Acta Informatica (2017) 54:399–433
DOI 10.1007/s00236-016-0261-6

ORIGINAL ARTICLE

Regular and context-free nominal traces

Pierpaolo Degano[1] · **Gian-Luigi Ferrari**[1] · **Gianluca Mezzetti**[1,2]

Received: 13 February 2014 / Accepted: 27 January 2016 / Published online: 23 February 2016

Abstract Two kinds of automata are presented, for recognising new classes of regular and context-free nominal languages. We compare their expressive power with analogous proposals in the literature, showing that they express novel classes of languages. Although many properties of classical languages hold no longer in the nominal case, we design a slight restriction of our models that preserve some interesting ones. In particular, we prove the emptiness problem decidable and we construct the intersection between (restricted) regular and context-free automata. By examples and walking through their properties we argue the relevance of our models in the context of the verification of resource usage patterns.

1 Introduction

Describing resources and reasoning about their usages is a difficult task that emerges in all scenarios where effective mechanisms are strongly required to control how possibly unboundedly many computational resources are granted to multiple and heterogeneous entities. A paradigmatic example are Cloud systems [24], that support network access to a shared pool of dynamically configurable computing resources with elastic usage requirements. Similarly, in the so-called Internet of Things [23], mobile devices are capable to dynamically discover, acquire and interact with multiple heterogeneous resources. The ability of dealing

This work has been partially supported by the MIUR PRIN project *Security Horizons*.

✉ Gian-Luigi Ferrari
giangi@di.unipi.it

Pierpaolo Degano
degano@di.unipi.it

Gianluca Mezzetti
mezzetti@cs.au.dk

1 Dipartimento di Informatica, Università di Pisa, Pisa, Italy

2 Present Address: Department of Computer Science, Aarhus University, Aarhus, Denmark

with unboundedly many resources also underpins the manipulation of XML schemata [39]; the orchestration of Web-services [41]; the development of security protocols (e.g. nonces and time-stamps) [1,20].

Abstract models are needed for describing the behaviour of programs involving a potentially unbounded number of computational resources, for detecting bugs and verifying properties of their usage. In this paper we will contribute to the definition of an abstract model based on languages over *infinite alphabets*, aka *nominal* languages. In particular, we will propose two new kind of automata that recognise a class of regular and of context-free languages with an infinite alphabet, through which one can abstractly represent several complex patterns of resource usages, as illustrated below.

In the context of programming languages, the term *resource* refers to software or operating system entities, such as graphic devices, file handles, network and database connections, as well as concurrent objects like locks. The following recursive function `updateFiles`, written in an F#-like syntax, abstractly implements a resource usage pattern, typical of many applications, e.g. for updating and synchronizing a storage service available on the cloud—this programming pattern is known in the programming language community under the name of *Resource Acquisition Is Initialization*[1] idiom. Intuitively, the code below monitors the changes to a certain number of files of interest. We omit the actual operations on the resources in order to focus on the control flow pattern followed for acquiring and releasing them.

In reaction to a change notification (here encoded in the `info` parameter) the routine updates the files if certain conditions are met (represented by the `updatable` condition). We assume that the metadata of the files of interest are stored into a suitable data structure, i.e. a *collection*, and that an iterator (`itCollection`) steps through the collection of files. Notice that the routine uses a *fresh* file `r` at each iteration. The `use` construct defines the scope of a resource usage: it binds similarly to `let`, but it additionally take care of disposing the resource when the it leaves its scope.

```
let rec updateFiles(itCollection, info) =
  if (itCollection.hasNext) {
   use r = new file(itCollection.next())
       if (updatable(r,info)){
            ...// updating file with info
        updateFile(itCollection, info))
        }
  // dispose file resource implictly called here
  }
  ...
```

Since we are only interested in how the routine above operates over its resources, we can abstractly represent a computation by only recording resource activations and disposals (within the scope). Intuitively, a *run-time monitor* can extract information from a run and represent it as the following trace

$$\mathtt{new(r1)new(r2)}\ldots\mathtt{new(rn)\ rel(rn)}\ldots\mathtt{rel(r2)rel(r1)}$$

where $r1, r2, \ldots, rn$ correspond to the fresh files created by `new` and disposed by `rel`. In the trace, the files appear following a pattern that has the same structure of the words in the

1 http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization.

context-free language $\{ww^R\}$, where w^R stands for the reverse of w. Additionally, the file resources $r1, r2, \ldots, rn$ are pairwise distinct because of the freshness constraint (granted by the invocations to `new`), and because resource disposal adheres to the scope imposed by the `use` construct. Note also that there are unboundedly many fresh file resources because the value of n depends on the actual size of the collection which is unknown *a priori*; we call this property *unbound freshness*.

It is worth noting that calculi for concurrent and distributed systems faced the similar problem of handling unboundedly many fresh (or restricted) names, in the development of the so called *nominal* calculi [12,25,31,36,44].

We now consider another aspect of resource usage, on a simple Java-like programming example. Consider a basic service for managing resources with the `try-with-resources`[2] statement that declares one or more resources, and that ensures each declared resource to be closed (disposed or released in our terms) at the end of the statement. Recall that only Java objects that implement the interface `java.lang.AutoCloseable` are resources.

```
public class MyObjectService implements AutoCloseable {
    private String metaData;

    public void doIt() {
        System.out.println("MyObjectService is doing it!");
        metaData = // ?
    }

    public String getInfo() {
        return metaData;
    }

    @Override
    public void close() throws Exception {
        System.out.println("MyObjectService is closed!");
        // dispose of resource implictly called here
    }
}

// Client code
String metaDataInfo = "suitable info";

try(MyObjectService myObj = new MyObjectService()){
  myObj.doIt();
  //actions on myObj
  metaDataInfo = myObj.getInfo();
}

System.out.println(metaDataInfo); // use of metaDataInfo
```

The client is granted access to all the services associated with the objects including the `getInfo()` method that handles the service metadata. After the invocations of `doIt`, the

[2] https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html.

client gets and stores inside a local object (here `String`) a (meta) property of the whole object, i.e. of the whole resource. When `try-with-resources` is completed the object resource is disposed. However, the client can still access and operate with the metadata of the resource. The following trace abstractly represents the creation of fresh resources and access to any of their public parts in a run.

```
new(myObj)access(myObj)access(myObj)access(myObj)rel(myObj)
access(metaDataInfo)
```

The last action represents what we call a *late* access to `myObj`, performed through the last statement on `metaDataInfo`. Through such a late access, the client can acquire possibly confidential (meta)data, leading to an unwanted information flow from the service to the client. Expressing and detecting at a suitable abstraction level the correct flow of information requires then the ability of tracking in exact terms how and *when* resources are accessed.

Although the Java program above may appear artificial, it is an instance of the well-known programming pattern for Object Oriented programming, called *object-pool*,[3] where the object kept is still accessible after its release. Note that nominal techniques are needed because pools are not necessarily finite. A paradigmatic example is the *Cached Thread Pools* of Java,[4] which can spawn an unbound number of threads, while reusing the ones that have been released. Late usage of resources also occurs in the composition of web services [21].

The examples above and the programming pragmatics show the importance of devising expressive and flexible abstractions to control the usage of resources. The problem becomes crucial by the current programming practice, where it is common to pick from the Web some scripts, or plugins, or code snippets, and assemble them into a bigger program, with little or no control about the correctness (e.g. security) of the whole program. Verifying properties may be cumbersome even for small programs, and it may also lead to rather complex checking machineries. Correctness properties are typically expressed in terms of formalisms which predicate over execution traces of, e.g., finite state machines, regular expressions, context-free grammars, linear temporal logics, and their extensions. The encoding of resource usage properties like unbound freshness and late usage discussed above cannot be directly expressed in standard specification formalisms, because it is crucial to deal with freshness of resources and their scope.

The literature has many proposals for statically abstracting program behaviour, as done in the simple traces given above (see [6,9,16,40,46], just to cite a few). Along this line, two of the authors of this paper introduced a framework to statically compute sound over-approximations of the possible run-time traces of a program [3–6]. This approximations are then model-checked [8] to guarantee that prescribed policies will hold at runtime. The over-approximations can be seen as context-free languages, and the present work aims at extending that approach also to nominal languages [34].

In summary, the contribution of this paper are the following. We introduce an abstract model for modelling resource usage that takes care of (i) expressing context-free behaviour, as well as regular ones (ii) handling unboundedly many resources, with (iii) unbound freshness, and (iv) late usage of resources.

We express context-free traces of resource usage through *Pushdown Nominal Automata* (PDNA$_\pm$), that extend classical pushdown automata. The alphabet of PDNA$_\pm$ is infinite, so we can undertake item (ii) above in the style of nominal techniques [22].

3 http://www.oodesign.com/object-pool-pattern.html.

4 http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html#newCachedThreadPool--.

To guarantee freshness of resources (i.e. of the symbols of the alphabet), the PDNA$_\pm$ exploit (finitely many) additional structures, called *m-registers*, that store resources in a stack-like fashion. A resource is fresh if no m-register contains it. Since m-registers have unbounded capacity, unbounded freshness, as stated in (iii), is guaranteed. As expected, when a resource is released, it is removed from the m-register that stores it, and so it can be re-used as fresh later on. Finally, we grant late usage allowing a resource to be mentioned after it has been disposed, as in item (iv). The stack of PDNA$_\pm$ plays a crucial role in this mechanism: a resource in use, i.e. occurring in a m-register, say N, can be saved in the stack and can remain there after it has been removed from N, ready to be recalled once occurring again on top of the stack. In the over-simplified example above, the metadata will be recorded in a m-register N, and a mention to it pushed on the stack. After disposing the object, one can still access its metadata through the hook stored in the stack.

Here we also investigate the problem of handling unboundedly many resources and of unbounded freshness in regular languages. To do that, we introduce the new model of *Finite State Nominal Automata* (FSNA$_\pm$), that are finite state automata enriched with a finite number of m-registers.

We then prove that the languages recognised by both FSNA$_\pm$ and PDNA$_\pm$ are closed under union, and the first ones are also closed under intersection, provided that symbols are not released. The intersection of a language accepted by a FSNA$_\pm$ with that of a PDNA$_\pm$ is recognised by a PDNA$_\pm$, provided that neither automata release resources. Neither the FSNA$_\pm$ and PDNA$_\pm$ languages, instead, are closed under complement.

We also establish the decidability of the emptiness for FSNA$_\pm$ and for the subclass of PDNA$_\pm$ in which symbols are not released. Consequently, it is feasible to model-check a property expressed as a (restricted) FSNA$_\pm$ against a model expressed as a (restricted) PDNA$_\pm$, by verifying the emptiness of their intersection, in the style of [49].

We also compare the expressiveness and some properties of our models with other proposals in the literature, that will be briefly surveyed. In particular, we consider the regular languages over infinite alphabet and their recognisers investigated in [7,14,18,27,29,31,33,47,48], as well as the context-free languages over infinite alphabet of [7,12,15,17,39,42,43].

The paper is organised as follows. Notation and abbreviations are summarised in Table 1. In Sect. 2 we introduce FSNA$_\pm$; we study their language theoretical properties in Sect. 3; and we compare their expressiveness and their properties with those of other models in the literature in Sect. 4. Section 5 introduces PDNA$_\pm$; investigates their properties in Sect. 6; and compares them with other proposals in Sect. 7.

2 Finite State Nominal Automata

As anticipated in the introduction, our automata abstractly model traces of software systems, with the focus on the pattern they follow when manipulating resources. A common pattern is that programs can only request a fresh resource, and not a specific one: think of object references, server mirrors, nonces, etc. Resources are often manipulated opaquely: the program is only allowed to test them for equality. Hence, fresh resources are interchangeable: the set of traces that a program can generate does not depend on the specific identity of the resources involved, but only on their relative equality and inequality. In the literature this property is called *language equivariance* [12].

To keep our presentation simple, we almost always consider languages over resources, disregarding actions on them. Occasionally, we shall consider actions on resources as well

Table 1 Notation

Math	
$i, j \in \underline{r} = \{i \mid 1 \leq i \leq r\}$	Set of indices in $\mathbb{N}$, the natural numbers
$L \lesseqgtr L', \lvert L \rvert$	Incomparable sets: $L \not\subseteq L'$ and $L \not\supseteq L'$, cardinality of L
$img(f)$	The image of the function f
Words	
$\Sigma = \Sigma_s \cup \Sigma_d, \Sigma_s \cap \Sigma_d = \emptyset \;\; (\{?, \top\} \cup \mathbb{N}) \cap \Sigma = \emptyset$	Alphabet with Σ_s *finite* set of *static* symbols and Σ_d *countably infinite* set of *dynamic* symbols
$a, b \in \Sigma;\;\; w \in \Sigma^*$	Symbols in Σ and words, where ε is the empty string
$w[i], \lvert w \rvert, w^R, \lVert w \rVert$	i-th symbol, length, reverse, and set of symbols of w
Automata	
$q \in Q$	State of an automaton
$\sigma \in \Sigma_s \cup \underline{r} \cup \{\varepsilon, \top\}$	Input symbol in a transition label
$Z \in \Sigma_s \cup \underline{r} \cup \{\varepsilon, ?\}$	Stack read symbol in a transition label
$\zeta \in (\Sigma_s \cup \underline{r})^*;\; z \in \Sigma_s \cup \underline{r}$	Stack write symbols in a transition label
$\Delta \in \{i+, i- \mid i \in \underline{r}\} \cup \{\varepsilon\}$	m-registers update in a transition label
$S, \lvert S \rvert$	A stack, its height, $\boxtimes$ is the empty stack
$r \in \mathbb{N}$	Number of registers
N, M	m-registers, $\boxtimes$ is the empty m-register
$\lVert N \rVert$	Set of (dynamic) symbols in N
C, ρ	A configuration, and a run $C_1 \to \cdots \to C_k$
$R, A;\;\; L(R), L(A)$	A FSNA$_\pm$, PDNA$_\pm$ automaton and their language
$\mathcal{L}(\text{FSNA}_\pm), \mathcal{L}(\text{VFA}), \ldots$	Set of languages accepted by FSNA$_\pm$, VFA, ... automata

(known as data-words [11] and briefly surveyed before Example 2), to illustrate the minor changes needed to deal with them (see e.g. Fig. 2). To account for freshness, resources are abstractly represented by symbols of an infinite alphabet $a \in \Sigma$, that we assume partitioned in a finite set of *static* symbols Σ_s and an infinite set of *dynamic* symbols Σ_d. The first is intended to represent the finite amount of resources known *before* program execution, while Σ_d contains the resources that may be *acquired or generated at run-time*.

Our automata use special data structures to record the dynamic symbols appearing in a recognised string, called *mindful registers* (*m-registers* for short). An m-register N is actually a stack S of symbols in Σ_d and an *activation state* ($x \in \{1, 0\}$). An empty stack makes the m-register *empty*, as well, and we denote it by $\boxtimes$. When the tag x is 1 then the m-register is *active*, otherwise the m-register is *inactive*. The operations on an m-register N are built on the standard `push`, `top` and `pop` operations as follows:

$$\text{s-push}(a, \langle x, S\rangle) = \langle 1, \texttt{push}(a, S)\rangle$$

$$\text{s-top}(\langle 1, S\rangle) = \texttt{top}(S)$$

$$\text{s-pop}(\langle x, S\rangle) = \begin{cases} \langle 0, \texttt{pop}(S)\rangle \text{ if } S \neq \boxtimes \\ \langle 0, S\rangle \text{ if } S = \boxtimes \end{cases}$$

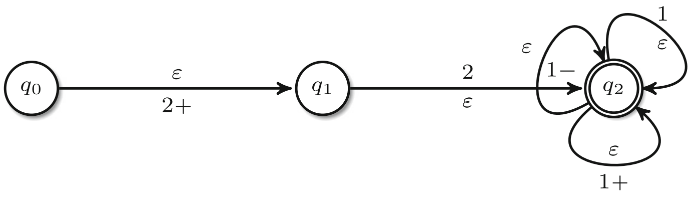


Fig. 1 The FSNA$_\pm$ R_0 accepting the language $\{aw \mid a \in \Sigma_d, w \in \Sigma_d^*, a \notin \|w\|\}$

An s-push operation makes an m-register active, regardless of its activation state. The operation s-top yields a value only if the m-register is active and not empty, otherwise it is undefined. Finally, after a s-pop, the m-register N becomes inactive. Note that s-popping an empty m-register results in a no-operation, so making it impossible to discern an inactive m-register from an empty one.

A symbol is *fresh* with respect to an m-register when it does not appear in its stack.

Before formally defining *Finite State Nominal Automata* (FSNA$_\pm$ for short) we intuitively illustrate their recognising mechanisms through the automaton R_0 in Fig. 1. A run on R_0 recognising a word w is a sequence of configurations leading from its initial state q_0 to its final state q_2. We assume that R_0 has two m-registers that will be part of configurations. In the initial configuration the two m-registers are empty and we render them as $[\boxtimes, \boxtimes]$.

The leftmost edge is $q_0 \xrightarrow[2+]{\varepsilon} q_1$, and following it the automaton reads no symbol (recorded by ε) and goes from the configuration $\langle q_0, [\boxtimes, \boxtimes]\rangle$ to a configuration of the form $\langle q_1, [\boxtimes, \boxed{\text{a}}]\rangle$ where a symbol $a \in \Sigma_d$ is s-pushed in the m-register number 2, as dictated by the label $2+$, provided a is fresh w.r.t. both the m-registers of R_0. By using the edge $q_1 \xrightarrow[\varepsilon]{2} q_2$ the automaton reaches the configuration $\langle q_2, [\boxtimes, \boxed{\text{a}}]\rangle$ and reads a, i.e. the s-top symbol of the m-register number 2, while nothing is done on the m-registers because of the label ε.

There are three edges looping in state q_2. The edge $q_2 \xrightarrow[1+]{\varepsilon} q_2$ s-pushes a *fresh* symbol in the m-register number 1 and reads no symbol; $q_2 \xrightarrow[\varepsilon]{1} q_2$ recognises the s-top symbol of m-register number 1 and leaves the m-registers untouched. Slightly differently, the edge labelled $q_2 \xrightarrow[1-]{\varepsilon} q_2$ s-pops a symbol from the m-register number 1 (because the label is $1-$) and recognises no symbol. After following it the m-register number 1 becomes inactive and the edge $q_2 \xrightarrow[\varepsilon]{1} q_2$ can not be followed.

A run on R_0 is $\langle q_0, abc, [\boxtimes, \boxtimes]\rangle \xrightarrow{\varepsilon} \langle q_1, abc, [\boxtimes, \boxed{\text{a}}]\rangle \xrightarrow{a} \langle q_2, bc, [\boxtimes, \boxed{\text{a}}]\rangle \xrightarrow{\varepsilon} \langle q_2, bc, [\boxed{\text{b}}, \boxed{\text{a}}]\rangle \xrightarrow{b} \langle q_2, c, [\boxed{\text{b}}, \boxed{\text{a}}]\rangle \xrightarrow{\varepsilon} \langle q_2, c, [\boxtimes, \boxed{\text{a}}]\rangle \xrightarrow{\varepsilon} \langle q_2, c, [\boxed{\text{c}}, \boxed{\text{a}}]\rangle \xrightarrow{c} \langle q_2, \varepsilon, [\boxed{\text{c}}, \boxed{\text{a}}]\rangle$.

The reader may convince himself that the language recognised by R_0 is $\{aw \mid a \in \Sigma_d, w \in \Sigma_d^*, a \notin ||w||\}$.

In the formal definition and hereafter we use some notation and abbreviations collected in Table 1. We denote the set of the natural numbers by $\mathbb{N}$, $\underline{k}$ is the segment of the natural numbers $\{i \mid 1 \leq i \leq k\}$, w is a word in Σ^* with length $|w|$ and i-th symbol $w[i]$, $\|w\|$ denotes the set of symbols used in w, ε is the empty word.

Definition 1 (*Finite State Nominal Automata*)
A *finite state nominal automaton* (FSNA$_\pm$) is the tuple $R = \langle Q, q_0, \Sigma, \delta, r, F\rangle$ where:

– Q is a finite set of states, $q, q_1, q', \ldots \in Q$

- $q_0 \in Q$ is the initial state
- $\Sigma = \Sigma_s \cup \Sigma_d$ is the infinite alphabet (Σ_s is finite, Σ_d denumerable, $\Sigma_s \cap \Sigma_d = \emptyset$)
- $r \in \mathbb{N}$ is the number *m-registers*
- δ is the transition relation: $(q, \sigma, q', \Delta) \in \delta$ with $\sigma \in \Sigma_s \cup \underline{r} \cup \{\varepsilon\}$, $\Delta \in \{i+, i- \mid i \in \underline{r}\} \cup \{\varepsilon\}$
  We call a transition *new* when $\Delta = i+$; *delete* when $\Delta = i-$; *update* when $\Delta \neq \varepsilon$.
  For brevity, we write $q \xrightarrow[\Delta]{\sigma} q' \in \delta$ whenever $(q, \sigma, q', \Delta) \in \delta$
- $F \subseteq Q$ is the set of final states

A *configuration* is a tuple $C = \langle q, w, [N_1, \ldots, N_r]\rangle$ where q is the current state, $w \in \Sigma^*$ is the word to be recognised and $[N_1, \ldots, N_r]$ is an array of r *m-registers* with symbols in Σ_d. The configurations $\langle q_f \in F, \varepsilon, [N_1, \ldots, N_r]\rangle$ are *final.*

The application of a transition is detailed by the following definition:

Definition 2 *(Recognising step)* Given an FSNA$_\pm$ R, a step $\langle q, w, [N_1, \ldots, N_r]\rangle \rightarrow \langle q', w', [N'_1, \ldots, N'_r]\rangle$ occurs if and only if there exists a transition $q \xrightarrow[\Delta]{\sigma} q' \in \delta$ such that both conditions hold:

1. $$\begin{cases} \sigma = \varepsilon & \Rightarrow w = w' \quad \text{and} \\ \sigma = i & \Rightarrow w = \text{s-top}(N_i)w' \quad \text{and} \\ \sigma \in \Sigma_s & \Rightarrow w = \sigma w' \end{cases}$$
2. $$\begin{cases} \Delta = i+ & \Rightarrow \exists b \in \Sigma_d . N'_i = \text{s-push}(b, N_i) \wedge \forall j . b \notin \|N_j\| \wedge \forall j\ (j \neq i). N_j = N'_j \text{ and} \\ \Delta = i- & \Rightarrow N'_i = \text{s-pop}(N_i) \quad \wedge \quad \forall j\ (j \neq i). N_j = N'_j \quad \text{and} \\ \Delta = \varepsilon & \Rightarrow \forall j . N_j = N'_j \end{cases}$$

Finally, the language accepted by an FSNA$_\pm$ R, which we call *(nominal) regular*, is

$$L(R) = \{w \in \Sigma^* \mid \exists \rho : C_1 = \langle q_0, w, [\boxtimes, \ldots, \boxtimes]\rangle \rightarrow^* C_k, \text{with } C_k \text{ final}\}$$

where $\rightarrow^*$ denotes the reflexive and transitive closure of the $\rightarrow$ relation.

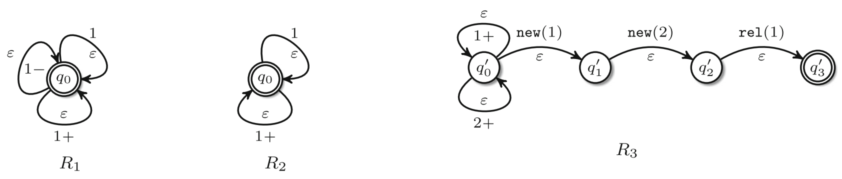
A couple of examples follow.

Example 1 The FSNA$_\pm$ R_1 in Fig. 2 recognises Σ^*. The run ρ_1 recognises the word aax, where x can be any symbol in Σ, even a itself, because the m-registers are empty when a fresh symbol is required by the edge labeled $1+$.

By removing the edge $q_0 \xrightarrow[1-]{\varepsilon} q_0$ from R_1 we obtain the automaton R_2 in Fig. 2. Without that deletion edge, there is no way to forget a symbol from the m-registers. Hence all the issued symbols are recorded in the m-registers stack, and when a new symbol is s-pushed it must be fresh with respect to all of them. The language accepted by the FSNA$_\pm$ R_2 is $L_0 = \{w \in \Sigma_d^* \mid \forall i, j\ (i \neq j).\ w[i] \neq w[j]\}$. The run ρ_2 recognises the string abc.

The next example considers traces of data-words [38]. A data-word is a finite sequence of positions each having a label which either represents an action and is taken from a finite alphabet, or data and is taken from an infinite alphabet. We feel free to use data-words to ease the development of our examples, indeed only minor variations of our automata definitions are required to handle a finite number of actions acting on both static and dynamic resources.

Example 2 Consider again the first example in the introduction showing the recursive function `updateFiles`. In the abstraction of a run, we actually considered traces in the form



$$\rho_1 = \langle q_0, aax, [\boxtimes]\rangle \xrightarrow{\varepsilon} \langle q_0, aax, \llbracket\boxed{\text{a}}\rrbracket\rangle \xrightarrow{a} \langle q_0, ax, \llbracket\boxed{\text{a}}\rrbracket\rangle \xrightarrow{a} \langle q_0, x, \llbracket\boxed{\text{a}}\rrbracket\rangle \xrightarrow{\epsilon} \langle q_0, x, [\boxtimes]\rangle \xrightarrow{\varepsilon} \langle q_0, x, \llbracket\boxed{\text{x}}\rrbracket\rangle \xrightarrow{x} \langle q_0, \varepsilon, \llbracket\boxed{\text{x}}\rrbracket\rangle$$

$$\rho_2 = \langle q_0, abc, [\boxtimes]\rangle \xrightarrow{\varepsilon} \langle q_0, abc, \llbracket\boxed{\text{a}}\rrbracket\rangle \xrightarrow{a} \langle q_0, bc, \llbracket\boxed{\text{a}}\rrbracket\rangle \xrightarrow{\varepsilon} \langle q_0, bc, \llbracket\boxed{\begin{matrix}\text{b}\\ \text{a}\end{matrix}}\rrbracket\rangle \xrightarrow{b} \langle q_0, c, \llbracket\boxed{\begin{matrix}\text{b}\\ \text{a}\end{matrix}}\rrbracket\rangle \xrightarrow{\varepsilon}$$
$$\langle q_0, c, \llbracket\boxed{\begin{matrix}\text{c}\\ \text{b}\\ \text{a}\end{matrix}}\rrbracket\rangle \xrightarrow{c} \langle q_0, \varepsilon, \llbracket\boxed{\begin{matrix}\text{c}\\ \text{b}\\ \text{a}\end{matrix}}\rrbracket\rangle$$

$$\rho_3 = \langle q'_0, \mathtt{new}(a)\mathtt{new}(b)\mathtt{rel}(a), [\boxtimes, \boxtimes]\rangle \xrightarrow{\varepsilon} \langle q'_0, \mathtt{new}(a)\mathtt{new}(b)\mathtt{rel}(a), \llbracket\boxed{\mathtt{new(a)}}, \boxtimes\rrbracket\rangle \xrightarrow{\varepsilon}$$
$$\langle q'_0, \mathtt{new}(a)\mathtt{new}(b)\mathtt{rel}(a), \llbracket\boxed{\mathtt{new(a)}}, \boxed{\mathtt{new(b)}}\rrbracket\rangle \xrightarrow{\mathtt{new}(a)} \langle q'_1, \mathtt{new}(b)\mathtt{rel}(a), \llbracket\boxed{\mathtt{new(a)}}, \boxed{\mathtt{new(b)}}\rrbracket\rangle \xrightarrow{\mathtt{new}(b)}$$
$$\langle q'_2, \mathtt{rel}(a), \llbracket\boxed{\mathtt{new(a)}}, \boxed{\mathtt{new(b)}}\rrbracket\rangle \xrightarrow{\mathtt{rel}(a)} \langle q'_3, \varepsilon, \llbracket\boxed{\mathtt{new(a)}}, \boxed{\mathtt{new(b)}}\rrbracket\rangle$$

Fig. 2 Three examples of FSNA$_\pm$ R_i and of their runs ρ_i. The automaton R_1 accepts Σ^*; note that the dynamic symbol x can be any symbol in Σ_d, even a, because the m-register is empty when x is s-pushed and there is no restriction on its freshness. The automaton R_2 accepts L_0 in Example 1; and R_3 accepts strings $\mathtt{new}(a)\mathtt{new}(b)\mathtt{rel}(a)$ ($a \neq b$)

of data-words, where the actions `new` and `rel` operate on files. The FSNA$_\pm$ R_3 in Fig. 2 accepts the unwanted traces where the resource that was picked second-last (`new(1)`) is released (`rel(1)`) before the release of the resource that was picked last (`new(2)`). Note that the (data-word) symbols σ assume the form `new(u)`, `rel(u)`, $u \in \underline{2} \cup \Sigma_s$.

Note that the constraint on s-pop to deactivate the affected m-register limits the expressiveness of FSNA$_\pm$. Indeed, by relaxing that condition, we would be able to count on each m-register, by performing a *new* transition to increment, a *delete* to decrement and using the active/inactive status of m-registers to discern whether it is zero or not. These mechanisms are very similar to the ones of r-counter machines [35].

We introduce now a sub-class of FSNA$_\pm$ where no delete transitions are allowed, as it is the case for the automaton R_2 in Fig. 2. As expected, this restriction reduces the expressiveness of FSNA$_\pm$, e.g. none of this restricted automata can accept Σ^*.

Definition 3 (FSNA$_+$) An FSNA$_+$ is a FSNA$_\pm$ with no delete transition $q \xrightarrow[i-]{\sigma} q'$.

In the statement below $\mathcal{L}$(FSNA$_+$), $\mathcal{L}$(FSNA$_\pm$) denote the classes of language of FSNA$_+$, FSNA$_\pm$, respectively.

Property 1 $\mathcal{L}(FSNA_+) \subsetneq \mathcal{L}(FSNA_\pm)$

Proof By contradiction, assume there exists a FSNA$_+$ with r m-registers that accepts Σ^*. Consider a string of the form ww with $|w| = |\|w\|| = r + 1$ and let $b \in \|w\|$ be such that $b \neq$ s-top(N_i) after w has been scanned; note that such a b always exists. However, ww can not be accepted because b still occurs in one of the r m-registers and thus can not be s-pushed again. □

We present now a variation of the FSNA$_+$s which can update more than one m-register in a single transition. This variation will be useful in the proof of Theorem 6, as expected, this parallelization does not extend the expressiveness of the FSNA$_\pm$. For simplicity, we permit below to update two m-registers only, the extension to any finite number being straightforward.

Definition 4 *(Finite Nominal Automata 2)* A *finite state nominal automaton 2* (FSNA$_+$2) is a tuple $R = \langle Q, q_0, \Sigma, \delta 2, r, F\rangle$ where:

- Q, q_0, Σ, r and F are as in Definition 1
- $\delta 2$ is the transition relation: $(q, \sigma, q', (\Delta_1, \Delta_2)) \in \delta 2$ with $q, q' \in Q, \sigma \in \Sigma_s \cup \underline{r} \cup \{\varepsilon\}, \Delta_1, \Delta_2 \in \{i+ \mid i \in \underline{r}\} \cup \{\varepsilon\}$, such that $\forall i \in \underline{r}. (\Delta_1, \Delta_2 \neq i-)$ and $(\Delta_1 = \Delta_2 \Rightarrow \Delta_1 = \Delta_2 = \varepsilon)$

Definition 5 *(Recognising step)*

A step $\langle q, w, [N_1, \ldots, N_r]\rangle \rightarrow \langle q', w', [N'_1, \ldots, N'_r]\rangle$ occurs iff there exists $q \xrightarrow[(\Delta_1,\Delta_2)]{\sigma} q' \in \delta 2$ such that

1. As in Definition 2
2. $$\begin{cases} \forall j \in \underline{r}\ (j \neq \Delta_1, \Delta_2). N_j = N'_j \text{ and} \\ \Delta_1 = i+ \Rightarrow \exists b_1 \in \Sigma_d.N'_i = \text{s-push}(b_1, N_i) \quad \wedge \quad \forall j \in \underline{r}. b_1 \notin \|N_j\| \\ \qquad \wedge \ \ b_1 \neq \text{s-top}(N_i) \quad \text{and} \\ \Delta_2 = i+ \Rightarrow \exists b_2 \in \Sigma_d.N'_i = \text{s-push}(b_2, N_i) \\ \qquad \wedge \ \ \forall j \in \underline{r}. b_2 \notin \|N_j\| \quad \wedge \quad b_2 \neq \text{s-top}(N_i) \end{cases}$$

As anticipated, the FSNA$_+$2 have the same expressive power of FSNA$_+$.

Theorem 1 *Given a FSNA$_+$2 A there exists an FSNA$_+$ A′ accepting the same language.*

Proof Let $A = \langle Q, q_0, \Sigma, \delta 2, r, F\rangle$ and define $A' = \langle Q \cup Q', q_0, \Sigma, \delta, r, F\rangle$ where Q' is a set of fresh states q_p, one for each transition $p \in \delta 2$ and δ is such that $\forall p = q \xrightarrow[(\Delta_1,\Delta_2)]{\sigma} q' \in \delta 2$ we have that $q \xrightarrow[\Delta_1]{\sigma} q_p, q_p \xrightarrow[\Delta_2]{\varepsilon} q' \in \delta$. It is now immediate proving the equality of the accepted languages. □

3 Properties of the FSNA$_\pm$

This section studies some language theoretical properties of the two classes of automata FSNA$_\pm$ and FSNA$_+$ introduced so far. The following property is immediate:

Property 2 *Increasing the number of m-registers increases the expressive power of FSNA$_\pm$.*

Any finite number of m-registers is not sufficient for breaking the barrier between regular and context-free languages, because m-registers are not full-fledged stacks: they become inactive after an s-pop and empty registers can not be distinguished from inactive ones. This is shown by the following “classical” example, showing a Dyck-like language that is not regular.

Example 3 Let $L_r = \{ww^R \in \Sigma_d^* \mid |w| = r \text{ and } \forall i, j\ (i \neq j). w[i] \neq w[j]\}$ then no FSNA$_\pm$ R with less that r states and r m-registers accepts L_r. Indeed, a standard argument on FSA proves that r states are required. Assume now that R has less than r registers N_i and accepts ww^R. By the pigeonhole principle, there is at least a symbol of w, say a, such that $\forall i.a \neq \text{s-top}(N_i)$ when w has been read. Since $a \in \|w\|$, a needs to be s-pushed while traversing w^R, but it is fresh so it can be replaced by any other (fresh) different symbol, which makes R to accept also ww'^R, where $w'^R \neq w^R$: contradiction.

We establish now a few closure properties w.r.t. standard language operations: union ($\cup$), intersection ($\cap$), complementation ($\overline{\ .\ }$), concatenation ($\cdot$) and Kleene star ($*$). To simplify and structure the proofs of these properties we need some auxiliary technical definitions.

In order to support the definition of the intersection automaton R, we will use the standard construction, that builds the product of two automata R_1 and R_2. As usual, a state of R is a pair of states of R_1 and R_2, but it has an additional component: the *merge* function defined below, that describes how the m-registers of the two automata to intersect are mapped into those of the intersection automaton. Given that Definition 6-8 aim to support the proof of the closure by intersection of FSNA$_+$ (and later PDNA$_+$), we assume all the mentioned m-registers to be active.

Definition 6 *(Merge function)* Let $m : \{1, 2\} \times \underline{\mathtt{r}} \to \underline{2\mathtt{r}}$ be a function. Stipulating $m_1(x) = m(1, x)$, $m_2(x) = m(2, x)$, m is a merge iff m_1 and m_2 are injective.

The m-registers i, j are *merged* by m, in symbols $i \overset{\mathrm{m}}{\longleftrightarrow} j$, when $m_1(i) = m_2(j)$, we write $i \overset{\mathrm{m}}{\not\longleftrightarrow} j$ when they are not merged or, by abuse of notation, whenever $i = \varepsilon$ or $j = \varepsilon$.

We stipulate that m extends to a relation between active m-registers such that

$m[N_1^1, \ldots, N_r^1, N_1^2, \ldots, N_r^2] = [M_1, \ldots, M_{2r}]$ iff $\forall i \in \underline{\mathtt{r}}, j \in \underline{2\mathtt{r}}$. N_i^1, N_i^2, M_j are active

and $\forall i, j \in \underline{\mathtt{r}}$

1. s-top$(N_i^1) =$ s-top$(M_{m_1(i)})$ and s-top$(N_i^2) =$ s-top$(M_{m_2(i)})$
2. s-top$(N_i^1) =$ s-top(N_j^2) when $i \overset{\mathrm{m}}{\longleftrightarrow} j$
3. $\bigcup_{i\in\underline{\mathtt{r}}} \|N_i^1\| \cup \bigcup_{i\in\underline{\mathtt{r}}} \|N_i^2\| = \bigcup_{i\in\underline{2\mathtt{r}}} \|M_i\|$

The m_1 (resp. m_2) component of m associates the state of the m-registers of the R_1 automaton (resp. R_2) with the ones of R. As we will show afterwards, the extension of the merge to a relation between m-registers can be used to develop an invariant between the states of the m-registers of R_1, R_2 and the states of the m-registers of R along a recognising run of a word w.

Definition 7 *(Effective Update)* Given two merge functions m, m', the *effective update* $\overset{m}{m'}(\Delta_1, \Delta_2)$ of $\Delta_1, \Delta_2 \in \{i+ \mid i \in \underline{\mathtt{r}}\} \cup \{\varepsilon\}$ is the pair $(\overline{\Delta_1}, \overline{\Delta_2})$ where:

if $\Delta_1 \overset{\mathrm{m}'}{\longleftrightarrow} \Delta_2$ then $\overline{\Delta_1} = m_1'(\Delta_1)$ and $\overline{\Delta_2} = \varepsilon$;

if $\Delta_1 \overset{\mathrm{m}'}{\not\longleftrightarrow} \Delta_2$ then

- $\overline{\Delta_1}$ is such that:
  - if $\Delta_1 = \varepsilon$ then $\overline{\Delta_1} = \varepsilon$
    else if $m_1(\Delta_1) \neq m_1'(\Delta_1)$ then $\overline{\Delta_1} = \varepsilon$ else $\overline{\Delta_1} = m_1'(\Delta_1)$
- $\overline{\Delta_2}$ is such that:
  - if $\Delta_2 = \varepsilon$ then $\overline{\Delta_2} = \varepsilon$
    else if $m_2(\Delta_2) \neq m_2'(\Delta_2)$ then $\overline{\Delta_2} = \varepsilon$ else $\overline{\Delta_2} = m_2'(\Delta_2)$

Definition 8 *(Evolution)* Given a merge m we say that a merge m' is an *evolution* of m with respect to Δ_1, Δ_2, in symbols $m \overset{\Delta_1,\Delta_2}{\curvearrowright} m'$, iff

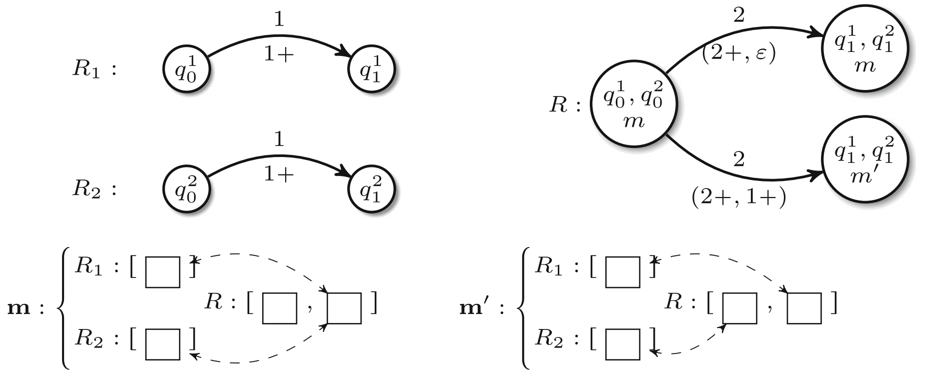


Fig. 3 *R* is a portion of the FSNA$_+$2 automaton recognising the intersection of the languages of R_1 and R_2, the diagrams at the *bottom* represent the merges m, m', where $m_1(1) = 2, m_2(1) = 2, m'_1(1) = 2, m'_2(1) = 1$

1. $\forall i \in \{1, 2\}, j \in \underline{r}\ (j \neq \Delta_1, \Delta_2).\ m_i(j) = m'_i(j)$
2. if $\Delta_1 \overset{m}{\leftrightarrow} j, j \neq \Delta_2$ then $\Delta_1 \overset{m'}{\not\leftrightarrow} j$
3. if $i \overset{m}{\leftrightarrow} \Delta_2, i \neq \Delta_1$ then $i \overset{m'}{\not\leftrightarrow} \Delta_2$

To intuitively illustrate the definitions above, in Fig. 3 we show a portion of the automaton $R = R_1 \cap R_2$ recognising the intersection of the two FSNA$_+$ R_1 and R_2.

The intersection automaton R is obtained by the standard construction that builds the new states as the product of the old ones. Additionally, each pair $\langle q^1, q^2 \rangle$ ($q^1, q^2 \in R_1, R_2$ resp.) is enriched with a *merge* function m. The m describes how the m-registers of the two automata are mapped into those of R

The idea underlying m is to guarantee the following invariant $\mathcal{I}$ along the runs: if R_1 and R_2 are in configurations $\langle q_0^1, w, [\boxed{\text{y}}] \rangle$ and $\langle q_0^2, w, [\boxed{\text{x}}] \rangle$ then *(i)* R will be in configuration $\langle \langle q_0^1, q_0^2, m \rangle, w, [\boxed{\text{h}}, \boxed{\text{z}}] \rangle$ and *(ii)* if two m-registers have the same s-tops then they are merged by m (and vice versa). This is illustrated in the left-most configurations of Fig. 4: if $x = y = a$ then m maps the two registers to one register of R (here the second one), and $z = a$. The edges of the automaton are also defined in the standard way. However, the m-register mentioned in σ of R is the one merged by m, provided that R_1 and R_2 agree on σ. Also the updates $(\overline{\Delta}, \overline{\Delta'})$ in R are determined by the updates Δ_1 of R_1 and Δ_2 of R_2 under the merge m, and form an effective update (see Definition 7).

Consider again Fig. 3. The transition $t : \langle \langle q_0^1, q_0^2 \rangle, m \rangle \xrightarrow[2+]{2} \langle \langle q_1^1, q_1^2 \rangle, m \rangle$ is present because there are $q_0^1 \xrightarrow[1+]{1} q_1^1$ and $q_0^2 \xrightarrow[1+]{1} q_1^2$ and m maps the first m-register of R_1 and that of R_2 to the second of R. Instead, the state $\langle \langle q_0^1, q_0^2 \rangle, m' \rangle$ (omitted in the figure) has no outgoing edges, because the symbols read by R_1 and R_2 are kept apart by m'.

There are transitions that only differ for the merge function in their target state. Not all the possible merges respect however the invariant $\mathcal{I}$ mentioned above. Indeed, we only keep those that are *evolution* (in the sense of Definition 8) of the merge in the source state, according to the updates Δ_1, Δ_2 of R_1, R_2 respectively. For example, the transition $\langle \langle q_0^1, q_0^2 \rangle, m \rangle \xrightarrow[(1+,2+)]{2} \langle \langle q_1^2, q_1^2 \rangle, m' \rangle$ permits the recognising step $C \xrightarrow{a} C'$, where the m-register of R_1 now has got a d, while that of R_2 has got c, and m' keeps them apart.

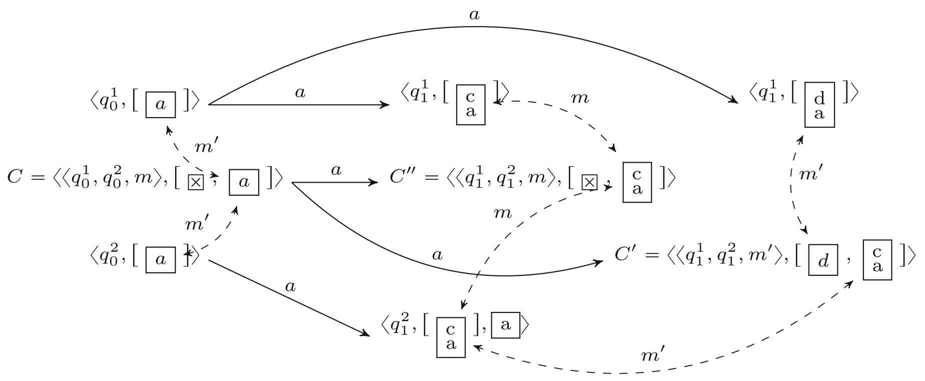

$\langle q_0^1, [\boxed{a}]\rangle$
$\langle q_1^1, [\boxed{c / a}]\rangle$
$\langle q_1^1, [\boxed{d / a}]\rangle$
$C = \langle\langle q_0^1, q_0^2, m\rangle, [\boxtimes, \boxed{a}]\rangle$
$C'' = \langle\langle q_1^1, q_1^2, m\rangle, [\boxtimes, \boxed{c / a}]\rangle$
$C' = \langle\langle q_1^1, q_1^2, m'\rangle, [\boxed{d}, \boxed{c / a}]\rangle$
$\langle q_0^2, [\boxed{a}]\rangle$
$\langle q_1^2, [\boxed{c / a}], \boxed{a}\rangle$
a
m
m'

Fig. 4 Two recognising steps of R (*middle*), built from steps of R_1 (*top*), and steps of R_2 (*bottom*). The step $C \xrightarrow{a} C'$ simultaneously updates two m-registers. In the figure the second component of the configurations, i.e. the word that might be recognised, is omitted for clarity

Instead, if both m-registers store the same dynamic symbol c, the merge is still m, and the transition t above enables the step $C \xrightarrow{a} C''$ and guarantees the invariant.

We are now ready to state and prove some closure properties of FSNA$_\pm$ and FSNA$_+$:

Theorem 2 *(Closure properties)*

	$\cup$	$\cap$	$\overline{\cdot}$	$\bullet$	$*$
$\mathcal{L}$(FSNA$_\pm$)	✓	×	×	✓	✓
$\mathcal{L}$(FSNA$_+$)	✓	✓	×	×	×

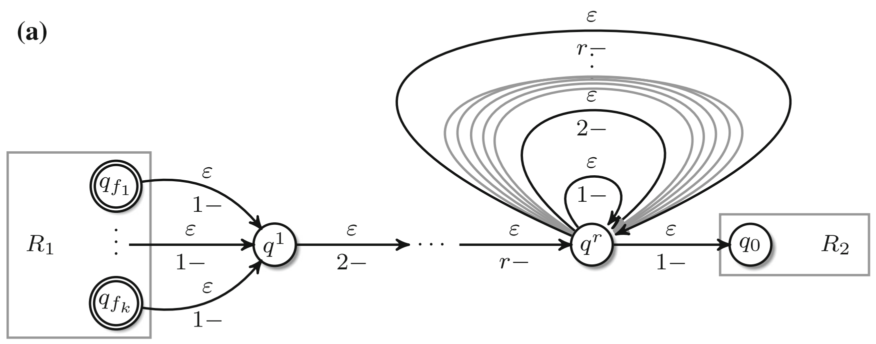
Proof Union: it suffices a new initial state with two outgoing ε-transition to the old initial states.

Concatenation (and Kleene star) for FSNA$_\pm$: a sequence of transitions from the final states of the first FSNA$_\pm$ make inactive all the m-registers, leading to a state, call it q^r, having loops that can empty them all (see Fig. 5a; note that the m-registers may not be emptied in the state q^r). Then an ε-transition goes from q^r to the initial state of the second automaton.

Complement of FSNA$_\pm$: Consider $L = \{w \mid \exists a \in \Sigma_d, n \in \mathbb{N}.a$ appears $2n + 1$ times in $w\}$ that is recognised by the FSNA$_\pm$ in Fig. 5b. Assume that its complement $\overline{L} = \{w \mid \forall a \in \Sigma_d.\ \exists n \in \mathbb{N}.\ a$ appears $2n$ times in $w\}$ is recognised by an automaton R with r m-registers. This automaton also accepts ww, where $w = a_1, \dots, a_{r+1}, \forall i, j\ (i \neq j).a_i \neq a_j$. However, after recognising w, there exists some a_i that is not s-top of any m-register. The word ww' where w' is obtained by w by replacing a_i with a fresh symbol b is accepted by R, as well: contradiction because $L \ni ww' \notin \overline{L}$.

Complement for FSNA$_+$: Property 1 (Σ^* is the complement of $\emptyset$) suffices.

Concatenation (and Kleene star) for FSNA$_+$: Consider $L = \{ww' \mid w \in L$ and $w' \in L'\}$, with L, L' languages of two FSNA$_+$ R, R'. If R accepts a string w such that $a \in w, a \in w'$ but $a \neq$ s-top(N_i) for all the m-registers in the final configuration of all accepting runs, then we obtain a contradiction because a can not be s-pushed since not fresh.



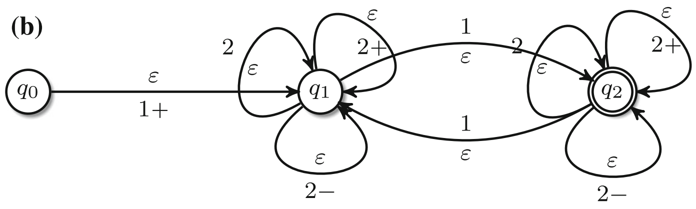


Fig. 5 **a** The concatenation automaton of two automata R_1 and R_2 has the states and the transition of both of them, as initial state the one of R_1, as final states the ones of R_2. The final states of R_1 are connected to q^r, q^r is connected to the initial state of R_2. The self-loops in q^r are used to empty the r m-registers. **b** An automaton recognising $L = \{w \mid \exists a.w[i] = a \text{ and } a \text{ appears } 2n + 1 \text{ times in } w\}$

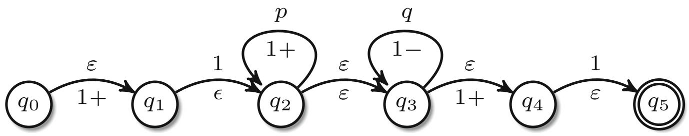


Fig. 6 The language $L_2 = \{ap^nq^mb \mid a \in \Sigma_d, a = b \Rightarrow m > n\}$

Intersection of FSNA$_\pm$ Let $L_1 = \{ap^nq^ma\}$, with $a \in \Sigma_d$ and p, q be chosen symbols in Σ_s. Clearly, L_1 is regular. Consider the language recognised by the automaton in Fig. 6: $L_2 = \{ap^nq^mb \mid a \in \Sigma_d, a = b \Rightarrow m > n\}$. Now, the language $L_1 \cap L_2 = \{ap^nq^ma \mid m > n\}$ can not be recognised by any FSNA$_\pm$.

Intersection of FSNA$_+$ We formalise here the intuitive construction given in Fig. 3, the proof uses Definition 9 and Lemmas 1 and 2 given below. Given two automata R_1 and R_2, we construct the intersection automaton $R_1 \cap R_2$ recognising $\mathcal{L}(R_1) \cap \mathcal{L}(R_2)$.

The proof of the equivalence $\mathcal{L}(R_1 \cap R_2) = \mathcal{L}(R_1) \cap \mathcal{L}(R_2)$ can be obtained by induction on the length of the runs by using Lemmas 1 and 2. Without loss of generality, we consider the simplifying assumption that all the registers are active in the initial configuration. Indeed all the m-registers can be initialised by initial ε-transitions, without modifying the recognised language.

Definition 9 *(Intersection Automaton)* The intersection automaton of two FSNA$_+$s $R_1 = \langle Q_1, q_0^1, \Sigma, \delta_1, r, F_1\rangle$ and $R_2 = \langle Q_2, q_0^2, \Sigma, \delta_2, r, F_2\rangle$ is the following FSNA$_+$2:

$$R_1 \cap R_2 = \langle \overline{Q}, \overline{q_0}, \Sigma, \overline{\delta}, 2r, \overline{F}\rangle, \text{ where}$$

– $\overline{Q} = Q_1 \times M \times Q_2$ where M is a set of merge functions for m-registers

- $\overline{q_0} = \langle q_0^1, m^*, q_0^2 \rangle$, with m^* s.t. m_1^*, m_2^* are the identity functions (i.e. the r registers of the two intersecting automata are merged onto the first r registers of $R_1 \cap R_2$)
- $\overline{F} = \{\langle q_1, m, q_2 \rangle \mid q_1 \in F_1, q_2 \in F_2, m \in M\}$
- $\langle q_1, m, q_2 \rangle \xrightarrow[\overline{\Delta_1}, \overline{\Delta_2}]{\overline{\sigma}} \langle q_1', m', q_2' \rangle \in \overline{\delta}$ iff $m \overset{\Delta_1, \Delta_2}{\curvearrowright} m'$ and

  $q_1 \xrightarrow[\Delta_1]{\sigma_1} q_1' \in \delta_1$ and $q_2 \xrightarrow[\Delta_2]{\sigma_2} q_2' \in \delta_2$ and
  - if $\sigma_1, \sigma_2 \in \underline{r}$ then $\overline{\sigma} = m_1(\sigma_1) = m_2(\sigma_2)$ and
  - if $\sigma_1, \sigma_2 \in \Sigma_s$ then $\overline{\sigma} = \sigma_1 = \sigma_2$ and
  - $(\overline{\Delta_1}, \overline{\Delta_2}) = \overset{m}{m'}\big(\Delta_1, \Delta_2\big)$

  or $q_1 \xrightarrow[\Delta_1]{\varepsilon} q_1' \in \delta_1$ and
  - $\overline{\sigma} = \varepsilon$ and
  - $q_2' = q_2$ and
  - $(\overline{\Delta_1}, \overline{\Delta_2}) = \overset{m}{m'}\big(\Delta_1, \varepsilon\big)$

  or $q_2 \xrightarrow[\Delta_2]{\varepsilon} q_2' \in \delta_2$ symmetric to the previous case.

Now we prove the two lemmata used before for proving that $R_1 \cap R_2$ accepts $L(R_1) \cap L(R_2)$. Intuitively, the first states that whenever two automata R_1, R_2 make a step with the same label, also the automaton $R_1 \cap R_2$ can perform the very same step.

Lemma 1 *Let R_1 and R_2 be two FSNA$_+$, let $a \neq \varepsilon$ and*
$step_1 : \langle q_1, aw, [N_1^1, \dots, N_r^1] \rangle \longrightarrow \langle q_1', w, [N_1'^1, \dots, N_r'^1] \rangle$ and
$step_2 : \langle q_2, aw, [N_1^2, \dots, N_r^2] \rangle \longrightarrow \langle q_2', w, [N_1'^2, \dots, N_r'^2] \rangle$ be steps of R_1 and R_2, respectively.

Then for any merge m and $[M_1 \dots, M_{2r}]$ m-registers of $R_1 \cap R_2$ such that

$$m([N_1^1, \dots, N_r^1, N_1^2, \dots, N_r^2]) = [M_1, \dots, M_{2r}]$$

there exists the step of $R_1 \cap R_2$
$\overline{step} : \langle \langle q_1, m, q_2 \rangle, aw, [M_1, \dots, M_{2r}] \rangle \longrightarrow \langle \langle q_1', m', q_2' \rangle, w, [M_1', \dots, M_{2r}'] \rangle$ with

$$m'([N_1'^1, \dots, N_r'^1, N_1'^2, \dots, N_r'^2]) = [M_1', \dots, M_{2r}']$$

Proof Assume that $q_1 \xrightarrow[\Delta_1]{\sigma_1} q_1'$ and $q_2 \xrightarrow[\Delta_2]{\sigma_2} q_2'$ justify step$_1$ and step$_2$. Then we have the following cases, depending on the labels of these transition.

- $\sigma_1, \sigma_2 \in \underline{r}$
  Define m' such that $\forall i, j \in \underline{r}\, (i \neq \Delta_1, j \neq \Delta_2). m_1'(i) = m_1(i), m_2'(j) = m_2(j)$.
  If s-top$(N_{\Delta_1}'^1)$ = s-top$(N_{\Delta_2}'^2)$ $(\Delta_1, \Delta_2 \neq \varepsilon)$

  - if $\forall i. i \overset{\mathrm{m}}{\not\leftrightarrow} \Delta_2 \wedge \Delta_1 \overset{\mathrm{m}}{\not\leftrightarrow} i$ then let $m_1'(\Delta_1) = m_2(\Delta_2)$ and $m_2'(\Delta_2) = m_2(\Delta_2)$.
  - Otherwise if $k \overset{\mathrm{m}}{\leftrightarrow} \Delta_2$ or $\Delta_1 \overset{\mathrm{m}}{\leftrightarrow} k$ then let $m_1'(\Delta_1) = m_2'(\Delta_2) = h \notin \mathrm{Img}(m)$.

  If s-top$(N_{\Delta_1}'^1) \neq$ s-top$(N_{\Delta_2}'^2)$ but

  - $m_1'(\Delta_1)$ (resp. $m_2'(\Delta_2)$) is such that s-top$(N_{\Delta_1}'^1)$ = s-top$(N_k'^2)$, $\Delta_1 \neq \varepsilon$ for some $k \neq \Delta_2$ then let $m_1'(\Delta_1) = m_2(k)$.
  - Otherwise,

- if $\varDelta_1 \overset{\mathrm{m}}{\leftrightarrow} k$ for some $k \neq \varDelta_2$, then let $m'_1(\varDelta_1) = h \notin \mathrm{Img}(m)$.
- Otherwise, if $\varDelta_1 \overset{\mathrm{m}}{\not\leftrightarrow} k$ for all $k \neq \varDelta_2$ then let $m'_1(\varDelta_1) \in \{h\} \cup m(\varDelta_1)$, with $h \notin \mathrm{Img}(m)$.

Recall that m is a merge and note that m, m' may possibly differ in $\varDelta_1, \varDelta_2$. Now we show that also m' is a merge, by showing its projections are injective. By contradiction, assume m' is not injective, then if $m'(\varDelta_1) \neq m(\varDelta_1)$ (resp. for $\varDelta_2$), by construction, it is only the case that $m'_1(\varDelta_1) = m'_2(k)$ for some k. If $k = \varDelta_2$ then $m'_1(\varDelta_1) = m'_2(k) \notin \mathrm{Img}(m)$, contradiction because m'_1, m'_2 are injective since there is no k s.t. $m'_1(k) = m'_1(\varDelta_1)$ or $m'_2(k) = m'_2(\varDelta_2)$. If $k \neq \varDelta_2$ then we have that only m'_2 can be non-injective, but this requires $m'_2(\varDelta_2) = m'_2(k), k \neq \varDelta_2$ but this is not possible by construction.

We show that $m \overset{\varDelta_1,\varDelta_2}{\curvearrowright} m'$: condition (1) is trivially satisfied by construction, conditions (2–3) are taken explicitly into account in the construction.

Since step$_1$ and step$_2$ fulfil the hypothesis, by condition 1.2 of Definition 2, it turns out that both $N^1_{\sigma_1}, N^2_{\sigma_2}$ are active and $a = \text{s-top}(N^1_{\sigma_1}) = \text{s-top}(N^2_{\sigma_2})$. This last fact implies that $m_1(\sigma) = m_2(\sigma)$ since m is merge. By letting $\overline{\sigma} = m_1(\sigma)$, conditions (1) and (2) imply that $\text{s-top}(M_{\overline{\sigma}}) = a$.

By construction of $A_1 \cap A_2$ we then have the transition

$$\bar{t} : \langle q_1, m, q_2 \rangle \xrightarrow[\overline{\varDelta_1}, \overline{\varDelta_2}]{\overline{\sigma}} \langle q'_1, m', q'_2 \rangle \in \overline{\delta}$$

where $(\overline{\varDelta_1}, \overline{\varDelta_2}) = \overset{m}{m'}(\varDelta, \varDelta')$.

Next we shall prove that m' is a merge of m-registers through which $[M'_1, \dots, M'_{2r}]$ can be obtained from $[M_1, \dots, M_{2r}]$ and $\overline{\text{step}}$ is justified by $\bar{t}$.

First, $\forall i \in \underline{r}(i \neq \overline{\varDelta_1}, \overline{\varDelta_2}).M_i = M'_i$. If $\overline{\varDelta_1} \neq \varepsilon \wedge m(\varDelta_1) = m'(\varDelta_1)$ then let $M'_{\overline{\varDelta_1}} = \text{s-push}(b_1, M_{\overline{\varDelta_1}})$ with $b_1 = \text{s-top}(N'^1_{\varDelta_1})$ otherwise let $M'_{\overline{\varDelta_1}} = M_{\overline{\varDelta_1}}$. Symmetrically for $M'_{\overline{\varDelta_2}}$ (note that $\overline{\varDelta_1} \neq \overline{\varDelta_2}$).

We prove now that $m'([N'^1_1, \dots, N'^1_r, N'^2_1, \dots, N'^2_r]) = [M'_1 \dots, M'_{2r}]$. The conditions (1) and (2) are satisfied because for all $i, j \in \underline{r}$ s.t. $i \neq \varDelta_1, j \neq \varDelta_2$ we have $m'_1(i) = m_1(i), m'_2(j) = m_2(j)$, the involved m-register are left untouched and m is a merge of m-registers. By construction of $M'_{m'(\varDelta_1)}$ (resp. $M'_{m'(\varDelta_2)}$) we also have that $\text{s-top}(M'_{m'(\varDelta_1)}) = \text{s-top}(N'_{\varDelta_1})$. The condition (3) is implied by the construction of m', condition (4) holds because b_1, b_2 are in $\bigcup_{i \in \underline{2r}} \|M'_i\|$ only if they are in $\bigcup_{i \in \underline{r}} \|N'^1_i\|$ or $\bigcup_{i \in \underline{r}} \|N'^2_i\|$ respectively.

We now show that the condition 2 of Definition 5 is satisfied. With the construction above $\bigcup_{i \in \underline{2r}} \|M_i\| = \bigcup_{i \in \underline{r}} \|M'_i\| \backslash \{b_1, b_2\}$. Since $\bigcup_{i \in \underline{r}} \|N^1_i\| \cup \bigcup_{i \in \underline{r}} \|N^2_i\| = \bigcup_{i \in \underline{2r}} \|M_i\|$, if $b_1 \in \bigcup_{i \in \underline{r}} \|N^2_i\|$ then $b_1 \in \bigcup_{i \in \underline{r}} \|M_i\|$, otherwise, by condition 2 of Definition 2 for step$_1$, $b_1 \notin \bigcup_{i \in \underline{r}} \|N^1_i\|$. This holds symmetrically for b_2. Then if b_1, b_2 are pushed on top of $M_{\overline{\varDelta_1}}, M_{\overline{\varDelta_2}}$ then they are fresh, i.e. $b_1, b_2 \notin \bigcup_{i \in \underline{r}} \|M_i\|$. Also, when both are pushed on top of $M_{\overline{\varDelta_1}}, M_{\overline{\varDelta_2}}$ we have $b_1 \neq b_2$, so satisfying condition 2 of Definition 5.

Since a satisfies condition 1 of Definition 5 for $\bar{t}$, the following step exists:

$$\langle\langle q_1, aw, m, q_2 \rangle, [M_1, \dots, M_r]\rangle \xrightarrow{a} \langle\langle q'_1, w, m', q'_2 \rangle, [M'_1, \dots, M'_r]\rangle$$

- if $\sigma_1, \sigma_2 \in \Sigma_s$, the proof is analogous to that of the previous case: take m' as above, then by construction of $R_1 \cap R_2$ we have the following transition, where $\overline{\sigma} = \sigma_1 = \sigma_2$

$$\bar{t} : \langle q_1, m, q_2 \rangle \xrightarrow[\overline{\Delta_1}, \overline{\Delta_2}]{\overline{\sigma}} \langle q_1', m', q_2' \rangle$$

With the same construction of $[M_1', \ldots, M_r']$ above, we obtain

$$\langle \langle q_1, aw, m, q_2 \rangle, [M_1, \ldots, M_r] \rangle \xrightarrow{a} \langle \langle q_1', w, m', q_2' \rangle, [M_1', \ldots, M_r'] \rangle$$

- $\sigma_1 = \varepsilon$ or $\sigma_2 = \varepsilon$, trivial. □

The following lemma states that whenever the automaton $R_1 \cap R_2$ makes a step, also the automata R_1 and R_2 can perform the very same step.

Lemma 2 *Let R_1 and R_2 be two FSNA$_+$, let $a \neq \varepsilon$ and let $\overline{step} : \langle \langle q_1, m, q_2 \rangle, aw, [M_1, \ldots, M_{2r}] \rangle \longrightarrow \langle \langle q_1', m', q_2' \rangle, w, [M_1', \ldots, M_{2r}'] \rangle$ be a step of $R_1 \cap R_2$ then for any $[N_1^1 \ldots, N_r^1], [N_1^2 \ldots, N_r^2]$ such that*

$$m([N_1^1, \ldots, N_r^1, N_1^2, \ldots, N_r^2]) = [M_1, \ldots, M_{2r}]$$

there exist two steps of R_1 and R_2
$step_1 : \langle q_1, aw, [N_1^1, \ldots, N_r^1] \rangle \longrightarrow \langle q_1', w, [N_1'^1, \ldots, N_r'^1] \rangle$ and
$step_2 : \langle q_2, aw, [N_1^2, \ldots, N_r^2] \rangle \longrightarrow \langle q_2', w, [N_1'^2, \ldots, N_r'^2] \rangle$ and

$$m'([N_1'^1, \ldots, N_r'^1, N_1'^2, \ldots, N_r'^2]) = [M_1', \ldots, M_{2r}']$$

Proof Assume that $\langle q_1, m, q_2 \rangle \xrightarrow[\overline{\Delta_1}, \overline{\Delta_2}]{\overline{\sigma}} \langle q_1', m', q_2' \rangle$ justifies $\overline{\text{step}}$.

- if $\overline{\sigma} \neq \varepsilon$ then by construction of $R_1 \cap R_2$ we have that $t_1 : q_1 \xrightarrow[\Delta_1]{\sigma_1} q_1'$ and $t_2 : q_2 \xrightarrow[\Delta_2]{\sigma_2} q_2'$. We only prove the case $\overline{\sigma} \in \underline{r}$; the others follow from a similar argument.
  - if $\overline{\sigma} \in \underline{r}$:
    let $[N_1'^1, \ldots, N_r'^1]$ (resp. $[N_1'^2, \ldots, N_r'^2]$) be such that $\forall i \in \underline{r}\ (m_1'(i) \neq \overline{\Delta_1} \wedge m_1'(i) = m_1(i)).N_i'^1 = N_i^1$. If $\Delta_1 \neq \varepsilon$ then take

$$N_{\Delta_1}'^1 = \text{s-push}(\text{s-top}(M'_{m_1'(\Delta_1)}), N_{\Delta_1}^1)$$

    Condition 1 of Definition 2 is satisfied for the transition t_1 because s-top$(M_{\overline{\sigma}})$ = s-top$(N_{\sigma_1}) = a$ since $\overline{\sigma} = m_1(\sigma_1)$ by construction and m is a merge of m-registers. We prove now condition 2 of Definition 2. For $b = \text{s-top}(M'_{m_1'(\Delta_1)})$ we consider the two cases: $\overline{\Delta_1} \neq \varepsilon$ and $\overline{\Delta_1} = \varepsilon$. In the first case the condition is guaranteed by the fact that $b \notin \bigcup_{j \in \underline{2r}} \|M_j\|$, hence, since m is a merge of m-registers (condition 3) $b \notin \bigcup_{j \in \underline{r}} \|N_j^1\|$. When $\overline{\Delta_1} = \varepsilon$ then we have $m_1'(\Delta_1) \neq m_2'(\Delta_2)$, in this case $m_1'(\Delta_1) \neq m_1(\Delta_1)$, by injectivity of m_1', by the fact that m is a merge of m-registers and by the fact that m_1' only differs from m_1 in Δ_1 we have that $\forall i \in \underline{r}.\text{s-top}(M_{m_1'(\Delta_1)}) \neq \text{s-top}(N_i^1)$. For $c = \text{s-top}(M'_{m_2'(\Delta_2)})$. We consider two cases: $\overline{\Delta_2} \neq \varepsilon$ and $\overline{\Delta_2} = \varepsilon$. In the first case the condition is guaranteed by the fact that $c \notin \bigcup_{j \in \underline{2r}} \|M_j\|$, hence, since m is a merge of m-registers (condition 3) $c \notin \bigcup_{j \in \underline{r}} \|N_j^2\|$. When $\overline{\Delta_2} = \varepsilon$ then we have two cases: $m_2'(\Delta_2) = m_1'(\Delta_1) = \overline{\Delta_1}$ or $m_2'(\Delta_2) \neq m_1'(\Delta_1)$. In the first case

the condition is satisfied by the same reasoning above because $c = \text{s-top}(M'_{\overline{\Delta_1}})$, the second case is verified only when $m'_2(\Delta_2) \neq m_2(\Delta_2)$, in this case, by injectivity of m'_2, by the fact that m is a merge of m-registers and by the fact that m'_2 only differs from m_2 in Δ_2 we have that $\forall i \in \underline{\mathtt{r}}.\text{s-top}(M_{m'_2(\Delta_2)}) \neq \text{s-top}(N^2_i)$.
Hence all the conditions for t_1, t_2 are satisfied, so both step$_1$ and step$_2$ exist.
We are left to prove that $m'([N'^1_1, \dots, N'^1_r, N'^2_1, \dots, N'^2_r]) = [M'_1 \dots, M'_{2r}]$. The conditions (1) and (2) are satisfied because for all $i, j \in \underline{\mathtt{r}}$ s.t. $i \neq \Delta_1, j \neq \Delta_2$ we have $m'_1(i) = m_1(i), m'_2(j) = m_2(j)$, the involved m-register are left untouched and m is a merge of m-registers. By construction of N'_{Δ_1} (resp. N'_{Δ_2}) we also have that $\text{s-top}(M'_{m'(\Delta_1)}) = \text{s-top}(N'_{\Delta_1})$. The condition (3) is implied by the construction of N'_{Δ_1} because (for some j) $m'_1(\Delta_1) = m'_2(j)$ implies $\text{s-top}(N'^1_{\Delta_1}) = \text{s-top}(M'_{m'(\Delta_1)}) = \text{s-top}(N'^2_j)$, condition (4) holds because b, c are in $\bigcup_{i\in\underline{\mathtt{r}}} \|N'^1_i\|$ or $\bigcup_{i\in\underline{\mathtt{r}}} \|N'^2_i\|$ only if they are in $\bigcup_{i\in\underline{2\mathtt{r}}} \|M'_i\|$ respectively. □

Having proved both Lemmas 1 and 2, we conclude the proof of the whole theorem. □
Note that the same argument used in the proof of the intersection between two FSNA$_\pm$ suffices to establish the following property.

Property 3 *There exists languages L_1 and L_2, accepted by FSNA$_\pm$ and FSNA$_+$, respectively such that $L_1 \cap L_2$ is not a regular nominal language.*

We now study the decidability of some typical problems in automata theory, namely those of membership, universality and emptiness. Given an automaton R, the first and the second problems amount to check if a word w and Σ^* are accepted by R; the third if $\mathcal{L}(R) = \emptyset$.

Theorem 3 *1. The membership problem for FSNA$_\pm$ is decidable*
2. *The universality problem is undecidable for FSNA$_\pm$, while for FSNA$_+$ it is decidable, and the answer is always negative*
3. *The emptiness problem is decidable for FSNA$_\pm$*

Proof 1. A trivial linear non-deterministic procedure suffices.
2. Theorem 4 proved in the next sub-section guarantees that FSNA$_\pm$ are more expressive than FMA. Now the statement follows because universality is undecidable for FMA [39]. Since FSNA$_+$ can not generate Σ^*, the second claim is proved.
3. The actual content of the m-registers is negligible when reasoning about emptiness, only their activation states are important because a step can be inhibited by an inactive m-register. So, we can abstract a configuration $\langle q, w, [N_1, \dots, N_r]\rangle$ as a pair $\langle q, [x_1, \dots, x_r]\rangle$, where x_i is the activation state of N_i. Suppose now that there exist an accepting run for w. The problem is now reduced to verify the emptiness of a non-nominal finite state automaton. □

4 Expressiveness of FSNA$_\pm$

In the literature there are many nominal languages, working on infinite alphabets or on data-words. We consider here only those that are intuitively *regular*, in that they can not express Dyck-like languages, e.g. the language $\{ww^R\}$ when $|w|$ is not bounded (see Example 3). An incomplete list of the regular languages in the literature includes *variable finite automata* (VFA) [27], *finite memory automata* (FMA) [29] and their extension with non-deterministic reassignment (NFMA) [30], *Usage Automata* (UA) [7], *fresh-register automata* (FRA) [47]

Table 2 A comparison of the closure properties of FSNA$_{\pm}$, FSNA$_{+}$ automata with the ones of other automata in the literature, namely VFA [27], FMA [29], UA [7], FRA [47], HRA [48], CRA [14] and NFMA [30]

	$\cup$	$\cap$	$\overline{\cdot}$	$\bullet$	$*$
$\mathcal{L}$(FSNA$_{\pm}$)	✓	×	×	✓	✓
$\mathcal{L}$(FSNA$_{+}$)	✓	✓	×	×	×
$\mathcal{L}$(VFA)	✓	✓	×	??	??
$\mathcal{L}$(FMA)	✓	✓	×	✓	✓
$\mathcal{L}$(UA)	✓	✓	×	✓	×
$\mathcal{L}$(FRA)	✓	✓	×	×	×
$\mathcal{L}$(HRA)	✓	✓	×	✓	✓
$\mathcal{L}$(CRA)	✓	✓	×	??	??
$\mathcal{L}$(NFMA)	✓	✓	×	✓	✓

and their evolution *history register automata* (HRA) [48], *class register automata* (CRA) [14], *Data Walking Automata* (DWA) [33], the variant of *HD-automata* in [18] and the `fp-automata` [31].

A notion similar to the one of the m-register can be found in HRA [48] and in the chronicles of the *chronicle deallocating automata* [32].

Table 2 recalls the closure properties of FSNA$_{\pm}$ and FSNA$_{+}$ and of those models above for which the literature provides these results.

The next theorem investigates the relationship among our models and the (regular) ones in the literature in terms of expressiveness. When considering data-words, we assume for simplicity that there is a single action α on resources, that will be omitted in words, i.e. we write a instead of $\alpha(a)$. We write $A \gtrless\!\!\!\!/\ B$ when two sets A, B are incomparable, i.e. $A \nsubseteq B, B \nsubseteq A$.

Theorem 4 (Expressiveness comparison)

1. $\mathcal{L}(FSNA_{\pm}) \supsetneq \mathcal{L}(VFA) \supsetneq \mathcal{L}(UA)$
2. $\mathcal{L}(FSNA_{\pm}) \supsetneq \mathcal{L}(FMA)$
3. $\mathcal{L}(FSNA_{+}) \supsetneq \mathcal{L}(UA)$
4. $\mathcal{L}(FSNA_{+}) \gtrless\!\!\!\!/\ \mathcal{L}(VFA)$
5. $\mathcal{L}(FSNA_{+}) \gtrless\!\!\!\!/\ \mathcal{L}(FMA)$
6. $\mathcal{L}(FSNA_{\pm}) \gtrless\!\!\!\!/\ \mathcal{L}(HRA)$
7. $\mathcal{L}(FSNA_{+}) \gtrless\!\!\!\!/\ \mathcal{L}(\texttt{fp-automata})$
8. $\mathcal{L}(FSNA_{\pm}) \supseteq \mathcal{L}(\texttt{fp-automata})$

Proof Sketch:

1. $\mathcal{L}$(FSNA$_{\pm}$) $\supsetneq$ $\mathcal{L}$(VFA) $\supsetneq$ $\mathcal{L}$(UA)
   The VFA have been proved more expressive than UA in [19]. A FSNA$_{\pm}$ simulates a VFA by using m-registers associated to each variable (and never s-popping them), the distinguished free variable y of VFA can be mapped to a m-register that is always s-popped after being used. The last condition matches the one of VFA requiring the symbols associated to each occurrence of y in the witnessing pattern to be different from the other variables, but possibly equal to another symbol associated with y. The language L_0 in the Example 1 belongs to $\mathcal{L}$(FSNA$_{\pm}$) but not to $\mathcal{L}$(VFA).

2. $\mathcal{L}(\mathrm{FSNA}_{\pm}) \supsetneq \mathcal{L}(\mathrm{FMA})$
The main differences between the two models are the following. The registers of FMA have an initial assignment, while $\mathrm{FSNA}_{\pm}$ have static resources playing the same role (and initialisation can anyway be done by initial ε-transitions that FMA have not). FMA associate the reassignment function ρ with states rather than with edges; the same effects can be obtained by $\mathrm{FSNA}_{\pm}$ when all the edges starting from a state q have the same Δ. Additionally, ρ reassigns a register using the input symbol, while $\mathrm{FSNA}_{\pm}$ update an m-register (through an ε-transition) and then recognize the fresh symbol in it. In FMA, all the registers have to be different, and a reassignment may update a register with the same symbol it already contains; the $\mathrm{FSNA}_{\pm}$ can simulate this behaviour using two edges with $\Delta = i+$ and $\Delta = \varepsilon$, accounting for the cases a new symbol is assigned or the same one is reassigned, respectively. So, $\mathcal{L}(\mathrm{FSNA}_{\pm}) \supseteq \mathcal{L}(\mathrm{FMA})$ and the language L_0 in Example 1 shows that inclusion is strict.
3. $\mathcal{L}(\mathrm{FSNA}_{+}) \supsetneq \mathcal{L}(\mathrm{UA})$
The expressiveness of UA is the same of VFA without the y (see [34]), so the construction in item 1 suffices (note that there will be no delete transitions).
4. $\mathcal{L}(\mathrm{FSNA}_{+}) \not\gtreqless \mathcal{L}(\mathrm{VFA})$ and 5. $\mathcal{L}(\mathrm{FSNA}_{+}) \not\gtreqless \mathcal{L}(\mathrm{FMA})$
Consider $L_0 = \{w \in \Sigma_d^* \mid \forall i, j.\, w[i] \neq w[j]\}$ of Example 1. We have that $L_0 \in \mathcal{L}(\mathrm{FSNA}_{+})$ but $L_0 \notin \mathcal{L}(\mathrm{VFA}) \cup \mathcal{L}(\mathrm{FMA})$. Also $\Sigma^* \in \mathcal{L}(\mathrm{FMA}) \cap \mathcal{L}(\mathrm{VFA})$ but $\Sigma^* \notin \mathcal{L}(\mathrm{FSNA}_{+})$.
5. $\mathcal{L}(\mathrm{FSNA}_{\pm}) \not\gtreqless \mathcal{L}(\mathrm{HRA})$
Consider the language $L = \{a_0 b_0 \dots a_n b_n \mid \forall i, j.\, i \neq j \Rightarrow a_i \neq a_j, b_i \neq b_j\}$, L is not recognised by any $\mathrm{FSNA}_{\pm}$ but it is recognised by an HRA because of the capability of using multiple stories. On the other hand the language $L' = \{a_1 \dots a_n b_1 \dots b_n \mid \forall i, j.\, i \neq j \Rightarrow a_i \neq a_j \wedge b_i \neq b_j, n - i \geq j \Rightarrow b_i \neq a_j\}$ is in $\mathcal{L}(\mathrm{FSNA}_{\pm})$ but not in $\mathcal{L}(\mathrm{HRA})$. This is because m-registers are stacks while places are sets.
6. $\mathcal{L}(\mathrm{FSNA}_{+}) \not\gtreqless \mathcal{L}(\texttt{fp-automata})$ and $\mathcal{L}(\mathrm{FSNA}_{\pm}) \supseteq \mathcal{L}(\texttt{fp-automata})$
Both $\mathrm{FSNA}_{\pm}$, FSNA_{+} recognise the language $\{a_1 \dots a_n \mid i \neq j \Rightarrow a_i \neq a_j\}$, that instead is not by any `fp-automata`. However FSNA_{+} can not recognise Σ^*, a language in $\mathcal{L}(\texttt{fp-automata})$. □

5 Pushdown Nominal Automata

The introduction discusses a simple program that picks up and releases unboundedly many resources. The behaviour of this program was conveniently abstracted as the Dyck-like context-free nominal language of words

$$\texttt{new(r1)new(r2)}\dots\texttt{new(rn) rel(rn)}\dots\texttt{rel(r2)rel(r1)} \qquad (1)$$

where $r1, r2, \dots, rn$ are pairwise distinct resources. Various kinds of recognisers for these context-free nominal languages have been recently proposed and justified by their theoretical and practical relevance [12,13,17,37,42].

Below, we extend $\mathrm{FSNA}_{\pm}$ with a stack, so obtaining our version of nominal, push-down automata. Of course, the nominal language $\{ww^R\}$ of Example 3 is accepted by one of them.

We store elements of the infinite alphabet Σ in the stack of a $\mathrm{PDNA}_{\pm}$, and we can push on it strings of symbols in Σ, possibly retrieved through the indexes of m-registers. By pushing, e.g., the string $a\,3\,b$ one actually pushes a s-top$(N_3)\,b$. A preliminary definition is in order to handle these cases.

Definition 10 Let $\zeta \in (\Sigma_s \cup \underline{r})^*$ and let S be a stack. Then, $Pushreg(\zeta, S)$ extends the standard push operation as follows

$$\begin{array}{l} Pushreg(\varepsilon, S) = S \\ Pushreg(z\,\zeta', S) = Pushreg(\zeta', push(\sigma, S)) \end{array} \qquad \text{where} \quad \sigma = \begin{cases} z & \text{if } z \in \Sigma_s \\ \text{s-top}(N_z) & \text{if } z \in \underline{r} \end{cases}$$

Definition 11 *(Pushdown Nominal Automata)* A *Pushdown Nominal Automata* (PDNA$_\pm$) is $A = \langle Q, q_0, \Sigma, \delta, r, F\rangle$ where:

- Q, q_0, r, F are as in FSNA$_\pm$ (Definition 1)
- δ is the transition relation: $(q, \sigma, Z, q', \Delta, \zeta) \in \delta$ with $\sigma \in \Sigma_s \cup \underline{r} \cup \{\varepsilon, \top\}$, $Z \in \Sigma_s \cup \underline{r} \cup \{\varepsilon, ?\}$, $\Delta \in \{i+, i- \mid i \in \underline{r}\} \cup \{\varepsilon\}$, $\zeta \in (\Sigma_s \cup \underline{r})^*$.
  For $(q, \sigma, Z, q', \Delta, \zeta) \in \delta$ we use the notation $q \xrightarrow[\Delta,\zeta]{\sigma,Z} q' \in \delta$

A configuration is a tuple $C = \langle q, w, [N_1, \ldots, N_r], S\rangle$ where $q, w, [N_1, \ldots, N_r]$ are as in FSNA$_\pm$ and S is a stack with symbols in Σ.

A configuration $\langle q_f \in F, \varepsilon, [N_1, \ldots, N_r], \boxtimes\rangle$ is *final* ($\boxtimes$ is the empty stack).

As defined below, PDNA$_\pm$ may use in a richer way than standard pushdown automata the top of the stack, call it a. First, we can compare the current symbol in the input with a, if the symbol σ in the transition to be applied is $\top$. Also, if $Z = \varepsilon$ the string obtained from ζ is pushed on the stack, as explained above. Instead, if $Z = i$ the top a is popped from the stack, provided that the s-top of the i^{th} m-register is a. Finally, if $Z =?$ a pop occurs, with no further constraints.

Definition 12 *(Recognising Step)* Given a PDNA$_\pm$ A, the step $\langle q, w, [N_1, \ldots, N_r], S\rangle \rightarrow \langle q', w', [N_1', \ldots, N_r'], S'\rangle$ occurs iff $q \xrightarrow[\Delta,\zeta]{\sigma,Z} q' \in \delta$ and the following hold

1. condition 1 of Definition 2 and $\sigma = \top \Rightarrow w = top(S)w'$ and
2. condition 2 of Definition 2 and
3. $\begin{cases} Z = \varepsilon \Rightarrow S' = Pushreg(\zeta, S) \text{ and} \\ Z = i \Rightarrow S' = Pushreg(\zeta, pop(S)) \wedge top(S) = \text{s-top}(N_i) \text{ and} \\ Z =? \Rightarrow S' = Pushreg(\zeta, pop(S)) \end{cases}$

Finally, the language accepted by a PDNA$_\pm$ A, which we call *(nominal) context-free*, is

$$L(A) = \left\{w \in \Sigma^* \mid \exists\rho : C_1 = \langle q_0, w, [\boxtimes, \ldots, \boxtimes], \boxtimes\rangle \rightarrow^* C_k, \text{with } C_k \text{ final}\right\}$$

The following example shows that PDNA$_\pm$ are able to express the above `new-rel` language on data-words (1).

Example 4 The PDNA$_+$ accepting (1) is in Fig. 7b. The labels of transitions, but Δ, contain `new`(u), `rel`(u), $u \in \underline{r} \cup \Sigma_s$. Figure 7b also shows the run for `new`(a) `new`(b) `rel`(b) `rel`(a); also here we omit the strings in configurations and we only mention the symbols in the m-registers. Note that, only keeping the names of the resources, we get $\bigcup_{r\in\mathbb{N}} L_r = \{ww^R \mid w \in \Sigma_d^*\}$, for L_r of Example 3.

The *late usage* pattern discussed in the introduction can be expressed by PDNA$_\pm$. This comes with the ability of recognising symbols on the stack, while they have been deleted from the m-registers. The following example witnesses this feature.

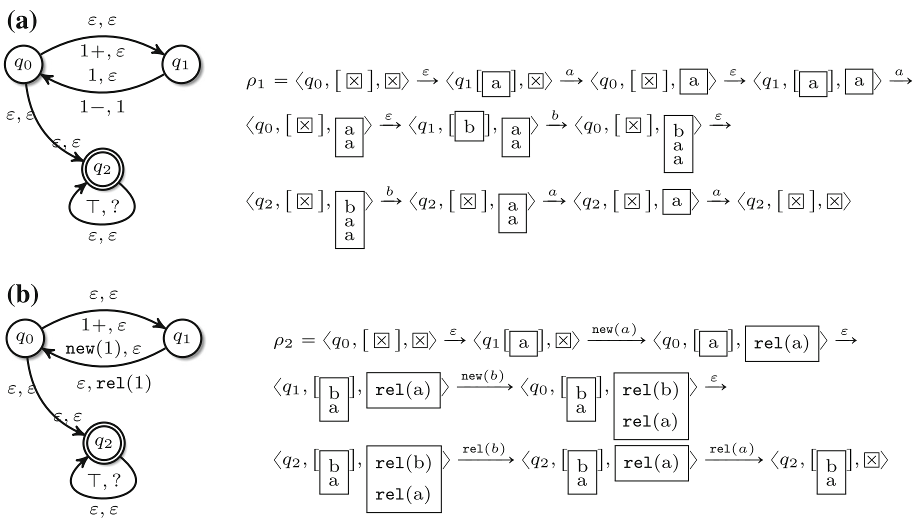

(a)
q_0
q_1
q_2
ε, ε
$1+, \varepsilon$
$1, \varepsilon$
$1-, 1$
ε, ε
ε, ε
$\top, ?$
ε, ε
$\rho_1 =$
(b)
q_0
q_1
q_2
ε, ε
$1+, \varepsilon$
$\mathtt{new}(1), \varepsilon$
$\varepsilon, \mathtt{rel}(1)$
ε, ε
ε, ε
$\top, ?$
ε, ε
$\rho_2 =$

Fig. 7 **a** A PDNA$_\pm$ accepting $\{ww^R \mid w \in \Sigma_d^*\}$, and a run on *aabbaa*. **b** A PDNA$_+$ for the data-word language of the `updateFiles` function in the introduction, and a run on `new(a) new(b) rel(b) rel(a)`. Strings are omitted in configurations

Example 5 Figure 7a shows a PDNA$_\pm$ accepting $L_p = \{ww^R \mid w \in \Sigma_d^*\}$, and a run accepting *aabbaa* (for brevity, we do not write the strings to be recognised in the configurations, as the current symbols label the steps). The automaton behaves just as a FSNA$_\pm$ in the 1st, 3rd, 5th and 7th steps of ρ_1. Additionally, in this initial part of the run, the stack is involved in the 2nd, 4th and 6th steps. They all occur because of edge $q_1 \xrightarrow[1-,1]{1,\varepsilon} q_0$, that causes the symbol in the m-register 1 to be pushed on the stack. In steps 8th,9th,10th the edge $q_2 \xrightarrow[\varepsilon,\varepsilon]{\top,?} q_2$ causes the top of the stack to be (successfully) matched with the current symbol (as dictated by the label $\top$) and popped (because of ?).

As done for FSNA$_\pm$ we introduce the class of automata that update two m-registers at the same time, and the sub-class of PDNA$_\pm$ without delete transitions.

Definition 13 *(*PDNA$_+$ and PDNA$_+$2*)*

- A PDNA$_+$ is a PDNA$_\pm$ with no edges $q \xrightarrow[i-,\zeta]{\sigma,Z} q'$.
- A PDNA$_+$2 is a PDNA$_+$ with transitions of the form $q \xrightarrow[(\Delta_1,\Delta_2),\zeta]{\sigma,Z} q'$ (cf. Definition 4)

As expected, the class of languages accepted by PDNA$_\pm$ strictly includes that accepted by PDNA$_+$. Indeed, the same proof of Property 1 applies here. Just as done for FSNA$_+$2, we can prove that PDNA$_+$2 and PDNA$_+$ have the same expressive power. In spite of the reduced expressiveness, PDNA$_+$ can accept a wide class of (Dyck-like) context-free languages, for instance the automaton in Example 4 is a PDNA$_+$.

6 Properties of PDNA$_\pm$

Obviously, the class of pushdown nominal languages includes the regular ones.

Property 4 $\mathcal{L}(FSNA_{\pm}) \subsetneq \mathcal{L}(PDNA_{\pm})$

Proof Inclusion is trivially proved: from a given FSNA$_{\pm}$ obtain the equivalent PDNA$_{\pm}$ by adding labels $\zeta = \varepsilon$, $Z = \varepsilon$ to each edge. Example 3 suffices to prove that the inclusion is strict. □

We now study under which operators the classes of languages accepted by PDNA$_{\pm}$ and PDNA$_{+}$ are closed.

Theorem 5 *(Closure properties)*

	$\cup$	$\cap$	$\overline{\cdot}$	$\bullet$	$*$
$\mathcal{L}$(PDNA$_{\pm}$)	✓	×	×	✓	✓
$\mathcal{L}$(PDNA$_{+}$)	✓	×	×	×	×

Proof Union:
The construction is the same of Theorem 2 in both cases; of course the initial ε-transitions do not alter the stack ($\zeta = Z = \varepsilon$).

Intersection and complement:
Follows from the classical results on context-free languages, the same languages of the classical counterexamples can be used in our case.

Concatenation and Kleene star:
The proof in Theorem 2 applies here as well; note that the stack is empty in the initial and final configurations. □

A classical result in automata theory is that the class of context-free languages is closed by intersection with the class of regular ones. We investigate the same property in the nominal case, and we find that only the intersection of FSNA$_{+}$ and PDNA$_{+}$ is a PDNA$_{+}$ (hence a PDNA$_{\pm}$).

Theorem 6 *(Intersection)*

is a	FSNA$_{\pm}$ $\cap$ PDNA$_{\pm}$	FSNA$_{\pm}$ $\cap$ PDNA$_{+}$	FSNA$_{+}$ $\cap$ PDNA$_{\pm}$	FSNA$_{+}$ $\cap$ PDNA$_{+}$
PDNA$_{\pm}$	×	×	×	✓
PDNA$_{+}$	×	×	×	✓

Proof Consider the FSNA$_{\pm}$ language $L_1 = \{ap^n bq^m r^{n'} cs^{m'} d \mid a = c \Rightarrow n' > n, b = d \Rightarrow m' > m, a \neq b, c \neq d\}$ in Fig. 8 and the PDNA$_{+}$ language $L_2 = a\{p\}^* b\{q\}^*\{r\}^* a\{s\}^* b$. The language $L_1 \cap L_2 = \{ap^n bq^m r^{n'} cs^{m'} d \mid n' > n, m' > m, a \neq b\}$, by classical reasoning on context-free languages, is not recognised by any PDNA$_{+}$ nor PDNA$_{\pm}$. Note that L_1 can be recognised by the PDNA$_{\pm}$ obtained by adding $Z = \varepsilon$, $\zeta = \varepsilon$ to all the edges in Fig. 8 and L_2 is a nominal regular language recognised by both a FSNA$_{+}$ and a FSNA$_{\pm}$. This justifies the entries × of the statement.

We prove now that PDNA$_{+}$ $\cap$ FSNA$_{+}$ is a PDNA$_{+}$:

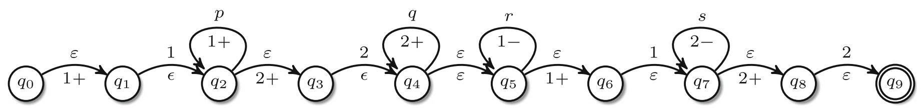


Fig. 8 A FSNA$_\pm$ accepting the language $L_1 = \{ap^n bq^m r^{n'} cs^{m'} d \mid a = c \Rightarrow n' > n, b = d \Rightarrow m' > m, a \neq b, c \neq d\}$

The proof follows step by step that of Theorem 2, with additional care to manage the stack, which however is only determined by how the PDNA$_+$ handles it. The detailed construction follows.

Given the PDNA$_+$ $\langle Q_1, q_0^1, \Sigma, \delta_1, r, F_1\rangle$ and the FSNA$_+$ $\langle Q_2, q_0^2, \Sigma, \delta_2, r, F_2\rangle$, their *intersection automaton* (of type PDNA$_+$2) is $\langle \overline{Q}, \overline{q_0}, \Sigma, \overline{\delta}, 2r, \overline{F}\rangle$, where

- $\overline{Q} = Q_1 \times M \times Q_2$, with M set of merge functions
- $\overline{q_0} = \langle q_0^1, m^*, q_0^2\rangle$ where m^* is a merge such that m_1^*, m_2^* are the identity functions (i.e. the m-registers of the two intersecting automata are initially merged onto the firsts r register of the intersection automaton)
- $\overline{F} = \{\langle q_1, m, q_2\rangle \mid q_1 \in F_1, q_2 \in F_2, m \in M\}$
- $\langle q_1, m, q_2\rangle \xrightarrow[\overline{\Delta_1}, \overline{\Delta_2}, \overline{\zeta}]{\overline{\sigma}, \overline{Z}} \langle q_1', m', q_2'\rangle \in \overline{\delta}$ iff $m \overset{\Delta_1, \Delta_2}{\curvearrowright} m'$ and

  $q_1 \xrightarrow[\Delta_1, \zeta]{\sigma_1, Z} q_1' \in \delta_1$ and $q_2 \xrightarrow[\Delta_2]{\sigma_2} q_2' \in \delta_2$ and ($\sigma_1, \sigma_2 \in \underline{r}$ or $\sigma_1, \sigma_2 \in \Sigma_s$) and

  - if $\sigma_1, \sigma_2 \in \underline{r}$ then $\overline{\sigma} = m_1(\sigma_1) = m_2(\sigma_2)$ and
  - if $\sigma_1, \sigma_2 \in \Sigma_s$ then $\overline{\sigma} = \sigma_1 = \sigma_2$ and
  - $(\overline{\Delta_1}, \overline{\Delta_2}) = \overset{m}{m'}(\Delta_1, \Delta_2)$, $\overline{Z} = m_1(Z)$, $\overline{\zeta} = m_1(\zeta)$

  or $q_1 \xrightarrow[\Delta_1, \zeta]{\top, Z} q_1' \in \delta_1$ and $q_2 \xrightarrow[\Delta_2]{\sigma_2} q_2' \in \delta_2$ and $\sigma_2 \in \underline{r}$, $\overline{\sigma} = m_2(\sigma_2)$ and $\overline{\zeta} = m_1(\zeta)$

  and either $Z = k \in \underline{r}$ implies $k \overset{m}{\leftrightarrow} \sigma_2$, $\overline{Z} = m_2(\sigma_2)$, $(\overline{\Delta_1}, \overline{\Delta_2}) = \overset{m}{m'}(\Delta_1, \Delta_2)$,

  or $Z = ?$ implies $\overline{Z} = m_2(\sigma_2)$, $(\overline{\Delta_1}, \overline{\Delta_2}) = \overset{m}{m'}(\Delta_1, \Delta_2)$

  or $q_1 \xrightarrow[\Delta_1 \zeta]{\varepsilon, Z} q_1' \in \delta_1$ and $\overline{\sigma} = \varepsilon$, $(\overline{\Delta_1}, \overline{\Delta_2}) = \overset{m}{m'}(\Delta_1, \varepsilon)$, $\overline{Z} = m_1(Z)$, $\overline{\zeta} = m_1(\zeta)$

  or $q_2 \xrightarrow[\Delta_2]{\varepsilon} q_2' \in \delta_2$ $\overline{\sigma} = \varepsilon$, $(\overline{\Delta_1}, \overline{\Delta_2}) = \overset{m}{m'}(\Delta_1, \varepsilon)$, $\overline{Z} = \varepsilon$, $\overline{\zeta} = \varepsilon$ □

We finally prove that the emptiness problem is decidable for PDNA$_+$. The proof relies on a variant of the classical pumping lemma. Roughly it says that, given a language L recognised by a PDNA$_+$, there exists a constant n, such that any string $w \in L, |w| > n$ can be decomposed as $w = uvxyz$, such that also $w' = u'x'z'$ belongs to L with u', x', z' obtained from u, x, z by carefully substituting (distinguished) dynamic symbols and erasing v and y. Before proving it, we need some auxiliary definitions and lemmata.

Since we focus on the emptiness problem, we are interested in the existence of a word, rather than in its actual shape. Therefore, whenever immaterial, we feel free to omit from now onwards the word in configurations and the input symbol in transitions.[5]

[5] Consequently, we have that $\langle q, [N_1, \ldots, N_r], S\rangle \to \langle q', [N_1', \ldots, N_r'], S'\rangle$ iff there exists $q \xrightarrow[\Delta, \zeta]{\sigma, Z} q' \in \delta$ satisfying the conditions $(2-3)$ of Definition 12 and the following holds:

Notation From now onwards, assume as given a PDNA$_+$ and let $C = \langle q, [N_1, \dots, N_r], S\rangle$ be a configuration; $\rho = C_1 \to^* C_k, C_i = \langle q_i, w_i, [N_1^i, \dots, N_r^i], S_i\rangle$ be a run; $B = [B[1], \dots, B[n]]$ denote an array, and let $B[i, \dots, j], i \leq j$ denote the portion of the array between the i-th and j-th positions. We will also use the array notation for stacks, assuming that the leftmost item $S[1]$ is the bottom and $S[i]$ is the i-th element in it, the height of a stack S will be denoted $|S|$.

Also, call *swap* f an injective partial function $f : \Sigma_d \rightharpoonup \Sigma_d$, and its homomorphic extensions to strings, tuples, array and stacks.

What follows extends similar definitions and proofs of [2].

Definition 14 *(C-rep)* Let E_S be the set of symbols occurring in the stack S of a configuration C such that $e \in E_S$ iff $\forall i.top(N_i) \neq e$. Let

$$first(e, S) = \begin{cases} 1 & \text{if } top(S) = e \\ first(e, pop(S)) + 1 & \text{otherwise} \end{cases}$$

Let $f_S : E_S \to \{1, \dots, |S|\}$ to be such that $f_S(e) = first(e, S)$ (note that f_S is injective).

The function C-rep(S) that returns a stack of symbols in $\Sigma_s \cup \{i \mid 1 \leq i \leq r\} \cup \{\underline{i} \mid 1 \leq i \leq |S|\}$ is defined by:

- C-rep([]) = []
- C-rep($b :: S'$) $= a :: C$-rep(S') iff
  - $b \in \Sigma_s, a = b$ or
  - $b \in \Sigma_d, \exists i.b = top(N_i), a = i$ or
  - $b \in \Sigma_d, \forall i.b \neq top(N_i), a = \underline{f_S(b)}$

Definition 15 *(Activation state)* The *activation state* of the m-registers of a configuration C is an array $m = [m[1], \dots, m[r]]$ where $m[i] = 1$ iff N_i is active, $m[i] = 0$ otherwise.

Definition 16 *(Representative state)* The *representative state* of a configuration C is the triple (q, m, R) where m is the activation state of the m-registers and $R = C$-rep(S), i.e. R represents S on C. We write $C \sim C'$ to indicate that C has the same representative state of C'.

Lemma 3 *Let $C_1 \to C_1'$ then for any configuration C_2 such that $C_2 \sim C_1$ there exists C_2' such that $C_2 \to C_2'$ and $C_2' \sim C_1'$.*

Footnote 5 continued

$$(1m) \begin{cases} \sigma = \top \Rightarrow top(S) \text{ is defined} \\ \sigma = i \Rightarrow \text{s-top}(N_i) \text{ is defined} \end{cases}$$

If $\langle q, [N_1, \dots, N_r], S\rangle \to \langle q', [N_1', \dots, N_r'], S'\rangle$ because there exists $q \xrightarrow[\Delta,\zeta]{\sigma,Z} q' \in \delta$ satisfying the conditions $(1m, 2, 3)$ then by setting for any w' the word w such that

$$\begin{cases} \sigma &= \varepsilon \Rightarrow w = w' \\ \sigma &= \top \Rightarrow w = aw' \\ \sigma &= i \Rightarrow w = aw' \end{cases}$$

we have that also $\langle q, w, [N_1, \dots, N_r], S\rangle \to \langle q', w', [N_1', \dots, N_r'], S'\rangle$. By induction, if $\langle q, [N_1, \dots, N_r], S\rangle \to^* \langle q', [N_1', \dots, N_r'], S'\rangle$ then for any word w' there exists a word w such that $\langle q, w, [N_1, \dots, N_r], S\rangle \to^* \langle q', w', [N_1', \dots, N_r'], S'\rangle$.

Proof Let $t = (q, \sigma, Z, q', \Delta, \zeta) \in \delta$ be used for justifying the transition $C_1 \to C_1'$. We first show that t justifies also $C_2 \to C_2'$ by constructing a suitable $C_2' = \langle q', [M_1', \dots, M_r'], T' \rangle$. First of all, note that, being $C_1 \sim C_2$, the main stacks S and T have the same depth and the m-registers N_i and M_i have the same activation state. Therefore, since C_1 satisfies $(1m)$, also C_2 does, i.e. :

$$\begin{cases} \sigma = \top & \Rightarrow top(T) \text{ is defined} \\ \sigma = i & \Rightarrow \text{s-top}(M_i) \text{ is defined} \end{cases}$$

For the same reason, in condition (3), the operations $pop(T)$, $top(T)$ and s-top(M_i) are defined and so are the arguments of the operation Pushreg. Note that ζ may contain a reference to a register j and again we have that the required s-top(M_j) is defined, because the activation state of M_j is the same of N_j.

We are left to prove that condition (2) can be fulfilled by the M_j' and to prove that $C_1' \sim C_2'$. We proceed by cases on Δ.

Case $\Delta = i+$) Let $\forall j (j \neq i). M_j' = M_j$, $M_i' =$ s-push(c, M_i) with c to be $T[first(c, M_i)]$ if $c \in S$ to preserve the representative state, that requires to relate T' with S'. Otherwise we choose $c \notin M_j, \forall j \in \underline{r}$ and $c \notin T$. We are left to prove that $C_2' \sim C_1'$. Trivially C_2' and C_1' have the same state q'. The activation state of the m-registers is also the same, because in both configurations only the i-th is affected (if active it is left such as well it becomes active because of the s-push), while the activation state of the others is the same in C_1' and C_2' because $C_1 \sim C_2$. Also $E_S = E_T$ and the same ζ is pushed on both stacks, so $E_{S'} = E_{T'}$. Now $f_{S'}(\text{s-top}(N_i)) = f_{T'}(\text{s-top}(M_i))$, that proves $C_1' \sim C_2'$.

Case $\Delta = i-$) Let $M_j' = M_j, \forall j (j \neq i)$ and $M_i' =$ s-pop(M_i) which is defined because $C_1 \sim C_2$. The proof that the tuple $(q', \Delta, \zeta) \in \delta(q, \sigma, Z)$ justifies $C_2 \to C_2'$ is similar to the case above. Only the $i - th$ m-register is affected, if active it gets deactivated, it is left deactivated otherwise.

Case $\Delta = \varepsilon$) Letting $M_j' = M_j, \forall j \in \underline{r}$ suffices to fulfil condition (2). The proof that the transition $(q, \sigma, Z, q', \Delta, \zeta)$ belongs to δ justifies $C_2 \to C_2'$ is similar to the case above. Only the i-th m-register is affected, if active it gets deactivated, it is left deactivated otherwise. □

Definition 17 *(Level)* A level $G = (i, j, h)$ with height l on ρ is a triple (i, j, h) such that $1 \le i < j < h \le k$ and

- $|S_i| = |S_h|, \left|S_j\right| = |S_i| + l$
- $|S_i| \le |S_u| \le \left|S_j\right|$ for all $u.i \le u \le j$.
- $|S_h| \le |S_u| \le \left|S_j\right|$ for all $u.j \le u \le h$

Given a level G on ρ, define two indices $l_\downarrow^G$, $f_\uparrow^G$, called respectively *last-push* and *first-pop* of G.

$$l_\downarrow^G = \max\{y \le j \mid \left|S_y\right| = |S_i|\} \qquad f_\uparrow^G = \min\{y \ge j \mid \left|S_y\right| = |S_i|\}$$

Figure 9 shows an example of levels, $l_\downarrow$ and $f_\uparrow$.

Property 5 *Given a level* (i, j, h) *with height* l *on* ρ*, for each* $k < l$ *there exists a level* (u, j, v) *for some* u, v *with height* k.

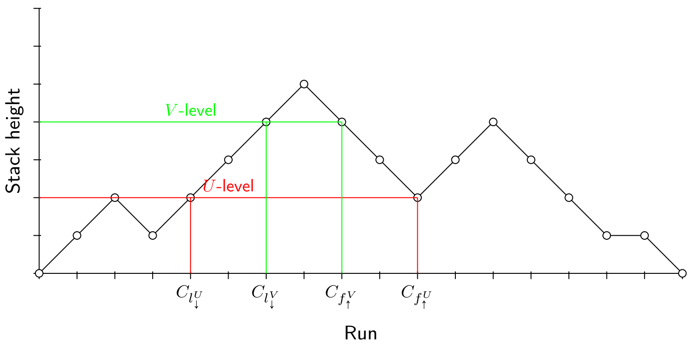


Fig. 9 An example of V-level and G-level on a run

Definition 18 *(Full state)* Let G be a level on ρ and let $C_{l_\downarrow^G} = \langle q, [N_1, \dots, N_r], a :: S\rangle$, $C_{f_\uparrow^G} = \langle q', [N'_1, \dots, N'_r], S'\rangle$.

The *full state* of a level G on ρ is the tuple (c, q, m, q', m') such that:

- If $a \in \Sigma_s$ then $c = a$, if $a \in \Sigma_d$ and $\exists i.top(N_i) = a$ then $c = i$, if $a \in \Sigma_d$ and $\forall i.top(N_i) \neq a$ then $c = \star$.
- m, m' are the activation states of the m-registers in $C_{l_\downarrow^G}$, $C_{f_\uparrow^G}$, respectively.

Property 6 *Let* $C_1 = \langle q_1, [N_1^1, \dots, N_r^1], S_1\rangle \to^n C_n = \langle q_n, [N_1^n, \dots, N_r^n], S_n\rangle$ *and let* $G = (1, j, n)$ *be a level on the above run with height* l*, for some* j*. Then there exists a* cutoff run $D_{l_\downarrow^G}, \dots, D_{f_\uparrow^G}$ *with* $D_i = \langle q_i, [N_1^i, \dots, N_r^i], S'_i\rangle$, $S'_i = T :: S_i[1, \dots, |S_i| - l]$ *for any stack* T *with* $top(T) = top(S_{l_\downarrow^G})$.

Proof Sketch: by definition of level the transitions does not depend on the content of the stack below the height of $S_{l_\downarrow^G}$ □

We now prove our restricted pumping lemma: a de-pumping lemma.

Lemma 4 *(De-Pumping Lemma) Let* $A = \langle Q, q_0, \delta, r, \Sigma_s \cup \Sigma_d, F\rangle$ *be a PDNA*$_+$ *and let* $p' = 2^r\,|Q|^2\,(|\Sigma_s| + r + 1)$ *and* $p = 2^r\,|Q|\,(|\Sigma_s| + r + p' + 1)^{p'} + 1$. *For each word* $w \in L(A)$ *such that* $|w| > p$ *and* ρ *is (one of) the shortest among its accepting runs, we can construct a word* $w' \in L(A)$ *with a strictly shorter accepting run.*

Proof Let $w \in L(A)$ such that $|w| > p$; take $\rho = C_1 \to \cdots \to C_k$, $C_i = \langle q_i, w_i, [N_1^i, \dots, N_r^i], S_i\rangle$ out of the set of the shortest accepting runs; and let l be the maximum height of the stack in ρ.

Case $l \le p'$) Recall that $|w| > p$, hence ρ contains at least p configurations. There are at most $2^r\,|Q|\,(|\Sigma_s|+r+l+1)^l < p$ different representative states of the configurations of ρ. Hence there are at least two configurations $C_x, C_y, x < y$ with the same representative state. By applying Lemma 3, from the run $C_y \to^* C_k$ we obtain that also $C_x \to^* F$ for some F with the same representative state of C_k. Therefore, also F is a final configuration. The thesis follows because the run $\rho' = C_1 \to^* C_x \to^* F$ is shorter than ρ.

Case $l > p'$) Note that there is a level on ρ with height l, say $G = (i, j, h)$. By Property 5, there exist at least l levels (u, j, v) with different heights that are levels on ρ.

There are only $p' < l$ different full states that can be associated with these levels, hence there exist two levels, say $U = (u_1, j, v_1)$ and $V = (u_2, j, v_2)$ with the same full state. Assume w.l.o.g. $n_U = |S_{u_1}| = |S_{v_1}| < |S_{u_2}| = |S_{v_2}| = n_V$. Let $C_{l_\downarrow^U}, C_{l_\downarrow^V}$ be the configuration with index last-push and let $C_{f_\uparrow^U}, C_{f_\uparrow^V}$ be the configuration with index first-pop of level U and V respectively (see Fig. 9).

By Lemma 6, from the run $C_{l_\downarrow^V} \to^* C_{f_\uparrow^V}$ it is possible to obtain a cut off run

$$D_{l_\downarrow^V} = \langle q_{l_\downarrow^V}, w_{l_\downarrow^V}, [N_1^{l_\downarrow^V}, \dots, N_r^{l_\downarrow^V}], T_{l_\downarrow^V} \rangle \to^*$$
$$D_{f_\uparrow^V} = \langle q_{f_\uparrow^V}, w_{f_\uparrow^V}, [N_1^{f_\uparrow^V}, \dots, N_r^{f_\uparrow^V}], T_{f_\uparrow^V} \rangle$$

where $T_i = S_{l_\downarrow^U} :: S_i[n_V, \dots, |S_i|]$.

Since $T_{l_\downarrow^V} = T_{f_\uparrow^V} = S_{l_\downarrow^U} = S_{f_\uparrow^U}$ and $C_{l_\downarrow^U}, C_{f_\uparrow^U}$ have the same full state of $C_{l_\downarrow^V}, C_{f_\uparrow^V}$, respectively, it follows that $D_{l_\downarrow^V} \sim C_{l_\downarrow^U}$ and $D_{f_\uparrow^V} \sim C_{f_\uparrow^U}$.

Consequently, by Lemma 3, from the run $D_{l_\downarrow^V} \to^* D_{f_\uparrow^V}$ we obtain the run $C_{l_\downarrow^U} \to^* H$ for some $H \sim D_{f_\uparrow^V} \sim C_{f_\uparrow^U}$. By the same lemma, from the run $C_{f_\uparrow^U} \to^* C_k$ we obtain a run $H \to^* F$, where F has the same representative state of C_k, hence it is final.

The thesis follows because the run $C_1 \to^* C_{l_\downarrow^U} \to^* H \to^* F$ is shorter than ρ. □

We eventually prove the decidability of the emptiness problem for our push-down nominal automata with no delete transitions; we conjecture that it is instead undecidable for PDNA$_\pm$.

Theorem 7 *Given a PDNA$_+$ A, it is decidable whether $L(A) = \emptyset$.*

Proof (Sketch) By repeatedly applying the De-Pumping Theorem 4, $L(A)$ is non empty if it contains a word w', made of distinguished symbols, and such that $|w'| \le p$, where p is the constant in Theorem 4. □

7 Expressiveness of PDNA$_\pm$

To the best of our knowledge, the literature has different notions of nominal context-free languages, i.e. able to express Dyck-like languages: *quasi context-free languages* (QCFL) [17], Usages introduced in [7], the context-free automata (that we call here NPA) in [13], DMPA [15], HOPAD [42], *Pebble* automata [39], and the *pushdown r-register system* (r-PDRS) [37].

Below we compare the properties and the expressive power of PDNA$_\pm$ with that of above models for different notions of nominal context-free languages.

Table 3 shows some closure properties of our automata with those of the other models, when presented in the literature. To the best of our knowledge, the closure of a context-free nominal language with a regular one has not been investigated explicitly for other nominal models, even if it is folklore that intersecting a QCFL with a FMA automaton can be done mimicking the construction of the intersection between FMA. Remarkably, we have constructed the PDNA$_+$ resulting from the intersection of a PDNA$_+$ with a FSNA$_+$ (see Theorem 6).

Table 3 A comparison of the closure properties of our automata with the ones of QCFL [17], Usages [7] and NPA [12]

	$\cup$	$\cap$	$\overline{\cdot}$	$\bullet$	$*$
$\mathcal{L}(\text{PDNA}_\pm)$	✓	×	×	✓	✓
$\mathcal{L}(\text{PDNA}_+)$	✓	×	×	×	×
$\mathcal{L}(\text{QCFL})$	✓	×	×	✓	✓
$\mathcal{L}(\text{Usages})$	✓	×	×	×	×
$\mathcal{L}(NPA)$	✓	??	??	??	??
$\mathcal{L}(Pebble)$	✓	??	??	??	??

Table 4 Decidability of the emptiness problem for QCFL [17], Usages [7], DMPA [15], Pebble [39] and r-PDRS automata

	PDNA_+	$\text{PDNA}_\pm$	QCFL	Usages	*DMPA*	*Pebble*	*r-PDRS*
Emptiness	✓	??	✓	✓	×	×	✓

The decidability and the complexity of the emptiness problem has been investigated for QCFL, Usages, DMPA, *Pebble* automata because of the relevance of this problem in verification. In [37] the authors shows the decidability of reachability on r-PDRS automata, which implies the decidability of their emptiness. In Table 4 we summarise the decidability results of the emptiness problem for the above models and for ours.

We now compare the expressiveness of $\text{PDNA}_\pm$ and PDNA_+ with that of other models in the literature. Also here, we assume that data-words have a single action, not displayed in words (see Theorem 4).

Theorem 8 *(Expressivness Comparison)*

– $\mathcal{L}(PDNA_+) \not\gtreqless \mathcal{L}(QCFL)$
– $\mathcal{L}(PDNA_\pm) \supsetneq \mathcal{L}(QCFL)$
– $\mathcal{L}(PDNA_+) = \mathcal{L}(Usages)$
– $\mathcal{L}(PDNA_\pm) \supsetneq \mathcal{L}(Usages)$
– $\mathcal{L}(PDNA_\pm) \not\gtreqless \mathcal{L}(DMPA)$

Proof – $\mathcal{L}(\text{PDNA}_\pm) \supsetneq \mathcal{L}(QCFL)$
We consider the *infinite alphabet pushdown automata* (IAPA) that recognise $\mathcal{L}(QCFL)$ [17]. The same argument in Theorem 4, item 2 ($\mathcal{L}(\text{FSNA}_\pm) \supsetneq \mathcal{L}(FMA)$) suffices for showing the inclusion, which is strict, because L_0 in the Example 1 is not recognised by any IAPA.

– $\mathcal{L}(\text{PDNA}_+) = \mathcal{L}(\text{Usages})$
Usages are built from (static and dynamic) symbols n (actually $\alpha(n)$, where α is an action on n) with operation of sequentialization $\cdot$, nondeterminism $+$, recursion and creation of a new dynamic symbol, through νn (see Table 5).
When sequentialising two processes, the second can not use any dynamic symbol used by the first one, just as it happens when two PDNA_+ are sequentialised by connecting the final states of the first with the initial one of the second. Since there is no deletion, the m-registers monotonically grow.

Table 5 Operational semantics of Usages

$$(\text{epsilon})\ \frac{}{\varepsilon \cdot U, \mathcal{R} \xrightarrow{\varepsilon} U, \mathcal{R}} \qquad (\text{act})\ \frac{}{\alpha(r), \mathcal{R} \xrightarrow{\alpha(r)} \varepsilon, \mathcal{R}}$$

$$(\text{dot})\ \frac{U, \mathcal{R} \xrightarrow{a} U', \mathcal{R}'}{U \cdot V, \mathcal{R} \xrightarrow{a} U' \cdot V, \mathcal{R}'} \qquad (\text{plus})\ \frac{}{U + V, \mathcal{R} \xrightarrow{a} U, \mathcal{R}'} \qquad \frac{}{U + V, \mathcal{R} \xrightarrow{a} V, \mathcal{R}'}$$

$$\frac{}{\nu n.U, \mathcal{R} \xrightarrow{\varepsilon} U\{r/n\}, \mathcal{R} \cup \{r\}}\ \text{if } r \in \Sigma_d \backslash \mathcal{R}$$

$$(\text{rec})\ \frac{}{\mu h.U, \mathcal{R} \xrightarrow{\varepsilon} U\{{}^{\mu h.U}/_{h}\}, \mathcal{R}}\ \text{capture avoiding}$$

Nondeterministic choice $+$ directly corresponds to the union of two automata.
Recursion can be dealt with as done in [6] by transforming an expression in a BPA, that has an immediate counterpart as a PDNA$_+$.
For each occurrence of a νn in the usage at hand we associate an m-register. Creation of a new symbol, i.e. reducing the νn, corresponds to updating the corresponding m-register. When a νn occurs within a recursive expression, a renaming occurs to guarantee freshness of the dynamic symbol to be generated. Note that only a finite number of m-registers is necessary, as the number of νn occurring in a usage is fixed and Property 7 holds.

- $\mathcal{L}(\text{PDNA}_\pm) \supsetneq \mathcal{L}(\text{Usages})$
  $\Sigma^* \in \mathcal{L}(\text{PDNA}_\pm)$, while $\Sigma^* \notin \mathcal{L}(\text{Usages})$ by Theorem 9.
- $\mathcal{L}(\text{PDNA}_+) \not\gtreqless \mathcal{L}(\text{QCFL})$
  The proof of Theorem 8, item 5 suffices (FSNA$_+ \supsetneq$ FMA).
- $\mathcal{L}(\text{PDNA}_\pm) \not\gtreqless \mathcal{L}(\text{DMPA})$ Using their multiple stacks, the DMPA can express the language (of patterns) $\{a^n b^n c^n\}$, that can not be recognised by a PDNA$_\pm$. However, their notion of freshness also requires that a new symbol can not occur in the stacks, which is not the case for PDNA$_\pm$. □

Note in passing that the expressiveness of `fp-automata` does not go beyond the one of PDNA$_\pm$, deallocation of `fp-automata` can be reproduced in PDNA$_\pm$ by delete transitions; swapping the contents of two registers in `fp-automata` via a permutation (there are only a finite number of them) can be done by PDNA$_\pm$ by suitably mentioning/updating/deleting the corresponding registers in the next states.

A detailed comparison between the expressiveness of PDNA$_\pm$ and r-PDRS [37] is out of the scope of this paper, mainly because recognising languages with an unbound number of fresh resources (unbound/global freshness), or with late usage, were not an issue for r-PDRS. Extending their main result on the decidability of reachability on r-PDRS in these cases does not seem straightforward. Indeed, the original result in [37] is based on the limited distinguishability lemma (Lemma 5.), which states that for any recognising run there exists an equally long run where only $3r$ symbols are used. This lemma would be false for e.g. $L = \{w \mid \forall i, j\ (i \neq j).w[i] \neq w[j]\}$ (accepted by a FSNA$_+$), because the length of any recognizing run of $w \in L$ is at least $|\|w\|| = |w|$, and it suffices to take a run longer than $3r$.

8 Conclusions

We modelled two new aspects of resource usage arising in current programming languages and paradigms, namely unbound freshness, i.e. the possibility of having unboundedly many fresh resources, and late usage, i.e. the possibility of referring to a resource even after it has been disposed. We do that through novel classes of automata for nominal languages, that read from the input unboundedly many fresh symbols. These symbols can explicitly be released, and re-used later on, as if they were fresh. We intuitively classify our automata as context-free PDNA$_\pm$, and regular FSNA$_\pm$, respectively, because they can recognise Dyck-like languages, and can not, respectively. We also considered a sensible restriction of them, PDNA$_+$ and FSNA$_+$, that can not release and re-use as fresh symbols already seen.

We studied some closure properties of our models: union, intersection, complementation, concatenation and Kleene star. We related the expressive power of our nominal automata to that of some analogous models in the literature and we investigated the decidability of the problems of emptiness and universality for them.

The decidability of the emptiness, in particular in the context-free case, is important because it provides foundations to some verification techniques. To the best of our knowledge, no class of nominal languages proposed so far is closed under complementation.

Lack of closure under complementation affects, for example, the standard *automata-based model-checking* procedure [49]. This wide-spread verification technique requires to represent both the model and the properties as languages L (typically context-free) and L' (typically regular) and to verify the emptiness of the intersection of L with the complement of L'. Indeed, an element of a non-empty intersection is a counterexample to the property in hand. However, this problem is mitigated when the property, i.e. a regular language, is not specified as the set of the accepted words, rather by the set of *non accepted* words, following the so-called *default-accept* paradigm [45]. In other words, one specifies the *unwanted* behaviour, so making complementation unnecessary at all. This is typically the case some properties arising in static analysis [26,40] and for security policies [4,45], that define behaviour that are deemed unacceptable. The specification of the property in the default-accept approach is usually done through an automaton (e.g. a *security automaton*), the language of which contains the unwanted behaviour. Operationally, one then builds a *run-time execution monitor* based on this automaton, that watches programs and performs a transition in correspondence with an execution step that possibly affects the property under monitoring. Right before the automaton reaches a final state, the program run is aborted, because it is about to violate some requirement. Following a static approach, one typically extracts an abstract behaviour from a program and model-checks it against the required property, expressed by an automaton where complementation is built-in (for security properties, see e.g. [6,8,28]).

The connection between nominal automata and program verification has been also investigated in [26]. The authors introduce a programmer-friendly language (TOPL) which can be used to express temporal properties of objects. A TOPL formula can be transformed to obtain a register automaton, which in turn can be used to monitor the execution of a Java program, seeking for violation of the property expressed by the formula, in a default-accept manner.

This paper contributes to this line of research, characterising a large class for which the automata based model-checking is feasible, when default-accept properties are of interest. We proposed PDNA$_+$ to express the model and FSNA$_+$ for the property, the intersection of which is again a PDNA$_+$. We also proved the decidability of emptiness problem for PDNA$_+$. These two results guarantee that model-checking is effective in the nominal setting, within the default-accept paradigm. Further investigation is required on understanding the impact the disposal mechanism has on the feasibility of this verification technique.

A different approach to verification has been recently proposed in [9]: compliance of a model with respect to a property is checked through a notion of simulation between automata. In [10], the same authors propose a logic with variables ranging over an infinite domain and establish a correspondence with a parametrised transition system. We can easily adapt their verification technique to our case, even though decidability of simulation is presently an open issue for our models.

Further research is needed to fill in the gap between our foundational results and prototypical implementations of our abstract model-checking procedure. Languages and logics which match the expressiveness of our models are also needed to make nominal technique easier to use for developers, in the style of [10,26].

Acknowledgements We thank Lillo Galletta, Alexander Kurz, Matteo Sammartino and Nikos Tzevelekos for many helpful discussions and the anonymouos referees for their suggestions.

Appendix: Usages

To compare the expressive power of PDNA$_\pm$ against Usages we need to extend the expressiveness theorems about Usages.

Theorem 9 *The language Σ^* is not generated by any Usages.*

Proof The proof has the following structure: a new transition system for Usages is given, this is provably equivalent to the original one [7]. Then a few lemmata are proved to provide support for the final argument.

In the following we will use the word *redex* to identify the source of a transition deduced by only applying an (instance of an) axiom.

Lemma 5 *If $U_0, \mathcal{R} \xrightarrow{\varepsilon} U_1, \mathcal{R}'$ and $\mu h.U$ is its redex, then $U_0\{{}^{\mu h.U}/_{h}\} = U_1$.*

Proof (By induction on the depth of the proof)

There are two exhaustive cases:

– Base case: $U_0 = \mu h.U$ and the thesis follows easily.

– Inductive case: $U_0 \xrightarrow{\varepsilon} U_1$ has been proved by $\dfrac{U_0' \xrightarrow{\varepsilon} U_1'}{U_0' \cdot V \xrightarrow{\varepsilon} U_1' \cdot V}$ by rule (seq) as last step. By the inductive hypothesis we have that $U_1' = U_0'\{{}^{\mu h.U}/_{h}\}$ and the thesis follows easily.

□

Lemma 6 *If $U_0, \mathcal{R}_0 \xrightarrow{a_0} U_1, \mathcal{R}_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} U_n, \mathcal{R}_n$ and k is the least index such that $\mu h.U$ is the redex of $U_k \xrightarrow{a_k} U_{k+1}$ and rule (rec) is never used in reducing $U_i, i < k$ then $U_i = C_i[\mu h.U]$ for some C_i and $a_k = \varepsilon$.*

Lemma 7 *Let $U_0, \mathcal{R}_0 \xrightarrow{a_0} U_1, \mathcal{R}_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} U_n, \mathcal{R}_n$ be a computation and let k be the least index such that $\mu h.U$ is the redex of $U_k \xrightarrow{a_k} U_{k+1}$ and rule (rec) is never used in reducing $U_i, i < k$ then $\exists U_i' \equiv U_i, i < k$ such that $U_0', \mathcal{R}_0 \xrightarrow{a_0} U_1', \mathcal{R}_1 \xrightarrow{a_1} \cdots U_k' \xrightarrow{a_{k+1}} U_{k+2}, \mathcal{R}_{k+2} \xrightarrow{a_{k+2}} \cdots \xrightarrow{a_{n-1}} U_n, \mathcal{R}_n$ and (rec) is never used reducing $U_i, 0 \leq i \leq k$.*

Proof By Lemma 6 $U_i = C_i[\mu h.U], i \leq k$, also $\mu h.U$ is never the redex of $U_i, i < k$, hence also $U_i' = C_i[U\{{}^{\mu h.U}/_{h}\}], \mathcal{R}_i \xrightarrow{a_i} U_{i+1}' = C_{i+1}[U\{{}^{\mu h.U}/_{h}\}], \mathcal{R}_{i+1}$. By Lemma 5 $U_k' = U_k[U\{{}^{\mu h.U}/_{h}\}] = U_{k+1}$. □

Lemma 8 *Let* $U_0, \mathcal{R}_0 \xrightarrow{a_0} U_1, \mathcal{R}_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} U_n$ *then* $\exists U'_0 \equiv U_0$ *such that* $U'_0, \mathcal{R}_0 \xrightarrow{a_0 a_1 \ldots a_{n-1}} U'_m, \mathcal{R}_m$ *for some* $U'_m, \mathcal{R}_m$ *such that* $\forall i. U'_i, \mathcal{R}_i \xrightarrow{a_i} U'_{i+1}, \mathcal{R}_{i+1}$ *is deduced using (rec) rule.*

Proof By repeated application of Lemma 7. □

Property 7 *(of capture avoiding substitutions) Given* $\mu h.U$*, if* n *occurs in the scope of* νn*, then* $U\{^{\mu h.U}/_{h}\}$ *contains a term* $\nu n'.U'$*,* $n' \neq n$ *and* n *does not occur in* U'*.*

Proof Follows because unfolding a recursion is capture-avoiding and all the bound names are different. □

For simplicity , we write a_n for the dynamic resource replacing a name n in U when the rule (new) is applied.

Let U with k occurrences of νn_i be such that $[\![U]\!] = \Sigma^*$ and let $s = a_{n_1} a_{n_2} \ldots a_{n_{k+1}} a_{n_1} a_{n_2} \ldots a_{n_{k+1}}$. By Lemma 8, there exists $U' \equiv U$ such that $U', \emptyset \xrightarrow{s} \overline{U}, \{a_{n_1}, \ldots, a_{n_{k+1}}\} \cup \mathcal{R}$ with no transition deduced using rule (rec).

Since $\nu.n_{k+1}$ does not occur in U, then there exists a subterm of U of the form $\mu h.\overline{U}$, with $\nu n_i.U''$, $(0 < i \leq k)$ in $\overline{U}$. Therefore, $U' = U\{\mu h.\overline{U}/h\}$ and the replacing term contains $\nu n_{k+1}.U''\{n_{k+1}/n_i\}$, $n_{k+1} \neq n_i$ for some i because our assumption of keeping bound names apart.

Therefore $\nu n_{k+1}.U''\{n_{k+1}/n_i\}$ occurs in U' for some U'' (by Lemma 8) and n_i does not occur in $U''\{n_{k+1}/n_i\}$. Also n_{k+1} must occur at least twice in $U''\{n_{k+1}/n_i\}$ and nowhere else. Since $U''\{n_{k+1}/n_i\}$ can not generate a_{n_i}, it can not generate $a_{n_{k+1}} a_{n_1} \ldots a_{n_k} a_{n_{k+1}}$. □

References

1. Abadi, M., Needham, R.M.: Prudent engineering practice for cryptographic protocols. IEEE Trans. Softw. Eng. **22**(1), 6–15 (1996). doi:10.1109/32.481513
2. Amarilli, A., Jeanmougin, M.: A proof of the pumping lemma for context-free languages through pushdown automata (2012). arXiv:1207.2819
3. Bartoletti, M., Costa, G., Degano, P., Martinelli, F., Zunino, R.: Securing Java with local policies. JOT **8**(4), 5–32 (2009)
4. Bartoletti, M., Degano, P., Ferrari, G.L.: Planning and verifying service composition. JCS **17**(5), 799–837 (2009)
5. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Semantics-based design for secure web services. IEEE Trans. Softw. Eng. **34**(1), 33–49 (2008)
6. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Local policies for resource usage analysis. ACM Trans. Program. Lang. Syst. **31**(6), 23 (2009)
7. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Model checking usage policies. Math. Struct. Comput. Sci. **25**(3), 710–763 (2015)
8. Bartoletti, M., Zunino, R.: LocUsT: a tool for checking usage policies. Tech. Rep. TR08-07, University of Pisa (2008)
9. Belkhir, W., Chevalier, Y., Rusinowitch, M.: Guarded variable automata over infinite alphabets (2013). arXiv:1304.6297
10. Belkhir, W., Rossi, G., Rusinowitch, M.: A parametrized propositional dynamic logic with application to service synthesis. In: Advances in Modal Logic 10, invited and Contributed Papers from the Tenth Conference on “Advances in Modal Logic, Groningen, The Netherlands, 5–8 August 2014, pp. 34–53 (2014). http://www.aiml.net/volumes/volume10/Belkhir-Rossi-Rusinowitch.pdf
11. Benedikt, M., Ley, C., Puppis, G.: Automata vs. logics on data words. In: Dawar, A., Veith, H. (eds.) CSL, LNCS, vol. 6247, pp. 110–124. Springer, Berlin (2010)
12. Bojańczyk, M., Klin, B., Lasota, S.: Automata theory in nominal sets (2011). http://www.mimuw.edu.pl/sl/PAPERS/lics11full.pdf

13. Bojanczyk, M., Klin, B., Lasota, S.: Automata with group actions. In: LICS, pp. 355–364. IEEE Computer Society, Washington, DC, USA (2011). doi:10.1109/LICS.2011.48
14. Bollig, B.: An automaton over data words that captures EMSO logic. In: Katoen, J.P. , König, B. (eds.) CONCUR 2011, LNCS, vol. 6901, pp. 171–186. Springer, Berlin (2011)
15. Bollig, B., Cyriac, A., Gastin, P., Kumar, K.N.: Model checking languages of data words. In: Birkedal, L. (ed.) FOSSACS 2012, LNCS, vol. 7213, pp. 391–405. Springer, Berlin (2012)
16. Chaki, S., Rajamani, S.K., Rehof, J.: Types as models: model checking message-passing programs. In: Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16–18, 2002, pp. 45–57 (2002)
17. Cheng, E.Y.C., Kaminski, M.: Context-free languages over infinite alphabets. Acta Inf. **35**(3), 245–267 (1998)
18. Ciancia, V., Tuosto, E.: A novel class of automata for languages on infinite alphabets. Tech. rep., CS-09-003, University of Leicester, UK (2009)
19. Degano, P., Ferrari, G.L., Mezzetti, G.: Nominal automata for resource usage control. In: Moreira, N., Reis, R. (eds.) CIAA 2012, LNCS, vol. 7381, pp. 125–137. Springer, Berlin (2012)
20. Dierks, T., Rescorla, E.: RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2 (2008). http://tools.ietf.org/html/rfc5246
21. Ferris, C., Farrell, J.: What are web services? Commun. ACM **46**(6), 31 (2003). doi:10.1145/777313.777335
22. Gabbay, M., Pitts, A.: A new approach to abstract syntax with variable binding. Form. Asp. Comput. **13**(3), 341–363 (2002)
23. Gershenfeld, N., Krikorian, R., Cohen, D.: The internet of things. Sci. Am. **291**(4), 76 (2004)
24. Gong, Z., Gu, X., Wilkes, J.: PRESS: predictive elastic resource scaling for cloud systems. In: Proceedings of the 6th International Conference on Network and Service Management, CNSM 2010, Niagara Falls, Canada, October 25–29, 2010, pp. 9–16 (2010). doi:10.1109/CNSM.2010.5691343
25. Gordon, A.D.: Notes on nominal calculi for security and mobility. In: Focardi, R., Gorrieri, R. (eds.) FOSAD 2000, LNCS, vol. 2171, pp. 262–330. Springer, Berlin (2001)
26. Grigore, R., Distefano, D., Petersen, R.L., Tzevelekos, N.: Runtime verification based on register automata. In: Tools and Algorithms for the Construction and Analysis of Systems—19th International Conference, TACAS 2013, Lecture Notes in Computer Science, vol. 7795, pp. 260–276. Springer, Berlin (2013)
27. Grumberg, O., Kupferman, O., Sheinvald, S.: Variable automata over infinite alphabets. In: Dediu, A.H., Fernau, H., Martín-Vide, C. (eds.) LATA, LNCS, vol. 6031, pp. 561–572. Springer, Berlin (2010)
28. Jensen, T.P., Métayer, D.L., Thorn, T.: Verification of control flow based security properties. In: 1999 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 9–12, 1999, pp. 89–103 (1999)
29. Kaminski, M., Francez, N.: Finite-memory automata. TCS **134**(2), 329–363 (1994)
30. Kaminski, M., Zeitlin, D.: Finite-memory automata with non-deterministic reassignment. Int. J. Found. Comput. Sci. **21**(5), 741–760 (2010)
31. Kurz, A., Suzuki, T., Tuosto, E.: On nominal regular languages with binders. In: Birkedal, L., (ed.) FOSSACS 2012, LNCS, vol. 7213, pp. 255–269. Springer, Berlin (2012)
32. Kurz, A., Suzuki, T., Tuosto, E.: Nominal regular expressions for languages over infinite alphabets. Extended abstract (2013). arXiv:1310.7093
33. Manuel, A., Muscholl, A., Puppis, G.: Walking on data words. In: Mogens, N., Branislav, R. (eds.) Computer Science-Theory and Applications, pp. 64–75. Springer, Berlin (2013)
34. Mezzetti, G.: Nominal context-free behaviour. Ph.D. thesis, University of Pisa (2014)
35. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs, NJ (1967)
36. Montanari, U., Pistore, M.: π-calculus, structured coalgebras, and minimal hd-automata. In: Mogens, N., Branislav, R. (eds.) MFCS 2000, LNCS, vol. 1893, pp. 569–578. Springer, Berlin (2000)
37. Murawski, A.S., Ramsay, S.J., Tzevelekos, N.: Reachability in pushdown register automata. In: Csuhaj-Varjú, E., Dietzfelbinger, M., Ésik, Z. (eds.) Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25–29, 2014. Proceedings, Part I, Lecture Notes in Computer Science, vol. 8634, pp. 464–473. Springer, Berlin (2014). doi:10.1007/978-3-662-44522-8_39
38. Neven, F., Schwentick, T., Vianu, V.: Towards regular languages over infinite alphabets. Math. Found. Comput. Sci. **2001**, 560–572 (2001)
39. Neven, F., Schwentick, T., Vianu, V.: Finite state machines for strings over infinite alphabets. ACM Trans. Comput. Log. (TOCL) **5**(3), 403–435 (2004)
40. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, 1st ed. 1999. corr. 2nd printing, 1999 edn. Springer, Berlin (2005)
41. Papazoglou, M.P.: Web Services—Principles and Technology. Prentice Hall, Englewood Cliffs (2008). http://vig.pearsoned.com/store/product/1%2C1207%2Cstore-12521_isbn-0321155556%2C00.html

42. Parys, P.: Higher-order pushdown systems with data. In: Faella, M., Murano, A. (eds.) GandALF, EPTCS, vol. 96, pp. 210–223 (2012)
43. Perrin, D., Pin, J.: Infinite words: automata, semigroups, logic and games. Pure Appl. Math. **141** (2004)
44. Pitts, A.M., Stark, I.D.B.: Observable properties of higher order functions that dynamically create local names, or what's new? In: Andrzej, M.B., Stefan, S. (eds.) MFCS 1993, LNCS, vol. 711, pp. 122–141. Springer, Berlin (1993)
45. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. (TISSEC) **3**(1), 30–50 (2000)
46. Skalka, C., Smith, S.F., Horn, D.V.: Types and trace effects of higher order programs. J. Funct. Program. **18**(2), 179–249 (2008)
47. Tzevelekos, N.: Fresh-register automata. ACM SIGPLAN Not. **46**(1), 295–306 (2011)
48. Tzevelekos, N., Grigore, R.: History-register automata. In: Pfenning, F. (ed.) Foundations of Software Science and Computation Structures - 16th International Conference, FOSSACS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16–24, 2013. Proceedings, vol. 7794, pp. 17–33. Springer, Berlin (2013)
49. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: LICS, pp. 332–344. IEEE Computer Society (1986)