', M_0')$ is the encoding of A through the renaming ξ^n, where:*

- *$Q' = Q \cup Q_o$ and the set of states $Q_o = \{q_\delta^l \mid \delta = (q, \alpha, q') \in \Delta, l \in [2, n]\}$ is used by the sequence of transitions simulating each observable transition of A, $q_0{}' = q_0$ and $F' = F$;*
- *$V' = V$ is the set of variables of A flattened on one layer;*
- *$M_0{}' = [\![M_0]\!]_\xi$ is the initial memory context of A renamed in case it is not initially empty;*
- *$\Delta' = \Delta'_o \cup \Delta_\varepsilon$ where Δ_ε is the set of all non-observable transitions of A, and Δ'_o contains the encoding of every observable transition $\delta = (q, \alpha, q')$ of A, which is a sequence of transitions with $q_\delta^1 = q$ and $q_\delta^{n+1} = q'$ such that*

$$\begin{aligned}\Delta'_o &= \{(q_\delta^l, v^l, \mathbf{x}, q_\delta^{l+1}) \mid \delta = (q, \alpha, q') \in \Delta, l \in [1, n], \alpha(l) = (v^l, \mathbf{x})\} \\ &\cup \{(q_\delta^l, \sharp, q_\delta^{l+1}) \mid \delta = (q, \alpha, q') \in \Delta, l \in [1, n], \alpha(l) = \sharp\}.\end{aligned}$$

Notice that the language accepted by the encoded ν-A $[\![A]\!]_\xi$ of a LaMA A is an over-approximation of the language accepted by A: $\xi(\mathcal{L}(A)) \subseteq \mathcal{L}([\![A]\!]_\xi)$. They are equal only when the LaMA has one layer (i.e., $n = 1$). Nonetheless, this construction may be used to test the membership of a word w to $\mathcal{L}(A)$. The proof is a simple induction on the length of the derivation of w and $\xi^n(w)$ [3].

▶ **Theorem 20.** *Let A be an n-LaMA. $w \in \mathcal{L}(A)$ if and only if $\xi^n(w) \in \mathcal{L}([\![A]\!]_\xi)$.*

Simulating the membership for HRA in LaMA. This section presents how to solve the membership problem for HRA using LaMA. The difference in expressiveness between HRA and LaMA comes from the ability of HRA of removing letters from histories when they are read. We resort to an encoding of words where each letter is duplicated and annotated with a number representing how many occurrences of that letter have been encountered so far. In detail, the first copy of the letter keeps the information on the number of occurrences of the letter seen so far and the second one the number of occurrences including the present one.

▶ **Example 21.** Take $w = abaca$ then the encoded word is $w' = a^0a^1\ b^0b^1\ a^1a^2\ c^0c^1\ a^2a^3$

The idea behind the encoding of observable transitions is to use the first copy to check the presence and absence of the letter in every variable (simulating the role of H_r) while the second one (that is always fresh) can be used to simulate writing and removal (hence simulating H_w). More precisely, once we add an annotated letter to a variable, the encoded automaton will ensure that the variable always stores the last seen occurrence of that letter. Thus, removing a letter from a history consists in not storing the last seen occurrence of the letter in the corresponding encoded variable. Clearly, all the letters annotated with a number smaller than the current one will not be used in any of the transitions, representing a form of garbage.

We consider a renaming function $\zeta_i : \mathcal{U} \to \mathcal{U}^2$ which replaces u by a pair of letters $u^{i-1}u^i$ for any $i \in \mathbb{N}^+$. Then, we define the encoding of words $\zeta : \mathcal{U}^* \to \mathcal{U}^*$ as follows $\zeta(u_1 \ldots u_m) = \zeta_{i_{u_1}}(u_1) \ldots \zeta_{i_{u_m}}(u_m)$ where each i_{u_j} is the number of occurrences of u_j seen in the prefix $u_1 \ldots u_j$. Notice that when considering the word up to letter u_j, $\zeta_{i_{u_j}}(u_j)\lfloor_2$ is always a new letter (e.g., a fresh letter with respect to those in $\zeta(u_1 \ldots u_j)$).

▶ **Definition 22** (Encoding of an HRA). *Let $A = (Q, q_0, F, \Delta_o \cup \Delta_\varepsilon, \{h_1, \ldots h_n\}, M_0)$ be an HRA, its encoding into an n-LaMA is $[\![A]\!]_\zeta = (Q', q_0', F', \Delta', V', n, M_0')$ where:*

- *$Q' = Q \cup Q_o$ and the set of states $Q_o = \{q_\delta \mid q \in Q, \delta = (q, H_r, H_w, q') \in \Delta\}$ is used by the sequence of transitions simulating each observable transition of A;*
- *$q_0' = q_0$ and $F' = F$;*
- *$V' = \{h^l, \omega^l \mid l \in [1, n]\}$ and for each layer $l \in [1, n]$, h^l plays the role of history h_l and ω^l is used to check the absence of letters in h^l.*
- *$M_0'(h^l) = \{\zeta_1(u)\lfloor_1 \mid u \in M_0(h_l)\}$ and $M_0'(\omega^l) = \emptyset$ for all $l \in [1, n]$ meaning that M_0' is as M_0 with all letters renamed with ζ_1 and empty for all extra variables;*
- *$\Delta' = \Delta'_\varepsilon \cup \Delta'_o$ with*
  - *$\Delta'_\varepsilon = \{(q, \{h^l \mid h_l \in H_\emptyset\}, q') \mid (q, H_\emptyset, q') \in \Delta_\varepsilon\}$, is the direct translation of the ε-transitions in A.*
  - *$\Delta'_o = \{(q, \alpha_{H_r}, q_\delta), (q_\delta, \alpha_{H_w}, q') \mid \delta = (q, H_r, H_w, q') \in \Delta_o\}$ with for all $l \in [1, n]$*

$$\alpha_{H_r}(l) = \begin{cases} (h^l, \mathbf{r}) & \text{if } h_l \in H_r \\ (\omega^l, \mathbf{w}) & \text{if } h_l \notin H_r \end{cases} \quad \text{and} \quad \alpha_{H_w}(l) = \begin{cases} (h^l, \mathbf{w}) & \text{if } h_l \in H_w \\ \sharp & \text{if } h_l \notin H_w \end{cases}$$

  the first simulating the guard part of the observable transition and the second the writing/relocation.

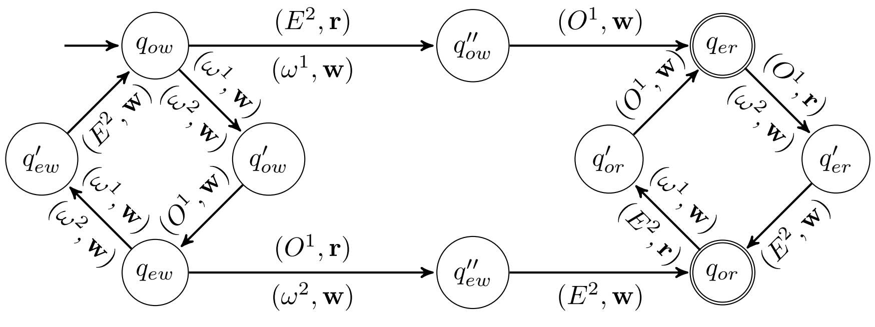


Figure 4 The 2-LaMA $[\![A_h]\!]_\zeta$, encoding of the HRA A_h from Example 16.

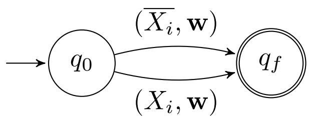
▶ **Example 23.** Fig. 4 depicts the encoding applied to the HRA of Example 16. Given the word $w = abcabb$ and its renaming $\zeta(w) = a^0a^1b^0b^1c^0c^1a^1a^2b^1b^2b^2b^3$, we present how the encoding works. A_h has two histories: $H = \{O, E\}$, thus the set of variables V of $[\![A_h]\!]_\zeta$ is $\{O^1, \omega^1\}$ on layer 1 and $\{E^2, \omega^2\}$ on layer 2. Let $(q_{ow}, M_\emptyset)$ be the initial state for both automata, with $M_\emptyset$ the memory context where all variables/histories are empty.

When reading the first letter a in A_h, only transition $(q_{ow}, M_\emptyset) \xrightarrow[A_h]{a} (q_{ew}, M_1)$ is enabled as a is not stored in any of the histories in $M_\emptyset$, as a consequence, a is added to O in M_1. In $[\![A_h]\!]_\zeta$, this transition is encoded with the sequence $(q_{ow}, M_\emptyset) \xrightarrow[[\![A_h]\!]_\zeta]{a^0} (q'_{ow}, M') \xrightarrow[[\![A_h]\!]_\zeta]{a^1} (q_{ew}, M'_1)$. The first transition when reading a^0, checks if a^0 is absent from both O^1 and E^2 using ω^1 and ω^2 with the injectivity constraint. When reading a^1 the transition $q'_{ow} \to q_{ew}$ writes the letter in O^1. Note that a^0 is still stored in ω^1 and ω^2, but it will never be read again (as the renaming ζ always increases the index of letters).

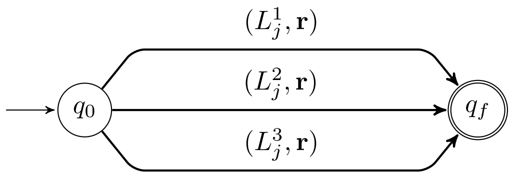
Then, when A_h read the first occurrence of b, the only enabled transition is $(q_{ew}, M_1) \xrightarrow[A_h]{b} (q_{ow}, M_2)$, where b is stored in E is M_2. And when reading c the only transition enabled is $(q_{ow}, M_2) \xrightarrow[A_h]{c} (q_{ew}, M_3)$ with O storing both a and c while E only stores b. In $[\![A_h]\!]_\zeta$, this sequence of transitions is encoded by enabling the sequence of transitions $(q_{ew}, M'_1) \xrightarrow[[\![A_h]\!]_\zeta]{b^0} (q'_{ew}, M''_1) \xrightarrow[[\![A_h]\!]_\zeta]{b^1} (q_{ow}, M'_2) \xrightarrow[[\![A_h]\!]_\zeta]{c^0} (q'_{ow}, M''_2) \xrightarrow[[\![A_h]\!]_\zeta]{c^1} (q_{ew}, M'_3)$. With M'_2 storing b^1 in E^2 and M'_3 storing c^1 in O^3 in addition to a^1. This is the only sequence of transition that can be enabled as b_0 was not stored in O^1 in the state (q_{ew}, M'_1) and c^0 was not stored in E^2 in (q_{ow}, M'_2).

When reading the second occurrence of a, the only enabled transition is $(q_{ew}, M_3) \xrightarrow[A_h]{a} (q_{or}, M_4)$ where a is transferred from O to E in M_4. In $[\![A_h]\!]_\zeta$ this is encoded by the sequence of transitions $(q_{ew}, M'_3) \xrightarrow[[\![A_h]\!]_\zeta]{a^1} (q''_{ew}, M''_3) \xrightarrow[[\![A_h]\!]_\zeta]{a^2} (q_{or}, M'_4)$. The first transition is the only one enabled in configuration (q_{ew}, M'_3) as a^1 is already stored in O^1, thus it would be impossible to write it in ω^1 to enable the transition to q'_{ew}. In M'_4, the letter a^2 is stored in E^2 along with b^1, while a^1 is still stored in O^1 but will never be read again in $\zeta(w)$, so it can be ignored. This is how the transfer mechanism is encoded in this construction.

Reading bb, the last two letters of w, will enable in A_h the sequence $(q_{or}, M_4) \xrightarrow[A_h]{b} (q_{er}, M_5)$ transferring b from E to O in M_5 and then enabling $(q_{er}, M_5) \xrightarrow[A_h]{b} (q_{or}, M_6)$ transferring b back from O to E in M_6. In $[\![A_h]\!]_\zeta$, this is encoded by reading the letters $b^1b^2b^2b^3$ and



(a) The gadget for the existentially quantified variable x_i.



(b) The 3SAT clause gadget for $C_j = (L_j^1 \vee L_j^2 \vee L_j^3)$.

Figure 5 Gadgets used for showing NP-hardness of the membership problem.

enabling the loop of transition between states q_{or}, q'_{or}, q_{er} and q'_{er}. Looking if the previous occurrence of b, here b^2 (resp. b^3), is stored in E^2 (resp. O^1) by reading in the variable. Also checking if it is absent from O^1 (resp. E^2) by writing in the ω of the same layer. Then writing the next occurrence of b, here b^3 (resp. b^4), in O^1 (resp. E^2) to encode its transfer.

Notice that, as before, the language recognized by $[\![A]\!]_\zeta$ is actually larger than $\mathcal{L}(A)$.

▶ Remark 24. In [11], HRA are presented with a set of registers able to store only one letter at a time. Their content is overwritten whenever a letter is written into it. The authors proved that HRA using only histories are as expressive as the ones using both histories and register. However, the construction presented to remove registers is exponential in their number. This is caused by the need of decoupling the overwriting into two phases, first, one uses the content to verify if an observable transition is enabled and then erases the content of histories. The exponential construction comes from the fact that to keep the languages equivalent, for each phase, one can use only one observable transition. Instead, to show membership we do not need to prove the equivalence of languages and the construction in Definition 22, already splits transitions into these two phases, using two observable transitions. Hence, it can be extended to registers avoiding the exponential cost.

▶ **Theorem 25.** *Let A be an HRA. $w \in \mathcal{L}(A)$ if and only if $\zeta(w) \in \mathcal{L}([\![A]\!]_\zeta)$.*

Complexity. The two previous encodings give polynomial reductions of the membership problem from HRA to LaMA and from LaMA to ν-A. Therefore, there is a polynomial reduction of the problem for HRA to ν-A. The expressiveness results from [4] give a linear construction from ν-A to LaMA and an exponential construction, in the number of layers, from LaMA to HRA. As ν-A are 1-LaMA, the same construction can be used to translate a ν-A into an HRA of polynomial size. This implies an equivalence of complexity class of the membership problem for ν-A and HRA, as well as for ν-A and LaMA. By transitivity, we get the same equivalence between LaMA and HRA. Next, we show that the membership problem for LaMA is NP-complete. For the hardness part, this is shown by resorting to a reduction from the 3SAT problem, while the completeness part follows by observing what would be the cost of executing a word on an automaton. Fig. 5 depicts the intuition behind the encoding of a 3SAT instance. The idea is that the gadget in Fig. 5a chooses non-deterministically the truth assignment of X_i or $\overline{X_i}$ and the one in Fig. 5b checks that this assignment indeed satisfies the given clauses.

▶ **Theorem 26.** *The membership problem for LaMA is NP-complete.*

Hence we can conclude that:

▶ **Corollary 27.** *The membership problems for ν-A, LaMA and HRA is NP-complete.*

As a direct consequence and looking at the expressiveness hierarchy in Fig. 1 we can also give a complexity class for the membership problem in FRA. Indeed, since FMA can be encoded into FRA [19], we can deduce NP-hardness, and completeness follows from their encoding into LaMA [4].

▶ **Corollary 28.** *The membership problem for FRA is NP-complete.*

4 Complexity of the non-emptiness problem

The non-emptiness problem consists in deciding whether the language accepted by an automaton is non-empty, or in other words checking if there is a path from the initial configuration to a final configuration. As mentioned before, in [11], it has been shown that deciding the non-emptiness for HRA is Ackermann-complete. Still, the complexity for non-emptiness is known neither for LaMA nor for ν-A. We start with the non-emptiness Problem for ν-A. We show that the problem is PSPACE-complete. To do so, we reduce the TQBF problem (true fully quantified Boolean formula) to ν-A non-emptiness. TQBF is known to be PSPACE-complete (Meyer-Stockmeyer theorem [1]).

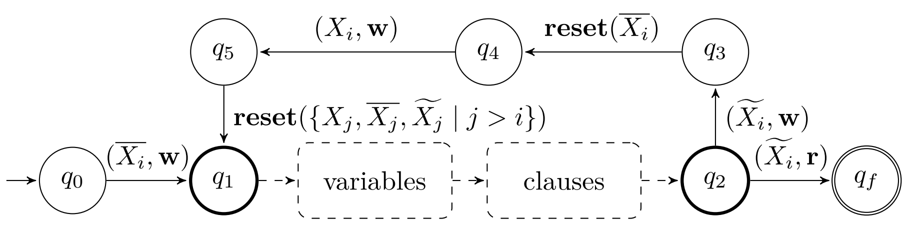
▶ **Lemma 29.** *The non-emptiness problem is PSPACE-hard for ν-A.*

Proof. Let νNEP be the short for non-emptiness Problem for ν-A.

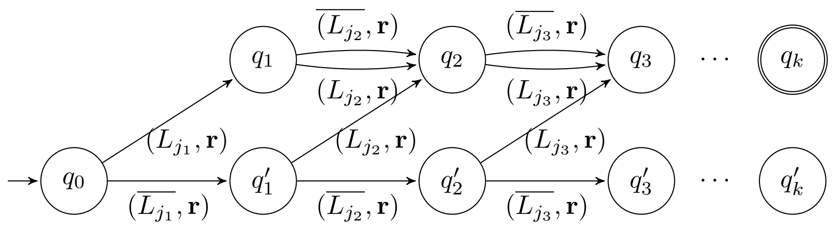
We show that TQBF can be reduced to νNEP. Let $Q_1x_1 \dots Q_nx_n(C_1 \wedge \dots \wedge C_m)$, be a fully quantified Boolean formula, where each $Q_i \in \{\forall, \exists\}$ and each C_j is a clause comprising at most n literals (x_i or $\overline{x_i}$). We assume that literals in clauses are ordered according to the order of variable declarations and at most one literal per variable is present. To encode TQBF in ν-A we consider:

- for each existentially quantified x_i, variables X_i and $\overline{X_i}$, and the gadget depicted in Fig. 5a, used in the proof of Theorem 26;
- for each universally quantified x_i, variables X_i, $\overline{X_i}$ and $\widetilde{X_i}$, and the gadget depicted in Fig. 6a. Variable $\widetilde{X_i}$ is used as a flag to indicate that all possible truth assignments of x_i have been considered. The initial transition of the gadget initializes variable $\overline{X_i}$ to 1_i. The dashed automaton connected between states q_1 and q_2, handling other variables and the clauses, is constructed recursively. The looping part starting in state q_2 writes letter 2_i into variable $\widetilde{X_i}$, which is used after browsing once again the dashed part to reach the final state q_f. After this, variable $\overline{X_i}$ is reset and then variable X_i is initialized to 1_i to consider the other truth assignment of x_i. From state q_5 to q_1 all the variables for x_j, with j from $i+1$ to n are reset to reinitialize their truth assignments;
- for each clause C_j, a clause gadget depicted in Fig. 6b. It tests literals one after the other and takes the oblique transition for the first which makes the clause satisfied, which means that the remaining literals are just read up to the end of the clause, which is satisfied if q_f is reached.

In order to construct the ν-A A encoding the instance of TQBF we connect first the clause gadgets by merging the final state of a clause gadget with the initial state of the next one. Let C be the resulting automaton. Then, we connect to C the gadgets for variable declarations starting from the nth, i.e., the last in the order of declarations. If the variable is under an existential quantifier, we connect the existential variable gadget in front of the automaton obtained so far by merging its final state with the initial state of C. If the variable is under a universal quantifier, we connect the corresponding gadget by merging the initial state of C with state q_1 of the gadget, and the final state of C with state q_2 of the gadget.


q_0
$(\overline{X_i}, \mathbf{w})$
q_1
variables
clauses
q_2
$(\widetilde{X_i}, \mathbf{r})$
q_f
$(\widetilde{X_i}, \mathbf{w})$
q_3
$\mathbf{reset}(\overline{X_i})$
q_4
$(X_i, \mathbf{w})$
q_5
$\mathbf{reset}(\{X_j, \overline{X_j}, \widetilde{X_j} \mid j > i\})$

(a) The gadget for universally quantified variable x_i.



(b) The gadget for TQBF clause $C_j = (L_{j_1} \vee \ldots \vee L_{j_k})$.

Figure 6 Gadgets used for showing NP-hardness of the emptiness problem for ν-A.

We connect this way, i.e., following the inverse order of declarations, the gadgets for all the remaining variable declarations. The initial state of the first declared variable gadget is the initial state of A and the unique state q_f of the final construction is the unique final state of A. Finally, the input word w is obtained recursively for each TQBF instance by the function *input*(ϕ) defined in Algorithm 1. The construction of the word follows the intuition given above (for the construction of the automaton), that is: it unfolds the loops generating the letters needed at each step.

Algorithm 1 Function to generate the word accepted by TQBF automaton.

```
1: function INPUT($\phi$)                                   ▷ $\phi = Q_1x_1 \ldots Q_nx_n\ C_1 \ldots C_m$
2:     $\forall i \in [1,n] : \mathit{init}(x_i) = 1_i$
3:     $\forall i \in [1,n] : \mathit{end}(x_i) = 2_i$
4:     $\forall j \in [1,m] : w_j$        ▷ contains exactly one $1_i$ for each $x_i$ or $\overline{x_i}$ present in clause $C_j$
5:     if $\phi = \emptyset$ then return $\epsilon$
6:     else if $\phi = \exists x_i\ \phi'$ then return $\mathit{init}(x_i).\mathit{input}(\phi')$
7:     else if $\phi = \forall x_i\ \phi'$ then return $\mathit{init}(x_i).\mathit{input}(\phi').\mathit{end}(x_i).\mathit{init}(x_i).\mathit{input}(\phi').\mathit{end}(x_i)$
8:     else if $\phi = C_i\ \phi'$ then return $w_i.\mathit{input}(\phi')$
```

Example: for the TQBF instance $\exists x_1\ \forall x_2\ \forall x_3\ \exists x_4((x_1 \vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3} \vee x_4))$, Fig. 7 represents a general construction schema of the corresponding ν-A, and the input word is $1_1 1_2 1_3 1_4 1_1 1_4 1_2 1_3 1_4 2_3 1_3 1_4 1_1 1_4 1_2 1_3 1_4 2_3 2_2 1_2 1_3 1_4 1_1 1_4 1_2 1_3 1_4 2_3 1_2 1_3 1_4 1_1 1_4 1_2 1_3 1_4 2_3 2_2$.

Note that every gadget of the automaton is deterministic, except for the existential variable gadget. The size of A is polynomial in the size of the TQBF expression. The length of the word generated by Algorithm 1 is in $\Omega(2^n)$ but it is not a parameter of the construction of A. Clearly, only a word generated by Algorithm 1 (or a consistent renaming) can be accepted by A starting with an empty memory context. Such a word can be accepted if and only if there is a solution to the TQBF instance. ◀

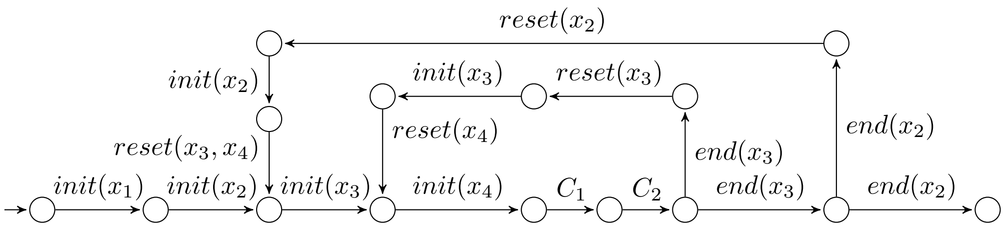


Figure 7 Schema of construction for TQBF instance $\exists x_1\ \forall x_2\ \forall x_3\ \exists x_4\ ((x_1 \vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3} \vee x_4))$.

It remains to show that the non-emptiness problem for ν-A is in PSPACE. This accounts for showing that if the language recognized by an ν-A is non-empty then it contains a word whose size together with the length of the transition path needed to accept it, are exponentially bounded with respect to the size of the ν-A. To this aim, we "build" a finite state machine (FSM) characterizing an abstraction of the state space of the ν-A.

If one can choose the letters to read, the idea is that observable transitions that write a letter in a variable are never blocking. Since the alphabet is infinite there is always a fresh letter that can be added, which we call a token. Instead, observable transitions that read a letter from a variable are blocking, in the sense that concerned variables must contain at least a letter (that we call a key). The first step towards the construction of the FSM is to build a *canonical* ν-A such that a word accepted by the canonical automaton will also be accepted by the initial ν-A A and each word accepted by A will have a corresponding canonical version. Consider a ν-A $A = (Q, q_0, F, \Delta, V, M_0)$, its *canonical* version $cano(A) = (Q, q_0, F, \Delta, V, M'_0)$ is an ν-A over the alphabets K and T, where:

- $K \subset \mathcal{U}$, such that $|K| = |V|$, is the set of *keys* k_v, each of them being associated with a variable $v \in V$. If $M_0(v) \neq \emptyset$, we select k_v in $M_0(v)$. Also, if $M_0(v) = \emptyset$, we select k_v such that $\forall v' \in V, k_v \notin M_0(v')$. The presence of a key in a variable v denotes the fact that v is non-empty.
- $T = \{t_1, t_2, \ldots\} \subset \mathcal{U}$, $K \cap T = \emptyset$, is an infinite set containing letters called *tokens* intended to be used only once, which are never stored in memory. Hence, no letter in T is present in the initial memory context of A, $\forall v \in V : M_0(v) \cap T = \emptyset$.
- For each $v \in V$, the initial memory context $M'_0(v)$ of $cano(A)$ is either empty if $M_0(v) = \emptyset$, or if $M_0(v) \neq \emptyset$, it only contains its key k_v.

Notice that a word w is accepted by $cano(A)$ if the following conditions hold:

1. $w \in (K \cup T)^*$, and if $t_i \in T$ appears in w then it occurs at most once,
2. let $(q_0, M'_0) \xRightarrow[cano(A)]{w} (q_f, M_f)$ with $q_f \in F$ be the accepting path for w then for each intermediate configuration (q, M) in the path and for each $k_v \in K$ either $k_v \in M(v)$ or for all $v' \in V$, $k_v \notin M(v')$.

Observe that $cano(A)$ is actually the same automaton as A but over a subset of the alphabet $\mathcal{U}$ and where for each $v \in V$, $M'_0(v) \subseteq M_0(v)$, hence it is easy to conclude that the language of $cano(A)$ is included in the one of A.

▶ **Lemma 30.** *Let A and $cano(A)$ be a ν-A and its canonical version. If a word $w \in \mathcal{L}(cano(A))$ then $w \in \mathcal{L}(A)$.*

We want to show that the language accepted by a ν-A A is empty if and only if the language accepted by $cano(A)$ is empty. The if part is the most involved and is the content of the following lemma.

▶ **Lemma 31.** *Let A be a ν-A and cano(A) and its canonical version. If $w \in \mathcal{L}(A)$ then there exists $w' \in \mathcal{L}(cano(A))$.*

Proof. Let $w = u_1 \dots u_n \in \mathcal{L}(A)$. Then there exists an accepting path $(q_0, M'_0) \overset{w}{\underset{A}{\Longrightarrow}} (q_f, M_f)$ with $q_f \in F$ and intermediate configurations $(q_i, M_i)_{i \geq 0}$. Depending on those intermediate configurations we build a new word $w' = u'_1 \dots u'_n$ and the corresponding path in *cano*(A) accepting w'. For each configuration (q_i, M_i), the construction maintains an invariant: $\forall v \in V, M'_i(v) = \{k_v\}$ if and only if $M_i(v) \neq \emptyset$. The proof proceeds by induction:
Base case: By construction the initial configuration of *cano*(A) satisfies the invariant.
Inductive step: We examine the transition $(q_i, M_i) \overset{\delta_i}{\underset{u_i}{\longrightarrow}} (q_{i+1}, M_{i+1})$. By inductive hypothesis, we know that there exists a sequence of transitions $(q_0, M'_0) \overset{u'_1 \dots u'_{i-1}}{\underset{cano(A)}{\Longrightarrow}} (q_i, M'_i)$ such that $\forall v \in V, M'_i(v) = \{k_v\}$ if and only if $M_i(v) \neq \emptyset$. We prove there is a letter u'_i leading to the configuration (q_{i+1}, M'_{i+1}) satisfying this property, through δ_i (by construction A and *cano*(A) are defined on the same set of transitions), we list all possible cases:

- $\boldsymbol{\delta_i = (q_i, \text{reset}, q_{i+1})}$**:** then $u_i = u'_i = \varepsilon$ and δ_i will lead to a configuration with $M'_{i+1}(v) = \emptyset$ if $v \in$ **reset** or $M'_{i+1}(v) = M'_i(v)$ otherwise. Hence satisfying the invariant.
- $\boldsymbol{\delta_i = (q_i, v, \text{r}, q_{i+1})}$**:** then $M_i(v) \neq \emptyset$ otherwise the transition could not be enabled, so $u'_i = k_v$ and by inductive hypothesis $M'_i(v) = \{k_v\}$. Since the memory context does not change for both automata, the invariant is satisfied;
- $\boldsymbol{\delta_i = (q_i, v, \text{w}, q_{i+1})}$**:** if $M_i(v) = \emptyset$, then $u'_i = k_v$, and k_v will be written in variable v in M'_{i+1} satisfying the invariant.
  If $M_i(v) \neq \emptyset$, then $u'_i = t_i \in T$ is a token and $M'_i(v) = M'_{i+1}(v)$ since tokens are not stored in memory. By inductive hypothesis we know that $M'_i(v) = \{k_v\}$ and as δ_i is a writing transition, then $M_{i+1} \neq \emptyset$, satisfying the invariant.

From the previous construction, it follows immediately that $w' \in \mathcal{L}(cano(A))$. ◀

Observe that, when reading a word $w \in \mathcal{L}(cano(A))$, we only need to store the letters belonging to K. Indeed, tokens in T may occur only once in w. This entails that tokens can only enable a write observable transition, while for read transitions keys are sufficient. Hence, in practice, tokens do not need to be added to the memory context. Hence the number of different configurations in *cano*(A) is bounded by $|Q| \cdot 2^{|K|}$ as:

- we have $|Q|$ states that can be encoded on $log|Q|$ bits, and
- there are $2^{|K|}$ possibilities to store the presence or not in the memory of letters in K (2 possibilities per letter encoded on 1 bit since each k_v can only be stored in v), so in total we need $|K|$ bits.

This shows that the number of configurations is finite. On top of this, as remarked above, transitions over letters in T do not add constraints on the memory context and they can be ignored. Hence the alphabet is now finite and we can reduce the non-emptiness of FSM to the non-emptiness problem of ν-A.

▶ **Lemma 32.** *The non-emptiness problem for ν-A is in PSPACE*

Proof. Given a ν-A $A = (Q, q_0, F, \Delta, V, M_0)$, its canonical form has at most $|Q|2^{|K|}$ configurations. The state space of *cano*(A) could be constructed as an FSM by merging all transitions of A writing a token from T going from state q to q' into a unique transition. This way, the FSM would have $O(|\Delta|2^{|K|})$ transitions as each configuration (q, M') of *cano*(A) has at most as many outgoing transitions as q in A. A formal definition of the construction is in [3].

Moreover, if the underlying FSM is non-empty it implies that there is a sequence of at most $O(|\Delta|2^{|K|})$ transitions from an initial state of A to one of its accepting state. Recall that finding a path between two vertexes/states in a graph (V, E) is a problem called PATH which is NL-complete [1]. The algorithm for PATH, in logarithmic space, could be adapted to find whether there exists a sequence of transitions from an initial state of A to an accepting state. Since this sequence of transitions is exponential in the size of A, we prove that the problem is in NPSPACE for ν-A. Since PSPACE=NPSPACE [1] we show that the non-emptiness problem for ν-A is in PSPACE.

The PATH algorithm adapted to our problem memorizes a state of A, the memory context of $cano(A)$ and a counter on $O(\log(|\Delta|) + |K|)$ bits. Each time that the counter is augmented by one, a transition starting in the memorized state will be chosen randomly and applied as follows: if this transition is a reset, then the state is updated and the memory is reset. If this transition is a write, then the state is updated and the corresponding key is added to the memory (if not already present). If the transition is a read then either the key is not in the memory and the algorithm halts and rejects or the state is updated. As soon as an accepting state is reached then the algorithm halts and accepts. If the counter reaches its maximum then the algorithm halts and rejects. Note that the FSM is not actually constructed in this algorithm, but only one of its paths is explored dynamically. ◀

▶ **Theorem 33.** *The non-emptiness problem for ν-A PSPACE-complete.*

Proof. By Lemmata 29 and 32. ◀

5 Conclusions

We have discussed the complexity of membership and non-emptiness for three formalisms ν-A, LaMA and HRA. We showed that concerning the membership problem, all three kinds of automata fall in the NP-complete class. Non-emptiness is more delicate. We proved that the non-emptiness problem for ν-A is PSPACE-complete.

For LaMA, we know the lower bound and the upper bound of the complexity class of the non-emptiness problem. As a consequence of Theorem 33 and from the expressiveness results in [4], the complexity is PSPACE-hard. However, it is a strict lower bound as we are able to construct a LaMA where the shortest accepted word is of size in $O(2^{2^n})$ with n the number of variables. In our previous work [4], we showed an exponential encoding of LaMA into HRA for which the non-emptiness problem is shown to be Ackermann-complete in [11]. This also gives us the Ackermann class membership. As one of the reviewers suggested, we believe that we could adapt to LaMA the proof in [11] to show that the problem is actually Ackermann-complete.

As for future work, apart from formally showing the Ackermann-completeness of the non-emptiness problem for LaMA, we plan to address other expressiveness issues of LaMA. Indeed the number of layers seems to create a hierarchy of expressiveness and complexity.

References

1 S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 2006. URL: `https://theory.cs.princeton.edu/complexity/book.pdf`.

2 Clement Bertrand. *Reconnaissance de motifs dynamiques par automates temporisés à mémoire.* Theses, Université Paris-Saclay, December 2020. URL: `https://theses.hal.science/tel-03172600`.

3 Clément Bertrand, Cinzia Di Giusto, Hanna Klaudel, and Damien Regnault. Complexity of Membership and Non-Emptiness Problems in Unbounded Memory Automata. Technical report, Université Côte d'Azur, CNRS, I3S, France ; IBISC, Univ. Evry, Université Paris-Saclay, France ; Scalian Digital Systems, Valbonne, France, 2023. URL: `https://hal.science/hal-04155339`.

4 Clément Bertrand, Hanna Klaudel, and Frédéric Peschanski. Layered memory automata: Recognizers for quasi-regular languages with unbounded memory. In Luca Bernardinello and Laure Petrucci, editors, *Application and Theory of Petri Nets and Concurrency - 43rd International Conference, PETRI NETS 2022, Bergen, Norway, June 19-24, 2022, Proceedings*, volume 13288 of *Lecture Notes in Computer Science*, pages 43–63. Springer, 2022. `doi:10.1007/978-3-031-06653-5_3`.

5 Clément Bertrand, Frédéric Peschanski, Hanna Klaudel, and Matthieu Latapy. Pattern matching in link streams: Timed-automata with finite memory. *Sci. Ann. Comput. Sci.*, 28(2):161–198, 2018. URL: `http://www.info.uaic.ro/bin/Annals/Article?v=XXVIII2&a=1`.

6 Henrik Björklund and Thomas Schwentick. On notions of regularity for data languages. *Theoretical Computer Science*, 411(4):702–715, 2010. Fundamentals of Computation Theory. `doi:10.1016/j.tcs.2009.10.009`.

7 Mikolaj Bojanczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4):27:1–27:26, 2011. `doi:10.1145/1970398.1970403`.

8 Aurélien Deharbe. *Analyse de ressources pour les systèmes concurrents dynamiques*. PhD thesis, Université Pierre et Marie Curie, France, September 2016. URL: `https://tel.archives-ouvertes.fr/tel-01523979`.

9 Aurelien Deharbe and Frédéric Peschanski. 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, June 23-27, 2014*, pages 102–111. IEEE Computer Society, 2014. `doi:10.1109/ACSD.2014.18`.

10 Stéphane Demri and Ranko Lazić. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Logic*, 10(3), April 2009. `doi:10.1145/1507244.1507246`.

11 Radu Grigore and Nikos Tzevelekos. History-register automata. *Log. Methods Comput. Sci.*, 12(1), 2016. `doi:10.2168/LMCS-12(1:7)2016`.

12 Orna Grumberg, Orna Kupferman, and Sarai Sheinvald. Variable automata over infinite alphabets. In Adrian-Horia Dediu, Henning Fernau, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications, 4th International Conference, LATA 2010, Trier, Germany, May 24-28, 2010. Proceedings*, volume 6031 of *Lecture Notes in Computer Science*, pages 561–572. Springer, 2010. `doi:10.1007/978-3-642-13089-2_47`.

13 Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994. `doi:10.1016/0304-3975(94)90242-9`.

14 Michael Kaminski and Daniel Zeitlin. Finite-memory automata with non-deterministic reassignment. *Int. J. Found. Comput. Sci.*, 21(5):741–760, 2010. `doi:10.1142/S0129054110007532`.

15 Ahmet Kara. *Logics on data words: Expressivity, satisfiability, model checking*. PhD thesis, Technical University of Dortmund, Germany, 2016. URL: `http://hdl.handle.net/2003/35216`.

16 Andrzej S. Murawski, Steven J. Ramsay, and Nikos Tzevelekos. Polynomial-time equivalence testing for deterministic fresh-register automata. In Igor Potapov, Paul G. Spirakis, and James Worrell, editors, *43rd International Symposium on Mathematical Foundations of Computer Science, MFCS 2018, August 27-31, 2018, Liverpool, UK*, volume 117 of *LIPIcs*, pages 72:1–72:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. `doi:10.4230/LIPIcs.MFCS.2018.72`.

17 Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3):403–435, July 2004. `doi:10.1145/1013560.1013562`.

18 Hiroshi Sakamoto and Daisuke Ikeda. Intractability of decision problems for finite-memory automata. *Theoretical Computer Science*, 231(2):297–308, 2000. doi:10.1016/S0304-3975(99)00105-X.

19 Nikos Tzevelekos. Fresh-register automata. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 295–306. ACM, 2011. doi:10.1145/1926385.1926420.