Learning Nominal Regular Languages with Binders

Thesis submitted for the degree of

Doctor of Philosophy

at the University of Leicester

by

Yi Xiao

Department of Informatics

University of Leicester

2019

Abstract

The theories of formal language and automata are fundamental in several areas of computer science. During decades of development, these theories have been stretched out and reach many branches and application contexts, ranging from lexical analysis, natural language processing, model checking, and system design. Recently, the applications of machine learning are spreading out rapidly. One interesting application, learning automata, gains sustained attention crossing the disciplines.

This dissertation investigates learning *nominal automata*, an extension of classical automata to alphabets featuring names. This class of automata capture *nominal regular languages*; analogously to the classical language theory, nominal automata have been shown to characterise *nominal regular expressions with binders*. These formalisms are amenable to abtract modelling resource-aware computations.

We propose $\mathsf{nL}^\star$, a learning algorithm on nominal regular languages with binders. Our algorithm generalises Angluin's algorithm $\mathsf{L}^\star$ with respect to nominal regular languages with binders. We show the correctness of $\mathsf{nL}^\star$ and study its theoretical complexity.

We also develop a implementation of $\mathsf{nL}^\star$ that we use to experimentally analyse different strategies for providing counterexamples to the learner. These strategies are designed on the base of the rich algebraic structure provided by our nominal setting. Further, we evaluate the behaviours of the process and its efficiency according to the different strategies with a set of experiments.

Acknowledgements

With no doubt, I press my most gratitude to Dr. Emilio Tuosto, my supervisor, for his patient guidance, enthusiastic encouragement, and critical advice of research work. I would like to extend my grateful thanks to Prof. Alexander Kurz for his guide and help in doing the theoretical research. The first lesson about theory research was taught by him. And, for a long time, Alexander gave me an incredible amount of things before he left the department. I want also thank Dr. José Miguel Rojas for his advice in testing technology.

Last but not least, I wish to thank my parents for their support and encouragement throughout my study.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis combines two areas of computer science, *nominal languages* and *learning automata*. Formal languages and (nominal) automata theories are fundamental branches of computer science to model and solve computational and logical problems. They play an important role in system design, computation, artificial intelligence and formal verification [24, 15, 17, 34].

In recent years, the applications of machine learning grew widely and rapidly. In this context, learning automata have increasingly gained attention. This thesis is devoted to the study of a learning algorithm for the regular nominal languages. We take inspiration from the $\mathsf{L}^\star$ algorithm of Angluin (also reviewed in Chapter 2).

As we will see, the design of the algorithm requires some ingenuity and opens up the possibility of interesting investigations due to richer structure brought in by names and name binding.

Besides the theoretical results, we also experimentally investigate the properties of the algorithm. For this we provide a prototype implementation and use it to analyse how the algebraic nominal structure of the languages and the strategies to "teach" the learner impact on efficiency.

1.1 Motivations

Before contextualising our results and discussing our approach with respect to the literature, we spell out the key ingredients and motivations of our work. Computation is intimately connected with *resource awareness*. Algorithms, programs, or protocols can hardly be useful if they do not carefully deal with computational resources such as data structures or memory, channels, devices, etc. Here, we do not restrict ourselves to a specific type of resources; rather we think of resources in a very abstract and general sense. We use *names* as models of resources and (abstract) operations on names as developed in *nominal* languages with binders (see Section 2.4 for an overview) as mechanisms to capture basic properties of resources: we focus on the *dynamic allocation and deallocation* of resources. More precisely, we take inspiration of binders with dynamic scoping of nominal languages in an operational context based on finite state *nominal* automata. The states of these automata have transitions to explicitly (i) allocate names, corresponding to *scope extrusion* of nominal languages, and (ii) to deallocate names corresponding to *garbage collection* of unused names.

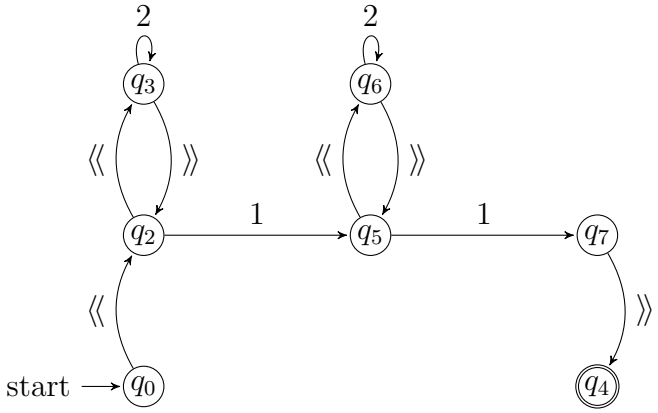
Our theory sets in the context of *nominal regular expressions* for which it has been proved a nice result that transfers the tradition Kleene theorem to the nominal case. In fact, the class of nominal languages that we consider can be characterised as those accepted by some nominal automata or, equivalently, that can be generated by *nominal regular expressions*. The latter algebraic presentation (that we borrow from the literature and review in Chapter 2) features, besides the usual operations of regular expressions (union, concatenation, Kleene-star), a name binding mechanism and a special *resource-aware* complementation operation. Our results rely on the closure properties of this class of languages that has been already demonstrated in the literature.

A main motivation for this proposal is the abstract characterisation of basic features of computations *with resources*. For instance, nominal automata have applications to the verification of protocols and systems [16, 17]. Other approaches to verify resource-aware computations have also been based on automata models [3, 12, 13] employ usage automata (UA) to express and model check patterns of resource-usage.

A distinguishing feature of the approach in [31, 32] is that allocation and deallocation of resources is abstracted away with *binders*. Inspired by the *scope extrusion* mechanism of the π-calculus, the allocation of a resource corresponds to an (explicit) operation that introduces a fresh name; likewise, the deallocation of a resource corresponds to an (explicit) operation to "free" names. We illustrate this idea with an example. Consider the following expression

$$\hat{E} = \langle n.\langle m.m^\star\rangle^\star n\langle k.k^\star\rangle^\star n\rangle$$

which is a nominal regular expression where n, m, k are names, $\_^\star$ is the usual Kleene-star operation, and subexpressions of the form $\langle n.E\rangle$ represent the binding mechanism whereby name n is bound (that is "local") to expression E. Intuitively, $\hat{E}$ describes a language of words starting with the allocation of a freshly generated name, conventionally denoted n, followed by the words generated by the subexpression $\langle m.m\rangle^\star$ post-fixed by n, and so on. Note that in $\hat{E}$ name m occurs in a nested binder for name n. According to [31], $\hat{E}$ corresponds to the following nominal automaton:



which from the initial state q_0 allocates a fresh name through the transition labelled $\langle\langle$ to state q_2. Notice how bound names are rendered in the nominal automaton: they are concretely represented as (strictly positive) natural numbers. This allows us to abstract away from the identities of bound names. In fact, the identity of bound names is immaterial and can be *alpha-converted*, that is replace with any other name provided that the name has not been used already. The use of numbers enables a simple "implementation" of alpha-conversion. More precisely, think of numbers as being addresses of registers of states. For instance, q_3 has 2 registers addressed by 1

and 2 respectively. Then the self-loop transition in state q_3 can consume any name n, provided that n is different than the name (currently) stored in register 1. Finally, note that the content of registers is local to states; once a deallocation transition $\rangle\!\rangle$ is fired, the content in last allocated register is disregarded.

For a practical example, we consider a scenario based on servers to show how nominal regular languages with binders can suitably specify usage policies of servers $S_1, \ldots, S_k$, such that, $\forall 1 \leq h \leq k$ S_h offers operations $\{o_{h_1}, \ldots, o_{h_k}\} = O_h$.

Given an alphabet $\Sigma = \bigcup_{h=1}^{k} \{l_{i_h}, l_{o_h}\} \cup O_h$ where symbols l_{i_h}, l_{o_h} represent basic input and output activities consider the regular expressions on Σ

$$E_h = (l_{i_h}\langle s.e_h l_{o_h}\rangle)^* \qquad e_h = (\sum_{o \in O_h} o.e_o)^*$$

Name s is a fresh session identifier which the symbol $\langle$ allocates when the session starts and the symbol $\rangle$ deallocates when the session ends. Note that s may occur in e_h to e.g., avoid re-authentication. Resources (activities, sessions, operations, etc.) can be abstracted as letters and names, and the (de)allocation represent the binding and freshness conditions. Intuitively, we give the following nominal regular expressions with $\Sigma = \bigcup_{h=1}^{k} \{l_{i_h}, l_{o_h}\} \cup \{\text{readFead}, \text{updateProfile}\}$ and $n_1 \neq n_2$ be distinct names. Operations readFeed, updateProfile allow users read a feed and to update a profile.

$$E_1 = (l_{i_1}\langle n_1.e_1 l_{o_1}\rangle)^* \quad e_1 = (n\ \text{readFeed} + \text{updateProfile}\ E_2)^* \quad E_2 = (l_{i_2}\langle n_2.e_2 l_{o_2}\rangle)^*$$

From the above equations, we see clearly the binders delimit the scope of n_1 and n_2. And, n_2 is nested in n_1. Intuitively, the approach above relaxes the condition of classical language theory that the alphabet of a language is constant. Binder allow us to extend the alphabet "dynamically".

Thinking about a real-life problem, such as operators of a bank account, we have a sequence of actions $o = \mathit{openAccount}(\text{“Yi”});\ \mathit{deposit}(o, 110);\ \mathit{withdraw}(o, 50);$ $\mathit{closeAccount}(o)$. We can transfer these actions into a model with a dynamical

range. First, the actions openAccount and closeAccount delimit the scope of an account. Within an account, the customer can deposit and withdraw money. The parameters "Yi", 110 and 50 are flexible. We could use names to present those parameters. Because the signification of these parameters is their classification not their value. Thus, we can express the sequence of actions as the following nominal regular expression with $\Sigma = \{openAccount, closeAccout, deposit, withdraw\}$ and $\{o, n_1, n_2\}$ be names.

$$\langle o.o\ openAccount\langle n_1.n_1\ deposit\rangle\langle n_2.n_2\ withdraw\rangle o\ closeAccount\rangle$$

Certainly, the scope of withdraw could be nested in the scope of deposit in case of an account which cannot be overdrawn. Changing the scopes of binders is a convenient way to classify a new authority. We discuss the details in Section 2.4.

1.2 Related Works

The contributions of this dissertation are combining two research areas: nominal automata and machine learning, that is, Angluin's learning algorithm.

Nominal Automata The pioneering work on languages on infinite alphabet is [28], and then, many automata models for infinite alphabets have been developed, such as finite memory automata [28], automata with pebbles [43], alternating register automata [14] and data automata [5]. And, the theory of nominal languages has been advocated as a suitable abstraction for computations *with resources* emerging from the so-called nominal calculi which bred after the seminal work introducing the π-calculus [38, 37, 51]. Abstract theories capturing the computational phenomena in this context have been developed in [20, 21, 19] in parallel with a theory of nominal automata [40, 18, 48]. The formal connections between these theories have been unveiled in [22]. Later, [31] proposed the notion of nominal regular languages and the use of nominal automata as acceptors of such languages.

This is strongly related to other approaches in the literature, where languages over infinite alphabets are considered. A form of regular expressions, called UB-

expressions, for languages on infinite alphabets investigated in [29]. In [53] pebble automata are compared to register automata. This class of languages are not suitable for our purposes as they do not account for freshness.

As observed in [31, 4] are not suitable to handle name binding as registers are 'global'; the nominal model in [4] is instead closer (see also the comment below Example 11.4 of [4]) to history dependent automata [48], which are also the inspiration for the model of automata in [31]. The nominal automata in [4] are (abstractions of) deterministic HD-automata (which can be seen as 'implementation' of orbit-finite nominal automata following the connection between nominal and named set of [22]). The nominal automata in [4] are based on orbit finite nominal sets in an arbitrary data symmetry. The basic sets are called G-set and the classical automaton to G-set is defined as G-automaton under some fixed data symmetry $(\mathbb{D}, G)$. Their nominal research has to be restricted on some class of well-structured G-sets since the orbit finiteness cannot be preserved in full generality, especially the Cartesian product.

Definition 1.2.1. *[4] A (right) action of a group G on a set X is a function $\cdot : X \times G \to X$, written infix, subject to axioms*

$$x \cdot e = x \qquad x \cdot (\pi\sigma) = (x \cdot \pi) \cdot \sigma$$

for $x \in X$ and $\pi, \sigma \in G$, where e is the neutral element of G. A set equipped with such an action is called a G-set.

Definition 1.2.2. *[4] A data symmetry $(\mathbb{D}, G)$ is a set $\mathbb{D}$ of data, together with a subgroup $G \leq Sym(\mathbb{D})$ of the symmetric group on $\mathbb{D}$.*

Definition 1.2.3. *[4] A set $C \subseteq \mathbb{D}$ supports an element $x \in X$ if $x \cdot \pi = x$ for all $x \in G$ that act as identity on C. A G-set is nominal in the symmetry $(\mathbb{D}, G)$ if every element of it has a finite support.*

Definition 1.2.4. *[4] A nondeterministic G-automaton consists of*

- *an orbit finite G-set A, called the input alphabet,*
- *a G-set Q, the set of states,*
- *equivariant subsets $I, F \subseteq Q$ of initial and accepting states,*

- *an equivariant transition relation* $\delta \subseteq Q \times A \times Q$.

The automaton is orbit finite if the set of states Q is so. An G-automaton is nominal if both the alphabet A and the state space Q are nominal G-set.

Unfortunately, the G-automata fail in determinisation and are not closed under complementation. But, they have pursued the G-automata for a wide variety of computation models. Coincided with the finite memory automata supports the determinisation. These representation automata is advocated in [6, 10, 39]. For example, as the Figure 1.1, given an infinite alphabet $A = \{a, b, c, d, \dots\}$, the automaton has an orbit finite set of registers/names $D = \{x, y, z\}$ for the alphabet A. Mainly different from [31], x, y, z are global. Without binders, the registers/names last to the end.

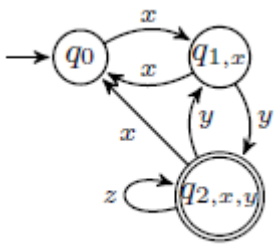


Figure 1.1: An orbit-finite nominal automata example (From Figure 2 in [39]).

This class of automata is more expressive than the classes of automata capturing nominal regular expressions as ours are nominal Kleene algebras [30]. In fact, as noted in [30] this automata accept languages with words having arbitrarily deep nesting of binders. However, orbit-finite nominal automata are not closed under any reasonable notion of complementation [4]. Note that the resource-sensitive complementation operation of [31] is essential in our context. On the other hand, the use of symmetries to capture binding offers a more flexible mechanism to express patterns or words that escape the constraints that the use of 'nested scoping' imposes in our language.

Angluin's Learning Algorithm One of the most known and used learning algorithm is $\mathsf{L}^\star$ introduced by Angluin [1]. Given a regular language, $\mathsf{L}^\star$ creates an

deterministic automaton model from observations that accepts the language. As surveyed in Section 2.3, given a regular language, $\mathsf{L}^\star$ creates a deterministic automaton that accepts the language. This is done by mimicking the "dialogue" between a *learner* and a *teacher*; the former poses questions about the language to the latter. In $\mathsf{L}^\star$ there are two types of queries the learner can ask the teacher: *membership* queries allow the learner to check whether a word belongs to the input language while with *equivalence* queries the learner checks if an automaton accepts or not the language. The automaton is "guessed" by the learner according to the answers the teacher provides to queries. The outcome of an equivalence query may be a *counterexample* selected by the teacher to exhibit that the automaton does not accept the language.

The $\mathsf{L}^\star$ algorithm has been extended to several classes of languages [7, 44, 46]. Using a categorical approach, the $\mathsf{L}^\star$ algorithm has been generalised to other classes of automata such as Moore and Mealy [27]. An interesting line of research is the one explored in [7] which applies learning automata to distributed systems based on message-passing communications to learn communicating finite-state machines [8] from message-sequence charts. Applications of learning automata are in [46] and [44]. The former defines a framework based on $\mathsf{L}^\star$ to fully automatise an incremental assume-guarantee verification technique and the latter proposes an optimised approach for integrated testing of complex systems. Variants of Angluin's algorithm for languages over infinite alphabets have attracted researchers' attention. An $\mathsf{L}^\star$ algoritm for register automata is given in [6] where so-called *session automata* support the notion of fresh data values. Session automata are defined over pairs of finite-infinite alphabets. Interestingly, session automata also have a canonical form to decide equivalence queries.

Like [6], [10] works on register automata and data language but the latter aims to the application of dynamic black-box analysis. The key point is that [10] uses a tree queries instead of membership queries and ensures the observation tables closeness and register-consistency. Further, [10] defines a new version of equivalence to achieve the correctness and termination of the algorithm.

We have introduced the representation of the nominal set and the nominal au-

tomata [4]. The representation of an infinite alphabet [6, 10] is developed in [4] using the Cartesian product of two finite sets since their target languages are more specific in the field of data languages.

Recently, [39] proposes a learning algorithm which extends $\mathsf{L}^\star$ to nominal automata. The main difference between our approach and the one in [39] is the representation of nominal languages and nominal automata. Language theories for infinite alphabets [4, 49] are used in [39], handling names through finitely-supported permutations. Accordingly, in [39] observation tables and the states of nominal automata are orbit-finite. Another difference is the operation on counterexamples. Unlike in $\mathsf{L}^\star$, [39] adds the counterexamples into columns. It is an interesting research direction to optimise our work on the operations of the counterexamples in the future.

1.3 Contributions

This session we briefly describe the contributions of this dissertation associated with the objectives. This dissertation explores the application of machine learning in a class of nominal regular language. More specifically, we are interested in designing a learning algorithm for regular nominal languages with binders. This class of nominal languages has been proposed as an abstract model of *resource-aware* computations in order to use names and operations over them to represent dynamic allocation and deallocation of resources.

Our main achievement is the design of a learning algorithm that generalises Angluin's $\mathsf{L}^\star$ algorithm to nominal regular languages with binders (published in [56]). We call our algorithm $\mathsf{nL}^\star$ (after *nominal* $\mathsf{L}^\star$), as a tribute to Angluin's work. This is attained by retaining the basic scheme of $\mathsf{L}^\star$ (query / response dialogue been a learner and a teacher) and ideas of $\mathsf{L}^\star$ (the representation of a finite state automaton with specific a *observation table*). Technically, this requires a revision of the main concepts of Angluin's theory. That is, we have to reconsider the data structure to represent observation tables and hence the notions of *closedness* and *consistency*. In particular, the type of queries and answers now have to account for names and

the allocation and deallocation operations on them. Thus, we have to refine the automata associated with the observation tables in order to satisfying these queries. Consequently, the counterexamples have an new challenge. The core computation algorithm for the counterexamples is still same as the Angluin's. But, how to choose a proper counterexample is an efficiency challenge.Considering the features of our nominal languages and automata, we state two strategies for finding counterexamples and experiment them.

Interestingly, this revision culminates in Theorem 3.2.6 showing that the nominal automata associated to closed and consistent (nominal) observation tables are minimal. This result highlights the adequacy of our constructions. Finally, we prove the correctness of $\mathsf{nL}^\star$ and analyse its complexity.

Another contributions are the implementation of $\mathsf{nL}^\star$ and an experimental evaluation of $\mathsf{nL}^\star$. These are obtained by developing a prototype implementation of $\mathsf{nL}^\star$ in Java, dubbed **ALeLaB**. Besides realising the data structures and the learning process of $\mathsf{nL}^\star$, our implementation is used to analyse two different strategies for generating counterexamples. These strategies are peculiar to the class of nominal automata we adopt in the dissertation. In fact, as we will see, the teacher can exploit (at least) two different policies when generating counterexamples involving generation of names. In the first policy the teacher tries to minimise the number of fresh names required in a counterexample. In the second policy instead the teacher favours the maximisation of counterexample. We experimentally analyse how these policies affect the convergence of $\mathsf{nL}^\star$.

Besides the evaluation of these policies, we finally experimentally analyse the behaviour of $\mathsf{nL}^\star$ and the different policies on a series of benchmarks according to the algebraic structure of the nominal regular expressions used to represent the input language. The experiments reveal that the use of symbol "P" can reduce a huge amount of time. Interestingly, the strategies have their own advantages in terms of varying operators mixing with binders. The results suggest future lines of work for optimising the learning algorithm.

1.4 Outline

The thesis has five chapters in addition to this introductory one. Chapter 2 introduces the basic theories on which our learning algorithm is developed. Chapter 3 presents the theoretical development of our learning algorithm for the regular nominal languages with binders. Then, Chapter 4 develops a implementation of our learning algorithm in Java. In Chapter 5, we discuss the strategies for counterexamples in terms of the behaviours of learning process. Finally, we summarise the contributions and outline possible future research directions in Chapter 6.

Chapter 2

Background

In this chapter we survey the principal ideas necessary for understanding our research. These concepts include regular languages, regular expressions, finite automata, nominal regular expressions and nominal automata. We also review the algorithms [25, 36, 50, 54] for construction of automata and the learning algorithm of Angluin's $\mathsf{L}^\star$ [1].

2.1 Regular Languages

Regular Languages, regular expressions and finite automata have a well-known relationship established by the Kleene Theorem [26]. A regular language can be represented by regular expressions and accepted by a finite state automaton. In this section, we introduce necessary notions and definitions for these concepts.

An *alphabet* is a set. Frequently, the elements of an alphabet are called letters or symbols. We denote a finite alphabet as Σ. A *string* is a sequence of symbols of an alphabet. Let w be a string, we denote the length of w as $|w|$. The string of length zero is called *empty string* and denoted by ϵ. The concatenation of two strings is denoted as $\_ \cdot \_$. A string can concatenate with itself. We define $\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$, where $\Sigma^0 = \{\epsilon\}$ and for each $n > 0$, $\Sigma^n = \{w \cdot w' \mid w \in \Sigma \text{ and } w' \in \Sigma^{n-1}\}$. A *language* L is a set of strings over an alphabet Σ, that is, $L \subseteq \Sigma^*$.

Other language operations we use are concatenation, union, *Kleene-star* and

complementation. Assuming that L and L' are languages over Σ, we have the following:

- concatenation $L \cdot L' = \{w \cdot w' \mid w \in L \textit{ and } w' \in L'\}$,
- union $L \cup L' = \{w \mid w \in L \textit{ or } w \in L'\}$,
- *Kleene-star* $L^* = \bigcup_{n=0}^{\infty} L^n = \begin{cases} \{\epsilon\} & n = 0 \\ L^{n-1} \cdot L & n \neq 0 \end{cases}$,
- complementation $L^C = \{w \in \Sigma^* \mid w \notin L\}$.

Complementation is an important concept which has differences in the nominal setting, as discussed later. These operators provide the mathematical concepts for constructing automata from regular expressions.

Definition 2.1.1 (Regular Expressions)**.** *A* regular expressions over an alphabet Σ *is a term derived from the grammar*

$$re ::= \epsilon | \ \emptyset \mid a \mid re \cdot re \mid re + re | \ re^*$$

where $a \in \Sigma$.

Regular expressions denote languages in a declarative way. Formally, regular expressions are defined as in Definition 2.1.1.We omit the well-know general definition of language of a regular expression. We denote the language of a regular expression re as $\mathcal{L}(re)$. Given two regular expressions re and re', $re + re'$ denotes the union of re and re', $re \cdot re'$ denotes the concatenation of re and re', and re^* denotes the Kleene-star of re. The order of precedence for the operators is important and parentheses are used to group operands where needed [26]. Roughly, the rules of precedence is as follows.

- The star operator * is highest precedence.
- Next in precedence is the concatenation operator $\cdot$.
- Finally, it is union operator $+$.

They are related to the core technique of converting a regular expression to an automaton. Especially, we should derive them to be proper for nominal regular expression.

Note that regular expression $\emptyset$ represents the language $\mathcal{L}(\emptyset) = \emptyset$. Regular expression ϵ represents the language $\mathcal{L}(\epsilon) = \{\epsilon\}$. Regular expression a represents the language $\mathcal{L}(a) = \{a\}$ accepting word a. In Example 2.1.2 below, we give some simple expressions to show the relationship between regular expressions and regular languages.

Example 2.1.2. Let $\Sigma = \{a, b\}$ be an alphabet, we give some examples showing regular expressions and associated languages.

- $a + b$ denotes the language $\{a, b\}$.
- $a \cdot b$ denotes the language $\{ab\}$.
- a^* denotes the language $\{\epsilon, a, aa, aaa, \dots\} = \{a^n \mid n \geq 0\}$.
- $(a + b)^*b$ denotes the language of strings ending with b.

Besides, the Myhill-Nerode Theorem [42] provides another way to proof languages to be regular. Regular languages can be visualised as graphs of finite nodes and edges, called automata. We denote the language of an automaton M as $\mathfrak{L}(M)$. We define finite automata formally as Definition 2.1.3.

Definition 2.1.3 (Finite Automata)**.** *A* finite automaton over alphabet Σ *is a five-tuple* $M = \langle Q, q_0, F, \delta \rangle$ *such that*

- Q *is a finite set of states,*
- q_0 *is the initial state,*
- $F \subseteq Q$ *is the finite set of final states,*
- $\delta : Q \times \Sigma \times Q$ *is a relation from states and alphabet symbols to states.*

Theorem 2.1.4. *[26] A language L is regular if and only if there exists a finite automaton M such that $L = \mathfrak{L}(M)$.*

Normally, we represent finite automata by directed graphs. We use different nodes distinguishing initial state, ordinary states, and final states. The edges with labels of the graph represent transitions with symbols. As Example 2.1.9 shows, an initial state is a single-cycle node with an arrow toward to it from nowhere, a normal state is a single-cycle node, a final state is a double-cycle node and in the case of initial state and final state is the same state, the state is represented as a double-cycle node with an arrow toward to it from nowhere. Transitions are arrows from a state to a state.

Definition 2.1.5 ($\mathrm{run}_M(w, q)$)**.** *Let* $M = (Q, \Sigma, \delta, q_0, F)$ *and* $w = a_1 \dots a_n$ *be a string over* Σ*. Then* $\mathrm{run}_M(\lambda, q)$ *is the sequence* $[q]$ *and* $\mathrm{run}(aw, q) = [q] \cdot \mathrm{run}(w, \delta(q, a))$*. We say that* q *is* reachable *if there is* w *such that* $\mathrm{last}(\mathrm{run}(w, q_0)) = q$*.*

Here, we write [...] for lists and use $\cdot$ for concatenation of lists. The function last takes a non-empty list to its last element.

Definition 2.1.6 (Language accepted by a state)**.** *Let* $M = (Q, \Sigma, \delta, q_0, F)$ *and let* $q \in Q$*. We define the language* $[\![q]\!]$ *as*

$$[\![q]\!] = \{w \in \Sigma^* \mid \mathrm{run}_M(w, q) = q\, q_1 \dots q_n, q_n \in F\}$$

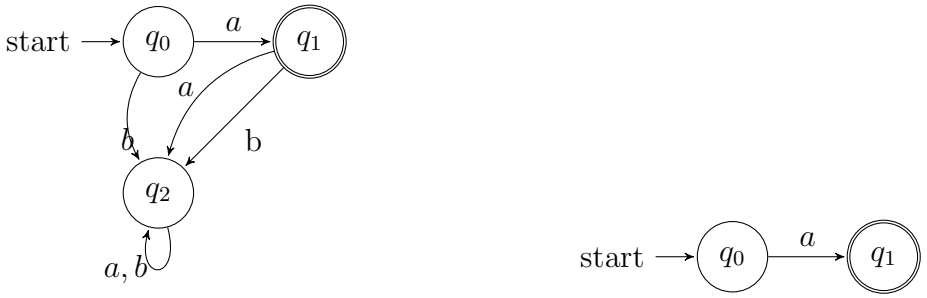
We also write $\mathcal{L}(M, q)$ *for* $[\![q]\!]$ *and call it the language accepted by* q *in* M*.*

Definition 2.1.7 (Equivalence)**.** *Two finite automata are* equivalent *if and only if they accept the same language.*

Theorem 2.1.8. *[26] For every regular language* L*, there exists a minimal finite automaton* M *accepting* L *and* M *is unique.*

A finite automaton is *deterministic* if and only if δ is a function $Q \times \Sigma \to Q$. In contrast, a *non-deterministic* finite automaton has no restrictions for δ. If a deterministic finite automaton and a non-deterministic finite automaton are equivalent, they accept a same language. Furthermore, Theorem 2.1.8 gives a theoretical support to the correctness of the learning automata accepted the same language as expected. Example 2.1.9 shows a deterministic finite automaton and a non-deterministic finite automaton which are equivalent.

Example 2.1.9. Let $\Sigma = \{a, b\}$ be an alphabet and $L = \{a\}$ be a language over Σ. A non-deterministic finite automaton and a deterministic finite automaton accepting L are display as Figure.



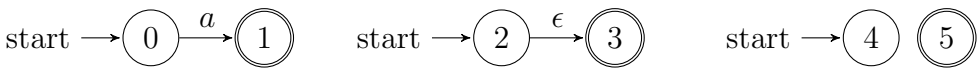
(a) DFA (b) NFA

2.2 Constructing Automata from Regular Expressions

In the implementation of our algorithm, we need to construct automata from nominal regular expressions and compare their equivalence. To do that, we review a few definitions and concepts.

First, we review the McNaughton-Yamada-Thompson algorithm, a method of converting a regular expression to a non-deterministic finite automaton. Since first stated in [54], Thompson's algorithm is widely used in combination with the McNaughton and Yamada algorithm [36]. The algorithm parses and splits a regular expression into its constituent subexpressions, and then constructs non-deterministic automata through a set of laws. We give an informal presentation of Thompson's algorithm.

The base cases are in Figure 2.2. And, we illustrate the other cases with Figure 2.3.



(a) $re = a$ (b) $re = \epsilon$ (c) $re = \emptyset$

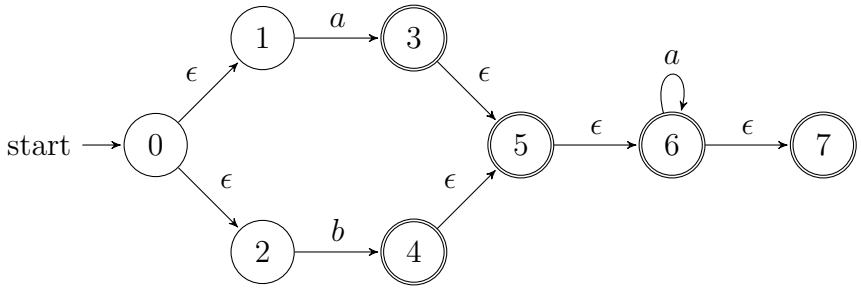
Figure 2.2: Automata representing base cases

The concatenation of expressions, as Figure 2.3a, is to merge the final state of the first automaton and the initial state of the second automaton. The initial state of the first automaton is the initial state of the completed automaton. The final state of the second automaton is the final state of the completed automaton.

The union of the expressions, as Figure 2.3b, needs two additional states to complete the whole automaton. One is the new initial state which goes via ϵ either to the initial state of first automaton or second automaton. Another one is the new final state. The final states of first automaton or second automaton become intermediate states of the completed automaton and goes via ϵ to the new final state.

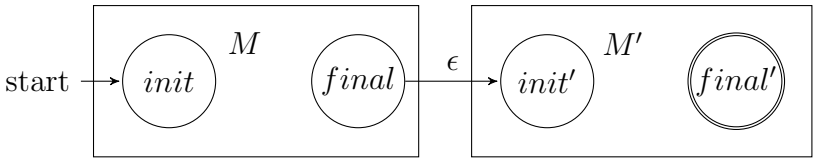
The Kleene-star of the expressions, as Figure 2.3c, also needs two additional states to complete the whole automaton. One is the new initial state which goes via ϵ either to the old initial state. Another state is the new final state. The old final states become intermediate states of the completed automaton and go via ϵ to the new final state. Moreover, there is a transition from the new initial state to the new final state via ϵ and transitions from the old final states to the old initial state via ϵ. We apply this construction in the next Example 2.2.1.

Example 2.2.1. Given a regular expression $re = (a + b)a^*$ over an alphabet $\Sigma = \{a, b\}$, the result of Thompson's construction on it is as following.

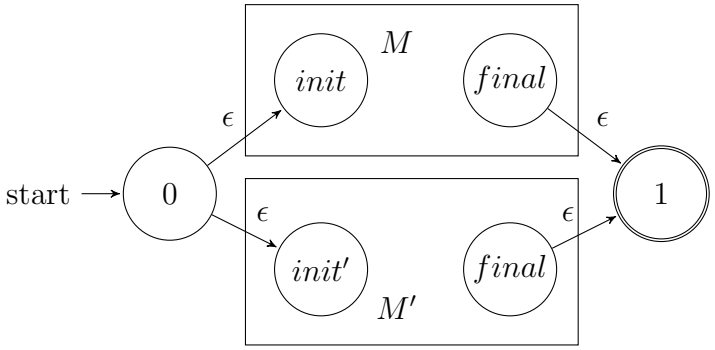


We need to deal with deterministic automata. Thus, we have another algorithm to convert it into a deterministic finite automaton. We use the powerset construction algorithm [50]. The algorithm computes the ϵ-closures [45] of the entire automata. The deterministic finite automaton consists of the reachable ϵ-closures.

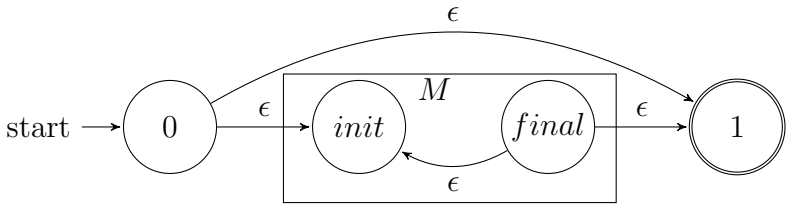
Example 2.2.2 show a deterministic automaton converted from the non-deterministic automaton in Example 2.2.1. The first set for the deterministic automaton is constructed from all states in non-deterministic automaton that are reachable from



(a) Construction for concatenation.



(b) Construction for union.

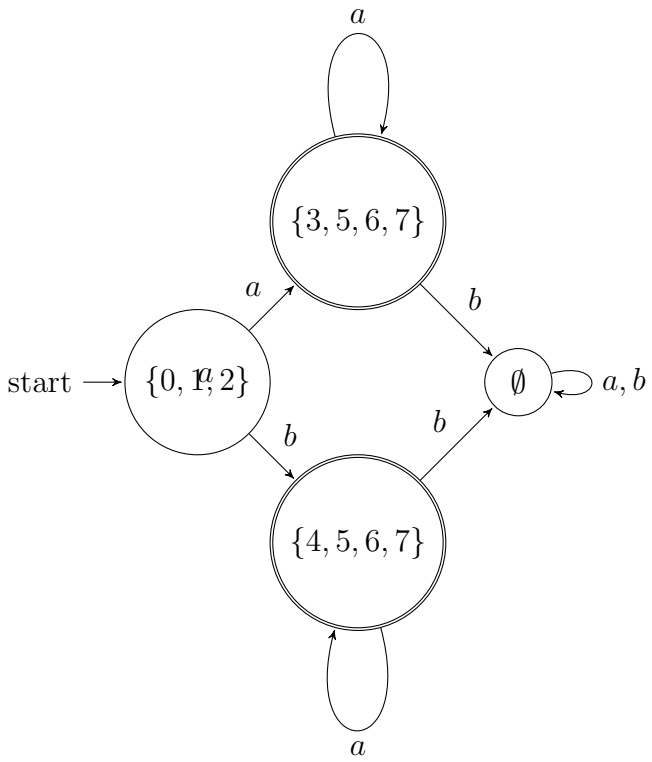


(c) Construction for Kleene-star.

Figure 2.3: Constructions for regular expressions with operators.

state 0 by ϵ-transitions. It is the set $0, 1, 2$, that is, constructing the initial state of deterministic automaton. A transition from $0, 1, 2$ by symbol a follows the transition from state 1 to state 3. Furthermore, state 3 has a ϵ-transition to state 5. Thus, state 3 and state 5 are in a ϵ-closure. Continuing checking ϵ-transitions, the ϵ-closure is the set $3, 5, 6, 7$. Thus, the second state of deterministic automaton is constructed for the set $3, 5, 6, 7$ and a transition from $0, 1, 2$ with a. And by the same seasoning the fully deterministic automaton is as shown in figure of Example 2.2.2.

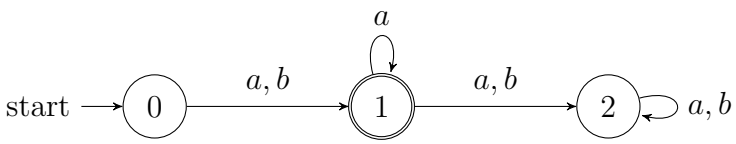
Example 2.2.2. Associated to Example 2.2.1, we convert the non-deterministic automaton into a deterministic automaton shown as following.



We also have to compare two automata for equivalence. To deal with that, we use Theorem 2.1.8 to decide the equivalence of two deterministic finite automata under their minimisation. There are several difference algorithms to minimise a deterministic finite automaton, such as Hopcroft's algorithm [25], Moore's algorithm [41] and Brzozowski's algorithm [9]. In our research, we use Hopcroft's algorithm which based on partition refinement.

A minimal deterministic finite automaton for the automaton in Example 2.2.2 is shown in Example 2.2.3. Every state in Example 2.2.2 is partitioned into groups by equivalence relation. First, the finial states and other states are distinguished. That is, states $\{3, 5, 6, 7\}$ and $\{4, 5, 6, 7\}$ are in a same group and others are in another group. Then, the states $\{3, 5, 6, 7\}$ and $\{4, 5, 6, 7\}$ are equivalent and stay in a group because they have same behaviours with all the input sequence. By the same reasoning, all states are partitioned, and finally, the groups represent the states for the minimal automaton. In order to save space, we rename the labels for states.

Example 2.2.3. Associated to Example 2.2.2, we minimise the deterministic finite automaton shown as following.



2.3 Angluin's Algorithm $\mathsf{L}^\star$

In this section, we review the algorithm $\mathsf{L}^\star$ introduced in [1] which learns a finite automaton accepting a given regular language L over Σ. The basic idea of the algorithm is to implement a dialogue between a "Learner" and a "Teacher". The Learner may ask the Teacher for *membership queries* "$w \in L$?" to check whether a word w is in the given language. Moreover, the Learner may submit an automaton M to the Teacher who replies "yes" if the language $\mathfrak{L}(M)$ is equivalent to L, or, replies "no" together with a counter-example showing that $\mathfrak{L}(M) \neq L$. The Teacher is assumed to answer all the Learner's questions correctly.

Observation tables are a key data structure in $\mathsf{L}^\star$. An observation table is a finite collection of strings over Σ, classifying them as members of L or not.

Definition 2.3.1 (Observation Tables)**.** *An* observation table (S, E, T) *consists of nonempty finite languages $S, E \subseteq \Sigma^*$ such that S is prefix-closed and E is suffix-closed, and a function $T : (S \cup S \cdot \Sigma) \cdot E \to \{0, 1\}$.*

The rows of an observation table are labelled by elements of $S \cup S \cdot \Sigma$, and the columns are labelled by elements of E with the entry for row s and column e given by $T(s \cdot e)$. A row of the table can be represented by a function $row(s) : E \to \{0, 1\}$ such that $row(s)(e) = T(s \cdot e)$. A string $s \cdot e$ is a member of L of (S, E, T) iff $T(s \cdot e) = 1$.

Definition 2.3.2 (Closed and Consistent Tables)**.** *An observation table (S, E, T) is* closed *when*

$$\forall w \in S \cdot \Sigma \,.\, \exists s \in S \,.\, row(w) = row(s)$$

An observation table (S, E, T) is consistent *when for all $a \in \Sigma$ and all $s, s' \in S$*

$$row(s) = row(s') \implies row(sa) = row(s'a).$$

A closed and consistent observation table induces a finite automaton as following.

Definition 2.3.3. *The automaton* $M = (Q, \delta, q_0, F)$ associated to a closed and consistent observation table (S, E, T) *is given by*

- $Q = \{row(s) \mid s \in S\}$,
- $q_0 = row(\epsilon)$,
- $F = \{row(s) \mid row(s)(\epsilon) = 1, s \in S\}$,
- $\delta(row(s), a) = row(s \cdot a),\ a \in \Sigma$.

To see that this is a well-defined automaton, note that initial state is defined since S is prefix-closed and must contain ϵ. Similarly, E is suffix-closed and must contain ϵ. And, if $s, s' \in S, row(s) = row(s')$, then $T(s) = T(s \cdot \epsilon)$ and $T(s') = T(s' \cdot \epsilon)$ are equal as defined. So, the set of final states is well-defined. The transition function is well-defined since the table is closed and consistent. Suppose s and s' are elements of S such that $row(s) = row(s')$. Since the table (S, E, T) is consistent, $\forall a \in \Sigma, row(sa) = row(s'a)$. And the value of $row(sa)$ is equal to such a $row(s'')$ for an $s'' \in S$, since the table is closed.

The learning progress of the Learner is shown in Figure 2.4. We want to learn a language L over an alphabet Σ. With the initialisation of $S = E = \{\epsilon\}$, the observation table (S, E, T) is initialised by asking for membership queries about ϵ and each element in Σ (line 2). Then the algorithm enters into the main loop (lines 3-23). Inside of the main loop, a while loop tests the current observation table (S, E, T) for closedness (line 5) and consistency (line 11).

If (S, E, T) is not closed, the algorithm finds s' in $S \cup \Sigma$ such that $row(s')$ is different from $row(s)$ for all $s \in S$. Then the string s' is added into S and new rows are added for strings $s' \cdot a$ for all $a \in \Sigma$. Thus, T is extended to $(S \cup S \cdot \Sigma) \cdot E$ by asking for membership queries about missing elements.

Similarly, if (S, E, T) is not consistent, the algorithm finds $s_1, s_2 \in S$,$e \in E$, and $a \in \Sigma$ such that $row(s_1) = row(s_2)$ but $row(s_1 \cdot a)(e) \neq row(s_2 \cdot a)(e)$. The string $a \cdot e$ is added into E. That is, each row in the table has a new column $a \cdot e$. T is extended to $(S \cup S \cdot \Sigma) \cdot E$ by asking for missing elements $row(s)(a \cdot e)$ for all $s \in (S \cup S \cdot \Sigma)$.

1: Initialisation: $S = \{\epsilon\}, E = \{\epsilon\}$.
2: Construct the initial observation table (S, E, T) by asking for membership queries about $(S \cup S \cdot \Sigma) \cdot E$.
3: **repeat**
4: **while** (S, E, T) is not closed or not consistent **do**
5: **if** (S, E, T) is not closed **then**
6: find $s' \in S \cdot \Sigma$ such that
7: $row(s) \neq row(s')$ for all $s \in S$,
8: add s' into S,
9: extend T to $(S \cup S \cdot \Sigma) \cdot E$ using membership queries.
10: **end if**
11: **if** (S, E, T) is not consistent **then**
12: find $s_1, s_2 \in S$,$e \in E$ and $a \in \Sigma$ such that
13: $row(s_1) = row(s_2)$ and $row(s_1 \cdot a)(e) \neq row(s_2 \cdot a)(e)$,
14: Add $a \cdot e$ into E,
15: extend T to $(S \cup S \cdot \Sigma) \cdot E$ using membership queries.
16: **end if**
17: **end while**
18: Construct an automaton M from table (S, E, T) and ask Teacher an equivalence query.
19: **if** Teacher replies a counterexample c **then**
20: add c and all its prefixes into S.
21: extend T to $(S \cup S \cdot \Sigma) \cdot E$ using membership queries.
22: **end if**
23: **until** Teacher replies yes to equivalence query M.
24: Halt and output M.

Figure 2.4: The Learner in $\mathsf{L}^\star$.

An associated automaton M is constructed when the observation table (S, E, T) is closed and consistent. And then, an equivalence query about M is asked for. The algorithm terminates and outputs M when the Teacher replies "yes" to the query. If the Teacher replies with a counterexample c, the string c and all its prefixes are added into S, and then T is extended by asking membership queries about new entries in $(S \cup S \cdot \Sigma) \cdot E$. A new round for the main loop of closedness and consistency starts. We give an example of how the algorithm works in Example 2.3.4.

Example 2.3.4. Let $\Sigma = \{a, b\}$. We apply Angluin's algorithm to learn the language $\mathcal{L}((a + b)a^*)$.

Step 1

In the first step, we initialise an observation table for $S_1 = \{\epsilon\}$ and $E_1 = \{\epsilon\}$, using membership queries:

T_1=

	ϵ
ϵ	0
a	1
b	1

(S_1, E_1, T_1) consistent? There is only one element in S, thus the table is consistent.

(S_1, E_1, T_1) closed? No, $row(a) \neq row(\epsilon)$.

So, S_1 should be extended and we go to Step 2.

Step 2

Let $S_2 = S_1 \cup \{a\}$ and $E_2 = E$ and then construct a new observation table (S_2, E_2, T_2) through membership queries.

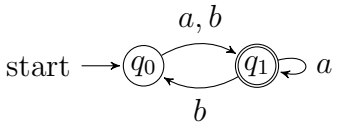
T_2=

	ϵ
ϵ	0
a	1
b	1
aa	1
ab	0

(S_2, E_2, T_2) closed? $\surd$

(S_2, E_2, T_2) consistent? $\surd$

Then, we compute the automaton:



Teacher replies with a counterexample, say, bba ([1] chooses a counterexample randomly). Note that $T_2(bba) = 1$ but $aaa \notin \mathcal{L}((a+b)a^*)$. We go to Step 3.

Step 3

To deal with the counterexample,let $S_3 = S_2 \cup \{b, bb, bba\}$ and $E_3 = E_2$. Then we construct a new observation table (S_3, E_3, T_3) through membership queries. Then we check the new table for closeness and consistency.

T_3=

	ϵ
ϵ	0
a	1
b	1
bb	0
bba	0
ab	0
aa	1
ba	1
bbb	0
$bbaa$	0
$bbab$	0

(S_3, E_3, T_3) consistent?

No, $row(\epsilon) = row(bb)$ but $row(a) \neq row(bba)$.

(S_3, E_3, T_3) closed? $\surd$

So, E_3 should be extended and we go to Step 4.

Step 4

Let $E_4 = E_3 \cup \{a\}$ and $S_4 = S_3$, and then construct a new observation table (S_4, E_4, T_4) through membership queries. Then we check the new table for closeness and consistency.

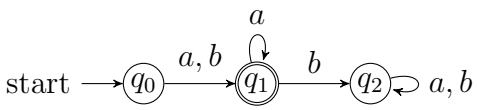
T_4=

	ϵ	a
ϵ	0	1
a	1	1
b	1	1
bb	0	0
bba	0	0
ab	0	0
aa	1	1
ba	1	1
bbb	0	0
$bbaa$	0	0
$bbab$	0	0

(S_4, E_4, T_4) consistent? $\surd$

(S_4, E_4, T_4) closed? $\surd$

From T_4 we obtain the automaton



which is confirmed by the Teacher to be the correct one.

And, indeed, the language of this automaton is $\mathcal{L}((a+b)a^*)$.

2.4 Nominal Languages with Binders

We use the nominal regular expressions introduced in [32, 33]. In the following, we recall the basic notions first and then we survey nominal languages with binders [32].

In computer science and logic, many languages have formulae with variables. In [31, 32, 33] variables called *names* are used to construct words for languages over infinite alphabets. Binders, like the λ in λ-calculus, present and distinguish the scopes of names. For example, [32] use the notation $\langle\langle i.t \rangle\rangle$ to represents the fact that scope of name i is the term t. Hereafter, we fix a countably infinite set of names $\mathcal{N}$.

Definition 2.4.1 (Nominal Languages)**.** *A* nominal word over $\mathcal{N}$ and Σ *is a term derived by the grammar*

$$w ::= \epsilon \mid a \mid i \mid w \cdot w \mid \langle\langle i.w \rangle\rangle$$

where $i \in \mathcal{N},\ a \in \Sigma$. *A* nominal language *is a set of nominal words* w.

A name i is bound in a word when the name is in the binder. For example, the name i is bound in word $\langle\langle i.ia \rangle\rangle$. Otherwise, the name i is free such as in ia.

Definition 2.4.2 (Nominal Regular Expressions)**.** *Given a set of names* $i \in \mathcal{N}$ *and* $a \in \Sigma$, *a* nominal regular expression *is a term derived from the grammar*

$$ne ::= \epsilon |\ \emptyset \mid a \mid i \mid ne \cdot ne \mid ne + ne |\ ne^* \mid \langle i.ne \rangle$$

In the expressions, the binders are in the representation of $\langle i._- \rangle,\ \forall i \in \mathcal{N}$. If the names in a nominal expression are all bound, the nominal expression is closed.

Example 2.4.3. Given $\Sigma = a, b$ and $i, j \in \mathcal{N}$, we have following nominal expressions

- aib, $a\langle j.i \rangle b$ are not closed.
- $a\langle i.i\langle j.i \rangle\rangle b$ is closed.

Definition 2.4.4 shows nominal languages of nominal regular expressions.

Definition 2.4.4. *[32] Let* ne *be a nominal regular expression The* nominal language $\mathcal{L}(ne)$ of ne *is defined as*

$$\mathcal{L}(\epsilon) = \{\epsilon\} \qquad \mathcal{L}(\emptyset) = \emptyset \qquad \mathcal{L}(i) = \{i\} \qquad \mathcal{L}(s) = \{s\}$$

$$\mathcal{L}(ne_1 + ne_2) = \mathcal{L}(ne_1) \cup \mathcal{L}(ne_2)$$

$$\mathcal{L}(\langle i.ne \rangle) = \{\langle\langle i.w \rangle\rangle \mid w \in \mathcal{L}(ne)\}$$

$$\mathcal{L}(ne_1 \cdot ne_2) = \mathcal{L}(ne_1) \cdot \mathcal{L}(ne_2) = \{w \cdot v \mid w \in \mathcal{L}(ne_1), v \in \mathcal{L}(ne_2)\}$$

$$\mathcal{L}(ne^*) = \bigcup_{k \in \mathbb{N}} \mathcal{L}(ne)^k, \text{ where } \mathcal{L}(ne)^k = \begin{cases} \{\epsilon\} & k = 0 \\ \mathcal{L}(ne) \cdot \mathcal{L}(ne)^{k-1} & k \neq 0 \end{cases}.$$

The closure properties of nominal languages are proofed in [32](see the Theorem 2.4.6 below). The main difference with respect to classical regular expressions is complementation. The words in the complementation of $\mathcal{L}(ne)$ may be infinite. Therefore, [32] states a finitary representation to explain and proof the closure properties concretely. They define $\theta(ne)$, the maximum depth of a nominal regular expression ne as

$$ne \in \{\epsilon, \emptyset\} \cup \mathcal{N} \cup \Sigma \implies \theta(ne) = 0$$

$$ne = ne_1 + ne_2 \text{ or } ne_1 \cdot ne_2 \implies \theta(ne) = max(\theta(ne_1), \theta(ne_2))$$

$$ne = \langle i.ne \rangle \implies 1 + \theta(ne)$$

$$ne = ne^* \implies \theta(ne).$$

Nominal regular expressions require a particular operator, dubbed resource sensitive complementation (of Definition 2.4.5). Since these words are not possible to be accepted in a finite automaton, our research considers nominal regular expression closed under resource sensitive complementation.

Definition 2.4.5. *[32] Let ne be a nominal regular expression. The* resource sensitive complementation of $\mathcal{L}(ne)$ *is the set* $\{w \notin \mathcal{L}(ne) \mid \theta(w) \leq \theta(ne)\}$ *(where the depth of a word is defined as the depth the corresponding expression).*

Theorem 2.4.6. *[32] Nominal regular languages are closed under union, intersection, and resource sensitive complementation.*

We now define the notions of nominal automata adopted here. Let $\mathbb{N}$ be the set of natural numbers and define $\underline{n} = \{1, \cdots, n\}$ for each $n \in \mathbb{N}$. Considering a set of states Q paired with a map $\|\_\| : Q \to \mathbb{N}$, let us define the local registers of $q \in Q$ to be $\underline{\|q\|}$. We use a definition of nominal automata [31] as Definition 2.4.7. Moreover, we describe how to allocate names via maps $\sigma : \underline{\|q\|} \to \mathcal{N}$ in Definition 2.4.10. Example 2.4.9 in following presents a nominal automaton for a nominal regular expression.

Definition 2.4.7 (Nominal Automata)**.** *[32] Let $\mathcal{N}_{fin} \subset \mathcal{N}$ be a finite set of names. A* nominal automaton with binders over Σ and $\mathcal{N}_{fin}$*,* $(\Sigma, \mathcal{N}_{fin})$*-automaton for short, is a tuple $M = \langle Q, q_0, F, \delta \rangle$ such that*

- *Q is a finite set of states equipped with a map $\|\_\| : Q \to \mathbb{N}$*
- *q_0 is the initial state and $\|q_0\| = 0$*
- *F is the finite set of final states and $\|q\| = 0$ for each $q \in F$*

- *for each* $q \in Q$ *and* $\alpha \in \Sigma \cup \mathcal{N}_{fin} \cup \{\epsilon, \langle\!\langle, \rangle\!\rangle\}$, *we have a set* $\delta(q, \alpha) \subseteq Q$ *such that for all* $q' \in \delta(q, \alpha)$ *must hold:*

  - $\alpha = \langle\!\langle \implies \|q'\| = \|q\| + 1$
  - $\alpha = \rangle\!\rangle \implies \|q'\| = \|q\| - 1$
  - *otherwise* $\implies \|q'\| = \|q\|$

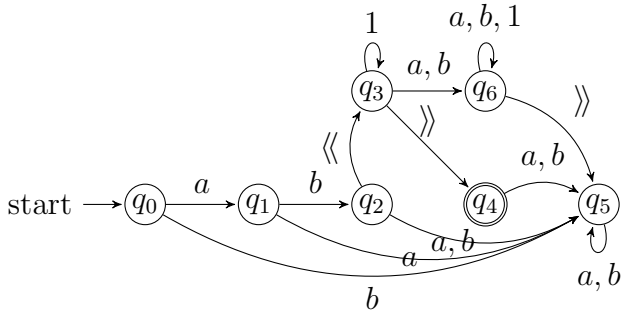
  A transition is a triple (q, α, q') *such that* $q' \in \delta(q, \alpha)$.

A nominal automaton M is *deterministic* if, for each $q \in Q$,

$$\begin{cases} |\delta(q,\alpha)| = 0, & \text{if } (\alpha = \langle\!\langle \text{ and } \|q\| = max\{\|q'\| \mid q' \in Q\}) \\ & \quad \text{or } (\alpha = \rangle\!\rangle \text{ and } \|q\| = 0) \\ |\delta(q,\alpha)| = 1, & \text{otherwise} \end{cases}.$$

Theorem 2.4.8. *[32] For each* $(\Sigma, \mathcal{N}_{fin})$*-automaton* M*, there exists a deterministic* $(\Sigma, \mathcal{N}_{fin})$*-automaton which recognises the same language as* M.

Example 2.4.9. Given $\Sigma = \{a, b\}$ and $i \in \mathcal{N}$, we have a nominal language $ne = ab\langle i.i^* \rangle$. Then we have an $(\Sigma, \mathcal{N}_{fin})$-automaton accepting the language as follows. Transitions with $\langle\!\langle$ and $\rangle\!\rangle$ represent allocating a name and deallocating a name respectively. Same in [32], transitions with 1 (the number of local register) represent getting with a name.



Let $M = \langle Q, q_0, F, \delta \rangle$ be a nominal automata over Σ and $\mathcal{N}_{fin}$, we denote the image of a map σ by $Im(\sigma)$ and the empty map by $\emptyset$. Let q be a state, w be a word whose free names are in $\mathcal{N}_{fin} \cup Im(\sigma)$ and $\sigma : \underline{\|q\|} \to \mathcal{N}$ be a map, a configuration of M denotes by $\langle q, w, \sigma \rangle$. A configuration $\langle q, w, \sigma \rangle$ is *initial* if $q = q_0$, w is a word whose free names are in $\mathcal{N}_{fin}$, and $\sigma = \emptyset$, such as $\langle q_0, \epsilon, \emptyset \rangle$; a configuration $\langle q, w, \sigma \rangle$ is accepting if $q \in F$, $w = \epsilon$, and $\sigma = \emptyset$, such as $\langle q_5, \epsilon, \emptyset \rangle$.

Definition 2.4.10. *[32] Given $q, q' \in Q$ and two configurations $t = \langle q, w, \sigma \rangle$ and $t' = \langle q', w', \sigma' \rangle$, M* moves from t to t' *if there is $\alpha \in \Sigma \cup \mathcal{N}_{fin} \cup \{\epsilon, \langle\langle, \rangle\rangle\}$ such that $q' \in \delta(q, \alpha)$ and*

$$\begin{cases} \alpha \in \underline{\|q\|}, & w = \sigma(\alpha)w',\ \sigma' = \sigma \quad \text{and } \forall i > \alpha, \sigma(\alpha) \neq \sigma(i) \\ \alpha \in \mathcal{N}_{fin} \backslash Im(\sigma), & w = aw',\ \sigma' = \sigma \\ \alpha \in \Sigma, & w = aw',\ \sigma' = \sigma \\ \alpha = \epsilon & w = w',\ \sigma' = \sigma \\ \alpha = \langle\langle, & w = \langle\langle w',\ \sigma' = \sigma[\|q'\| \mapsto n] \\ \alpha = \rangle\rangle, & w = \rangle\rangle w',\ \sigma' = \sigma_{|\underline{\|q'\|}} \end{cases}$$

where $\sigma[\|q'\| \mapsto n]$ extends σ by allocating the maximum index in $\underline{\|q\|}$ to n and $\sigma_{|\underline{\|q'\|}'}$ is restriction on $\underline{\|q'\|}$ of σ. (For more details see [31, 33])

Same in the classical automata theory, there is a Kleene Theorem for nominal regular expressions and nominal automata as Theorem 2.4.11. The proofs and constructions are provided in [32].

Theorem 2.4.11. *[32] Every language recognised by a nominal automaton is representable by a nominal regular expression. Conversely, every language represented by a nominal regular expression is acceptable by a nominal automaton.*

Example 2.4.12. Reviewing the example of a bank account in chapter 1.1, we explain the features of nominal regular languages with binders in detail.
We have a sequence of actions $o = openAccount(\text{“Yi”}); deposit(o, 110); withdraw(o, 50); closeAccount(o)$. Let $\Sigma = \{openAccount, closeAccout, deposit, withdraw\}$ be an infinite alphabet and $\{o, n_1, n_2\}$ be a set of names. We give two scenarios on those actions which producing different nominal expressions.

Debit accounts In this case, the balance of a bank account delimits the upper bound of the withdraw action. Thus, we prefer to nest the binders as follows.

$$\langle o.o\ openAccount\langle n_1.n_1\ deposit\langle n_2.n_2\ withdraw\rangle\rangle o\ closeAccount\rangle$$

Credit accounts In this case, the withdraw action is free in this sequence of the actions. Thus, we prefer to nest the binders as follows.

$$\langle o.o\ openAccount\langle n_1.n_1\ deposit\rangle\langle n_2.n_2\ withdraw\rangle o\ closeAccount\rangle$$

Certainly, in a real whole bank system, the actions are more than this sequence, and the binders for the abstract system need to be considered under more restrictions.

Chapter 3

Learning Nominal Automata

3.1 Preliminary

Before introducing our learning algorithm, some auxiliary notions are necessary. Since we reviewed nominal languages in Section 2.4, now we state new forms of representing languages and words for our learning algorithm.

There are usually many different ways to represent the same nominal words as Section 2.4 shows. For instance, $\langle\langle i.i \rangle\rangle$ and $\langle\langle j.j \rangle\rangle$ represent the same nominal word. According to the definition of $(\Sigma, \mathcal{N}_{fin})$-automaton [33], the infinite names of languages are represented as finite automata where the infinite transitions with names are represented by finite transitions with registers. Since our algorithm considers finite possibilities, we adopt a finitary representation of languages, expressions, and automata. Similarly to [11], we replace names in a nominal word with positive integers and represent nominal words in *canonical form* according to the following definition.

Definition 3.1.1 (Canonical Expressions)**.** *Let x be a number and ne a closed nominal regular expression. The* x-canonical representation $\chi(ne, x)$ *of ne is defined as follows*

- $ne \in \{\epsilon, \emptyset\} \cup \Sigma \implies \chi(ne, x) = ne$
- $\chi(ne + ne', x) = \chi(ne, x) + \chi(ne', x)$

- $\chi(ne \cdot ne', x) = \chi(ne, x) \cdot \chi(ne', x)$,
- $\chi(ne^*, x) = (\chi(ne, x))^*$
- $ne = \langle i.ne' \rangle \implies \chi(ne, x) = \langle x.\chi(ne'[x/i], x+1) \rangle$

The canonical representation *of ne is the term* $\chi(ne, 1)$.

Note that the map $\chi(\_, \_)$ does not change the structure of the nominal regular expression ne. Basically, $\chi(\_, \_)$ maps nominal regular expressions to terms where names are concretely represented as positive numbers.

Example 3.1.2. Given $\Sigma = \{a, b\}$, we give some examples of canonical representations of nominal expressions.

- aba is the canonical representations of itself; indeed $\chi(aba, 1) = aba$
- $\chi(\langle i.ai \rangle, 1) = \chi(\langle j.aj \rangle, 1) = \langle 1.a1 \rangle$ is the canonical representation of both $\langle i.ai \rangle$ and $\langle j.aj \rangle$
- the canonical representation of $\langle i.ai\langle j.ibj \rangle \rangle \langle j.j \rangle$ is

$$\chi(\langle i.ai\langle j.ibj \rangle \rangle \langle j.j \rangle, 1) = \langle 1.a1\chi((\langle j.ibj \rangle)[1/i], 2)\langle 1.1 \rangle = \langle 1.a1\langle 2.1b2 \rangle \rangle \langle 1.1 \rangle.$$

Note that the map $\chi(\_, \_)$ replaces names with numbers so that alpha-equivalent expressions are mapped to the same term (second example above).

According to the Kleene Theorem for nominal regular expressions, we state that canonical expressions have a relationship to nominal automata.

Lemma 3.1.3. *The nominal automaton associated to a nominal regular expression ne is equivalent to the nominal automaton associated to* $\chi(ne, x)$ *for any number* x.

Proof. Following Definition 3.1.1, let x be a number and ne is a closed nominal regular expression. Note that, given a regular expression ne, we can think of $\chi(ne, x)$ as a nominal regular expression and define the language of $\chi(ne, x)$ by induction on $\chi(ne, x)$ very much like in Definition 2.4.4 once each number is replaced with a fresh name. In this way it is easy to see that the language of ne and its canonical

representation coincide. Then, we let M be the nominal automaton accepting ne and M' be the nominal automaton accepting $\chi(ne, x)$. According to the Kleene Theorem, we know that $\mathfrak{L}(M) = \mathcal{L}(ne)$ and $\mathfrak{L}(M') = \mathcal{L}(\chi(ne, x))$. Then, we have $\mathfrak{L}(M) = \mathcal{L}(ne) = \mathcal{L}(\chi(ne, x)) = \mathfrak{L}(M')$. That is, M is equivalent to M'. □

Thus, we use canonical nominal regular expressions in order to represent nominal languages concretely. Let n be a natural number, the n-alphabet is defined as

$$\mathbb{A}_n = \begin{cases} \Sigma & n = 0 \\ \{\langle\!\langle, \rangle\!\rangle\} \cup \Sigma \cup \underline{n} & n > 0 \end{cases}$$

A string is called a *legal string* if it is a prefix of a nominal word.

Correspond to the depth of nominal expressions, we define the depth of legal strings $\|w\|$ as follows.

$$\|w\| = \begin{cases} 0 & \text{if there is no names in } w \\ n' & n' \text{ is the biggest number of nested binders in } w \end{cases}$$

We use $\|w\|$ as a parameter to distinguish the rows in observation tables. In this way, tables are more explicitly mapping to nominal automata. Details are discuss in following section.

3.2 Nominal Learning: Concepts

In this section, we introduce our learning algorithm based on nominal automata. Our Teacher still answers two kinds of queries: membership queries and equivalence queries regarding L a target nominal regular language over an n-alphabet $\mathbb{A}_n$.

In our algorithm, the Learner asks for membership queries about legal strings. If a string w is not a legal string, the Learner marks it as "$\perp$" in the observation table. The membership query consists of a legal string w and it has the three following possible answers:

- if $w \in L$, the answer is "1",

- if w is a prefix of a word in L, the answer is "P",
- otherwise, the answer is "0".

Remark. The answer "P" is used for efficiency. In fact, the teacher could answer "0" instead of "P". However, this would require the learner to ask more membership or equivalence queries. These would be explained in Chapter 5.

As in Angluin's $\mathsf{L}^\star$ algorithm, only the Teacher knows L. Unlike in $\mathsf{L}^\star$, the learner in our algorithm does not know the whole alphabet $\mathbb{A}_n$. The Learner knows Σ initially and learns names via counterexamples. We will see that the Learner knows the whole alphabet when the algorithm terminates.

3.2.1 Nominal observation tables

Observation tables are pivotal data structure to ensure the algorithm's functionalities. A closed and consistent observation table allows us to construct a minimal automaton. In order to deal with n-alphabet, we extend Angluin's observation tables to *nominal observation tables*, n-observation tables for short.

Definition 3.2.1 (n-observation table)**.** *A tuple* $(S, E, T, \mathbb{A}_n)$ *is an* n-observation table *if*

- $S \subseteq {\mathbb{A}_n}^*$ *is a prefix-closed set of legal strings, for all* $s \in S, \|s\| \leq n$,
- $E \subseteq {\mathbb{A}_n}^*$ *is suffix-closed,*
- $T : (S \cup S \cdot \mathbb{A}_n) \cdot E \to \{0, 1, P, \bot\}$.

As in Angluin's definition, an n-observation table $(S, E, T, \mathbb{A}_n)$ consists of rows labelled by legal strings in $S \cup S \cdot \mathbb{A}_n$ and columns labelled by strings in E:

$$row : (S \cup S \cdot \mathbb{A}_n) \to (E \to \{0, 1, P, \bot\})$$
$$row(s)(e) = T(s \cdot e)$$

In order to reflect the layers of nominal automata, we use $\|\_\|$ to distinguish rows. Therefore, we need the following auxiliary notion of equivalence of rows: in an

n-observation table $(S, E, T, \mathbb{A}_n)$, for all $s, s' \in S \cup S \cdot \mathbb{A}_n$,

$$row(s) \doteq row(s') \iff row(s) = row(s') \text{ and } \|s\| = \|s'\|.$$

Accordingly, the definition of closed and consistent table changes as follows.

Definition 3.2.2 (Closed and Consistent Tables)**.** *An* n-observation table $(S, E, T, \mathbb{A}_n)$ *is* closed *when*

$$\forall s' \in S \cdot \mathbb{A}_n . \exists s \in S. \;\; row(s') \doteq row(s).$$

An n-observation table $(S, E, T, \mathbb{A}_n)$ *is* consistent *when*

$$\forall \alpha \in \mathbb{A}_n . \forall s, s' \in S \;\; row(s) \doteq row(s') \implies row(s\alpha) \doteq row(s'\alpha).$$

3.2.2 From n-observation tables to Nominal Automata

Analogously to Angluin's theory, closed and consistent n-observation tables correspond to deterministic finite nominal automata.

Definition 3.2.3. *The* $(\Sigma, \mathcal{N}_{fin})$*-automaton* $M = (Q, q_0, F, \delta)$ *associated with a closed and consistent* n-observation table $(S, E, T, \mathbb{A}_n)$ *is defined as*

- $\Sigma = \mathbb{A}_n \setminus \{\{\langle\!\langle, \rangle\!\rangle\} \cup \underline{n}\}$, $\mathcal{N}_{fin} = \underline{n}$,
- *a set of states* $Q = \{(row(s), \|s\|) \mid s \in S\}$ *with a map* $\|\_\|_M$, *and* $\|q\|_M = \|s\|$ *for each* $q = (row(s), \|s\|) \in Q$,
- *an initial state* $q_0 = (row(\epsilon), \|\epsilon\|)$,
- *a set of final states* $F = \{(row(s), \|s\|) \mid row(s)(\epsilon) = 1, \|s\| = 0 \text{ and } s \in S\}$,
- *A transition function is a partial function* $\delta : Q \times \mathbb{A}_n \to Q$*: for all* $s \in S, \alpha \in \mathbb{A}_n$, $\delta((row(s), \|s\|), \alpha) = (row(s\alpha), \|s\alpha\|)$ *if* $s\alpha \in S \cup S \cdot \mathbb{A}_n$.

Accordingly, we define a partial function $\delta^* : Q \times {\mathbb{A}_n}^* \to Q$ inductively as follows

$$\delta^*(q, \epsilon) = q$$
$$\delta^*(q, aw) = \delta^*(\delta(q, a), w)$$

for all $a \in \mathbb{A}_n,\ w \in {\mathbb{A}_n}^*,\ q \in Q$. Note that $\delta^*(q,a) = \delta^*(q, a \cdot \epsilon) = \delta^*(\delta(q,a),\epsilon) = \delta(q,a)$. In our algorithm, the corresponding automata are also minimal as shown by Theorem 3.2.6.

Lemma 3.2.4. *Let $M = (Q, q_0, F, \delta)$ be the automaton associated with a closed and consistent n-observation table $(S, E, T, \mathbb{A}_n)$. Suppose $w, u \in {\mathbb{A}_n}^*$. We have $\delta^*(q, w \cdot u) = \delta^*(\delta^*(q,w), u)$ for all $q \in Q$.*

Proof. We prove by induction on length of w that $\delta^*(q, w \cdot u) = \delta^*(\delta^*(q,w), u)$, for all $q \in Q$.
If $w = \epsilon$:

$$\begin{aligned}
\delta^*(q, w \cdot u) &= \delta^*(q, \epsilon \cdot u) \\
&= \delta^*(q, u) && \text{by the definition of } \epsilon \\
&= \delta^*(\delta^*(q,\epsilon), u) && \text{by the definition of } \delta^*
\end{aligned}$$

If $w = aw'$:

$$\begin{aligned}
\delta^*(q, w \cdot u) &= \delta^*(q, aw' \cdot u) && w = aw' \\
&= \delta^*(\delta(q,a), w' \cdot u) && \text{associative property and definition of } \delta^* \\
&= \delta^*(\delta^*(\delta(q,a), w'), u) && \text{by induction hypothesis} \\
&= \delta^*(\delta^*(q, aw'), u) && \text{by the definition of } \delta^* \\
&= \delta^*(\delta^*(q,w), u)
\end{aligned}$$

This shows that $\delta^*(q, w \cdot u) = \delta^*(\delta^*(q,w), u)$ for all $q \in Q$ and all $w, u \in {\mathbb{A}_n}^*$. □

Theorem 3.2.5. *Assume that $M = (Q, q_0, F, \delta)$ is the automaton associated with a closed and consistent n-observation table $(S, E, T, \mathbb{A}_n)$.*

- *For all w in $S \cup S \cdot \mathbb{A}_n$, $\delta^*(q_0, w) = (row(w), \|w\|)$.*
- *For all w in $S \cup S \cdot \mathbb{A}_n$ and u in E, $\delta^*(q_0, w \cdot u)$ in F if and only if*

$$row(w)(u) = 1.$$

Proof. Let $w = w'a$ in $S \cup S \cdot \mathbb{A}_n$ and $u = a \cdot u'$ in E.

Since S is prefix-closed, all prefixes of w are in S, that is, w' is in S. We know:

$$\begin{aligned}
\delta^*(q_0, w) &= \delta^*(q_0, w'a) \\
&= \delta^*(\delta^*(q_0, w'), a) && \text{by Lemma 3.2.4} \\
&= \delta^*((row(w'), \|w'\|), a) && \text{by induction hypothesis} \\
&= \delta((row(w'), \|w'\|), a) && \text{by the definition of } \delta^* \\
&= (row(w'a), \|w'a\|) && \text{by the definition of } \delta \\
&= (row(w), \|w\|)
\end{aligned}$$

Since E is suffix-closed, all suffixes of u are in E. Depends on the length of u, we have two situations.

- When $u = \epsilon$, $row(w)(u) = row(w)(\epsilon)$ and $\delta^*(q_0, w \cdot u) = \delta^*(q_0, w)$. From preceding proof, $\delta^*(q_0, w) = (row(w), \|w\|)$. Because the table is closed, there is a $w' \in S$ such that $row(w') \doteq row(w)$. $\delta^*(q_0, w \cdot u)$ is in F if and only if $row(w')$ is in F from the definition of F. Thus $row(w')(\epsilon) = row(w)(\epsilon) = 1$, that is, $row(w)(u) = 1$.

- Assume that when the length of $u' \in E$ is n, we have that for all w in $S \cup S \cdot \mathbb{A}_n$ and u in E, $\delta^*(q_0, w \cdot u)$ in F if and only if $row(w)(u) = 1$. Let $u = au'$ and $u \in E$. Because the table is closed, there is a $w' \in S$ such that $row(w') \doteq row(w)$.

$$\begin{aligned}
\delta^*(q_0, w \cdot u) &= \delta^*(\delta^*(q_0, w), u) && \text{by Lemma 3.2.4} \\
&= \delta^*((row(w), \|w\|), u) && \text{by preceding proof} \\
&= \delta^*((row(w'), \|w'\|), u) && \text{since } row(w') = row(w) \\
&= \delta^*((row(w'), \|w'\|), au') && u = au' \\
&= \delta^*(row(w' \cdot a), u') && \text{by closedness and definition of } \delta \\
&= \delta^*(\delta^*(q_0, w' \cdot a), u') && \text{by preceding proof} \\
&= \delta^*(q_0, w' \cdot a \cdot u')
\end{aligned}$$

By induction hypothesis on u', $\delta^*(q_0, w' \cdot a \cdot u')$ is in F if only if $row(w' \cdot a)(u') = 1$. Because $row(w) \doteq row(w')$ and $u = au'$, $row(w' \cdot a)(u') = T(w' \cdot a \cdot u') = row(w')(a \cdot u') = row(w')(u) = row(w)(u)$. Therefore $\delta^*(q_0, w \cdot u)$ in F if and only if $row(w)(u) = 1$. □

Theorem 3.2.6. *The automaton $M = (Q, q_0, F, \delta)$ associated with a closed and consistent n-observation table $(S, E, T, \mathbb{A}_n)$ is minimal.*

Proof. Assume that $M = (Q, q_0, F, \delta)$ is an automaton associated with a closed and consistent n-observation table $(S, E, T, \mathbb{A}_n)$.

First, we show M observable, namely that $\forall q, q' \in Q$ we have $q = q'$ if $[\![q]\!] = [\![q']\!]$ (review the Definition 2.1.6). For $q, q' \in Q$ there are '$s, s' \in S$ such that $q = (row(s), \|s\|), q' = (row(s'), \|s'\|)$). Assume $[\![q]\!] = [\![q']\!]$. By Theorem 3.2.5, we have that $\forall e \in E.(row(s)(e) = 1 \Leftrightarrow row(s')(e) = 1)$. Thus $\forall e \in E.row(s)(e) = row(s')(e)$. Hence $row(s)$ and $row(s')$ are the same function $[E \to \{0, 1, P, \bot\}]$, which is to say that $q = q'$.

Second, we show M is reachable, namely that all $q \in Q$ there is $w \in {\mathbb{A}_n}^*$ such that $\text{run}_M(w, q_0) = q_0 \dots q$. By the definition of Q, there is $s \in S$ such that $row(s) = q$. By Theorem 3.2.5, we know $\delta^*(q_0, s) = row(s)$ for all $s \in S$ since S is prefix-closed. Thus M is reachable by Lemma 3.2.4. □

We are now ready to introduce a learning algorithm for our nominal automata.

3.3 The nL* Algorithm

We dubbed our algorithm nL*, after *nominal* L*. The algorithm is shown in Figure 3.1. The Learner in nL* is similar to the one in L*. Basically, our Learner modifies the initial n-observation table until it becomes closed and consistent in the nominal sense (according to notions introduced before). When the current n-observation table $(S, E, T, \mathbb{A}_n)$ is closed and consistent, the Learner would ask Teacher if the automaton associated with $(S, E, T, \mathbb{A}_n)$ accepts the input language L. If this is the case, the Teacher will reply 'yes' and the learning process halts. Otherwise, the learning process continues after the Teacher has produced a counterexample.

Because of the new definitions of closedness and consistency, we refine some actions about checking closedness (line 8) and consistency (line 14). If the current n-observation table $(S, E, T, \mathbb{A}_n)$ is not closed, the Learner finds a row s' such that for no $s \in S$ we have $row(s') \doteq row(s)$. If $(S, E, T, \mathbb{A}_n)$ is not consistent, the Learner finds a word $a \cdot e$, with $a \in \mathbb{A}_n$ and $e \in E$, such that for some $s_1 \in S$ and $s_2 \in S$ with $row(s_1) \doteq row(s_2)$, we have $row(s_1 \cdot a)(e) \neq row(s_2 \cdot a)(e)$.

1: Initialisation: $S = \{\epsilon\}, E = \{\epsilon\}$, $n = 0$, $\mathbb{A}_n = \Sigma$.
2: Asking for membership queries about ϵ and each $a \in \mathbb{A}_n$.
3: Construct the initial observation table $(S, E, T, \mathbb{A}_n)$.
4: **repeat**
5: **while** $(S, E, T, \mathbb{A}_n)$ is not closed or consistent **do**
6: **if** $(S, E, T, \mathbb{A}_n)$ is not closed **then**
7: **find** $s' \in S \cdot \mathbb{A}_n$ such that
8: $row(s) \doteq row(s')$ is not satisfied for all $s \in S$
9: **add** s' into S
10: **extend** T to $(S \cup S \cdot A_n) \cdot E$ using membership queries.
11: **end if**
12: **if** $(S, E, T, \mathbb{A}_n)$ is not consistent **then**
13: **find** $s_1, s_2 \in S$,$e \in E$ and $a \in \mathbb{A}_n$ such that
14: $row(s_1) \doteq row(s_2)$ but $row(s_1 \cdot a)(e) \neq row(s_2 \cdot a)(e)$.
15: **Add** $a \cdot e$ into E
16: **extend** T to $(S \cup S \cdot \mathbb{A}_n) \cdot E$ using membership queries.
17: **end if**
18: **end while**
19: Construct an automata M associated to $(S, E, T, \mathbb{A}_n)$.
20: Ask an equivalence query about M.
21: **if** Teacher replies a counterexample c **then**
22: add c and all its prefixes into S.
23: **extend** $\mathbb{A}_n$ with $\|s\|$ for all $s \in S$, $\|s\| > 0$.
24: **extend** T to $(S \cup S \cdot \mathbb{A}_n) \cdot E$ using membership queries.
25: **end if**
26: **until** Teacher replies yes to M.
27: Halt and output M.

Figure 3.1: The Learner of Learning Algorithm for Nominal Regular Languages with binders.

The main difference with respect to the algorithm of Angluin is that the Learner has partial knowledge of the alphabet. Learner's alphabet $\mathbb{A}_n$ is enlarged by adding names (line 23) during the learning process. More precisely, the Learner expands the alphabet if the counterexample requires to allocate fresh names. When the algorithm terminates, the Learner's alphabet $\mathbb{A}_n$ is the alphabet of the given language. Like in the original algorithm, our algorithm terminates when the Teacher replies 'yes'

to an equivalence query.

The Teacher answers two kinds of queries as Figure 3.2. The line 7 and line 8 are the optional part for labelling the P symbol in the table. The line 17 is the action for finding counterexamples and we have implemented it into two ways (see details in the next chapter).

```
1: Input:L
2: Initialisation: 𝔏(M_T) = L
3: answerMembershipQuery(string):
4: if string is accepted by M_T then
5:     return 1
6: else
7:     if string is accepted by a state that is not a sink state then
8:         return P
9:     else
10:        return 0
11:    end if
12: end if
13: answerEquivalenceQuery(M):
14: if M = M then
15:    return yes
16: else
17:    finding a counterexample c ∈ (M \ M_T) ∪ (M_T \ M)
18:    return c
19: end if
```

Figure 3.2: The Teacher of Learning Algorithm for Nominal Regular Languages with binders.

3.4 Correctness and Complexity

We now show that $\mathsf{nL}^\star$ is correct. That is, that eventually the Teacher replies "yes" to an equivalence query. In other word, $\mathsf{nL}^\star$ terminates with a "yes" answer to an equivalence query. Hence, the automaton submitted in the query accepts the input language.

Theorem 3.4.1. *The algorithm terminates, hence it is correct.*

Proof. We show that the if- and the while-statements terminate. Let us consider the if-statements first. It is easy to check that closedness and consistency are decidable

because these properties require just the inspection of the n-observation table (which is finite). Hence, the if-statements starting at lines 6 and 12 never diverge because their guards do not diverge and their then-branch is a finite sequence of assignments:

- The if-statement for closedness (line 6) terminates directly if the table is closed. Otherwise, in case of making a table closed, we find a row $s' \in S \cdot \mathbb{A}_n$ such that $row(s') \doteq row(s)$ is not satisfied for all $s \in S$. The algorithm adds s' into S. Since A'_n is finite and bounded by n, the sets S and E are both finite. Thus, $S \cdot \mathbb{A}_n$ is a finite set and there are finitely many choices for s'. That is, line 9 can only be executed finitely times. Besides, the content of rows is one of the permutations and combinations of $\{0, 1, P, \bot\}$ which also has finite possibilities. So we conclude that the branch terminates.

- Similarly, the if-statement for consistency (line 12) terminates directly if the table is consistent. Otherwise, to make the table consistent the algorithm searches for two rows $s_1, s_2 \in S$ satisfying the condition at lines 13 and 14. As in the previous case for closedness, to add elements into E there are only finitely many possibilities s_1, s_2, a, and e (line 15). Thus, the branch of the if-statement terminates.

Therefore, the while-statement (line 5) terminates in finite repetitions, since the algorithm makes a table closed and consistent in finite operations. Then, the algorithm will succeed in construct an automata M associated to a closed and consistent table. Next, the Learner asks for an equivalence query. The Teacher replies a counterexample c (line 21) or yes (line 26). It remains to prove that the Learner only asks finitely many equivalence queries.

Let M be the nominal automaton associated to the current n-observation table $(S, E, T, \mathbb{A}_n)$. Assume that the equivalence query about M fails. The Teacher has to find a counterexample c; this is finitely computable since the nominal regular expressions are closed under the operations of Kleene algebra and under resource complementation. Hence, the if-statement on line 21 goes the branch extending the table (line 22). Then, the algorithm will start a new loop (line 5) for the modified table by the counterexamples. As Theorem 3.2.6 proved, a closed and consistent table builds a minimal automaton. And a minimal automaton for a regular nominal language

has finite states. A new automaton M' will be constructed when the extended table $(S, E, T, \mathbb{A}_n)$ is closed and consistent. Since M' handles the counterexample, M' has more equivalent states to the minimal automaton accepted the given language, compared with M. Repeating this process, the automaton associated with a closed and consistent table has the same number of the minimal automaton which accepted the given language. Since the minimal automaton is unique, the two minimal automata are equal. The Teacher relies yes to the automaton at such a point. Therefore, with respect to the number of the states of the minimal automaton accepted the given language, equivalence queries are finite and the algorithm terminates finally. □

We remark that the proof of correctness is close to the original of the Angluin's because our modifications are all on the data structure not the logical structure. Let M be the minimal automaton accepting the given language and let M have $\mathfrak{s}$ states.

Lemma 3.4.2. *The algorithm produces equivalence queries $\mathfrak{s}$-1 times at most.*

Proof. If the n-observation table $(S, E, T, \mathbb{A}_n)$ is found to be incorrect by the counterexample c, then since the automaton M' associated with the table $(S, E, T, \mathbb{A}_n)$ must have at least one more state. In the worst case, the learning process produces the number of the states by monotonically increasing. It need to take $\mathfrak{s}$-1 times since the initialisation has one state. That is, the algorithm produces equivalence queries $\mathfrak{s}$-1 times at most. □

In the following, we analyse the number of membership queries in the worst case. Let $\mathfrak{b}$ be a bound on the maximum length of the counterexamples presented by the Teacher. The membership queries is based on the entries of rows and columns in the table.

From the Figure 3.1 (line 1), we know that S and E contain one element ϵ initially. As the algorithm runs, it will add one element to S when $(S, E, T, \mathbb{A}_n)$ is not closed (line 10). And it will add one element to E when $(S, E, T, \mathbb{A}_n)$ is not consistent (line16). For each counterexample of length at most $\mathfrak{b}$ presented by the Teacher, the algorithm will add at most $\mathfrak{b}$ elements to S (line 22).

Thus, the cardinality of S is depends on $\mathfrak{s}$ and $\mathfrak{b}$. In detail, S is at most

$$1 + (\mathfrak{s} - 1) + \mathfrak{b}(\mathfrak{s} - 1) = \mathfrak{s} + \mathfrak{b}(\mathfrak{s} - 1)$$

because $(S, E, T, \mathbb{A}_n)$ can be not closed at most $\mathfrak{s} - 1$ times. As the same as the Teacher replies counterexamples at most $\mathfrak{s} - 1$ times. And each time the Teacher replies with a counterexample of length $\mathfrak{b}$, S will be increased by at most $\mathfrak{b}$ elements. The cardinality of E is at most $\mathfrak{s}$, because $(S, E, T, \mathbb{A}_n)$ can be not consistent at most $\mathfrak{s} - 1$ times.

The cardinality of $S \cdot \mathbb{A}_n$ could calculate from two parts: the cardinality of S and the cardinality of $\mathbb{A}_n$. We already know the cardinality of S is at most $\mathfrak{s} + \mathfrak{b}(\mathfrak{s} - 1)$. As the definition of A_n, let $\mathfrak{k}$ be the cardinality of Σ. Therefore, the cardinality of A_n is $\mathfrak{k} + n + 2$, 2 for the binders operations. And, the cardinality of $S \cdot A_n$ is $(\mathfrak{k} + n + 2)(\mathfrak{s} + \mathfrak{b}(\mathfrak{s} - 1))$ at most.

Therefore, the maximum cardinality of $(S \cup S \cdot \mathbb{A}_n) \cdot E$ is at most

$$(\mathfrak{k} + n + 2)(\mathfrak{s} + \mathfrak{b}(\mathfrak{s} - 1))\mathfrak{s} = O((\mathfrak{k} + n)\mathfrak{b}\mathfrak{s}^2).$$

3.5 Running nL*: An Example

Given a finite alphabet $\Sigma = \{a, b\}$, we have an example of learning a language L representing as canonical nominal regular expression $cne = ab\langle 1^* \rangle$.

In the first step, we initialize $S_1 = \{\epsilon\}$, $E_1 = \{\epsilon\}$, $n = 0$ and $\mathbb{A}_0 = \Sigma$, and construct T_1 as follows.

Step 1

T_1=

$\|\_\|$		ϵ
0	ϵ	P
0	a	P
0	b	0

$(S_1, E_1, T_1, \mathbb{A}_0)$ consistent? There is only one row in S, thus the table is consistent.
$(S_1, E_1, T_1, \mathbb{A}_0)$ closed? No, $row(b) \neq row(\epsilon)$.
So, $S_2 \leftarrow S_1 \cup \{b\}$ and we go to step 2.

Step 2

Let $S_2 = S \cup \{b\}$ and $E_2 = E$ and then construct a new observation table $(S_2, E_2, T_2, \mathbb{A}_0)$ through membership queries.

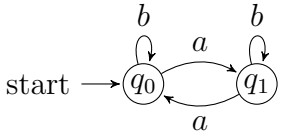
T_2=

$\|\_\|$		ϵ
0	ϵ	P
0	b	0
0	a	P
0	aa	0
0	ab	P

$(S_2, E_2, T_2, \mathbb{A}_0)$ closed? $\surd$

$(S_2, E_2, T_2, \mathbb{A}_0)$ consistent? $\surd$

Then, we compute the automaton M:



Teacher replies no and a counterexample, say, $ab\langle\langle 1.\rangle\rangle$. It is in L not in M. And we go to step 3.

Step 3

Let $S_3 \leftarrow S_2 \cup \{a, ab, ab\langle\langle 1., ab\langle\langle 1.\rangle\rangle\}$, $E_3 \leftarrow E_2$, and $n = 1$, and then, the alphabet is extended to $\mathbb{A}_1 = \Sigma \cup \underline{n} \cup \{\langle\langle, \rangle\rangle\}$. We should construct new observation tables sequentially through membership queries. Then we check the new table for closeness and consistency.

T_3=

$\|\_\|$		ϵ
0	ϵ	P
0	b	0
0	a	P
0	ab	P
1	$ab\langle\langle 1.$	P
0	$ab\langle\langle 1.\rangle\rangle$	1
1	$\langle\langle 1.$	0
0	a	P
0	$ab\langle\langle 1.\rangle\rangle a$	0
0	$ab\langle\langle 1.\rangle\rangle b$	0
1	$ab\langle\langle 1.1$	P
1	$ab\langle\langle 1.a$	0
...	...	...

$(S_3, E_3, T_3, \mathbb{A}_1)$ consistent?

No, $row(ab) = row(\epsilon)$ but $row(ab\langle\langle 1.) \neq row(\langle\langle 1.)$.

$(S_3, E_3, T_3, \mathbb{A}_1)$ closed?

No, $row(\langle\langle 1.)$ with $\|\langle\langle 1.\| = 1$ has a fresh content.

Step 4

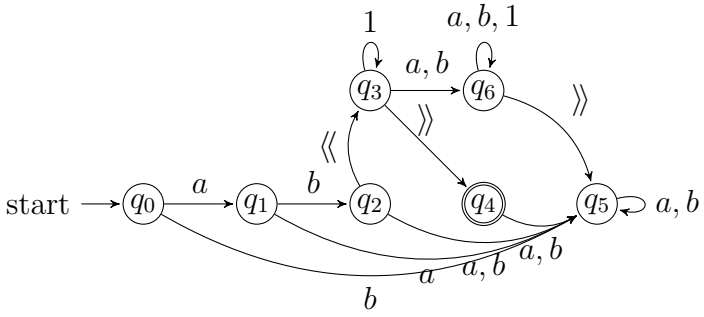
Let $S_4 \leftarrow S_3 \cup \{\langle\langle 1.\}$,$E_4 \leftarrow E_3 \cup \{\langle\langle 1.\}$, we should construct a new observation table $(S_4, E_4, T_4, \mathbb{A}_1)$ and check the new table for closeness and consistency.

T_4=

$\|\_\|$		ϵ	$\langle\langle 1.$
0	ϵ	P	0
0	b	0	0
0	a	P	0
0	ab	P	P
1	$ab\langle\langle 1.$	P	$\bot$
0	$ab\langle\langle 1.\rangle\rangle$	1	0
1	$\langle\langle 1.$	0	$\bot$
0	ba	0	0
0	bb	0	0
1	$ab\langle\langle 1.1$	P	$\bot$
1	$ab\langle\langle 1.a$	0	$\bot$
...	...	...	...

Once the table is closed and consistent, we ask an equivalence query.

Finally, the teacher replies "yes" to an equivalence query about. The learning progress terminates. The learner automaton is as below.



Chapter 4

Implementation

This chapter discusses the implementation of our learning algorithm, named **ALeLaB**[1]. Section 4.1 provides an overview of the architecture of **ALeLaB**; more technical details are in Section 4.2. More precisely we describe the format of inputs and outputs as well as some crucial data structures. Section 4.3 provides details of the implementation of the main software components of **ALeLaB**. A key aspect of **ALeLaB** is that is features two different strategies to find counterexamples; these strategies are based on a core feature of nominal automata and are explained in Section 4.4. Finally, Section 4.5 reports on the testing of **ALeLaB**.

4.1 Architectural Aspects

We describe the main architectural aspects of **ALeLaB** as a multi-layer architecture. This abstract architecture is given in Figure 4.1 and consists of the following layers:

- The User Interface layer has two roles: (i) the first role is to gather the information from the user to perform operations; (ii) the second role is to translate the outcome of **ALeLaB** in a format comprehensible to the user.
- The Data Calculations layer performs calculations with the information from the user, and transports information between the two surrounding layers. This layer achieves the main functionalities of our learning algorithm.

[1]Available on `https://github.com/easyxy1/ALeLaB`

- The Data Reader/Writer layer provides data/information to be stored and retrieved. The data can be extra files from the user's local file system.

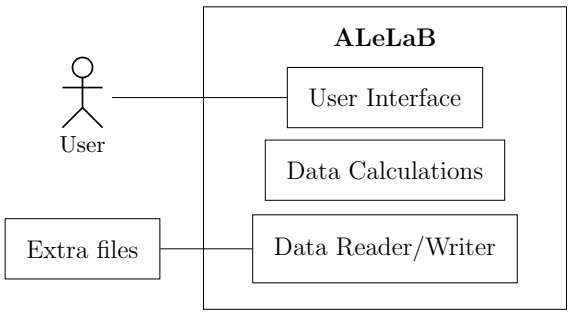


Figure 4.1: The architecture of **ALeLaB**

The user interacts with **ALeLaB** through a command-line interface (the development of a graphical user interface is left for future work). The Data Reader/Writer layer can store and retrieve data a file whose content represents a finite nominal automaton accepting the language L to be learnt. The name of this file is specified in the command line starting **ALeLaB** together with other parameters, in particular a string encoding the finite alphabet Σ. Details of data structures are described in Section 4.2.

The main components in **ALeLaB** are the ones realising the behaviour of the teacher and of the learner. They are implemented in the Data Calculations layer. Figure 4.2 yields a diagram describing the Data Calculations layer. Dashed arrows represent data transported across layers and solid arrows represent data flowing within the current layer. Dashed boxes presents the data sets in the Data Reader/Writer layer. The inputs L and Σ are the target language and its corresponding finite alphabet respectively. They are provided by users.

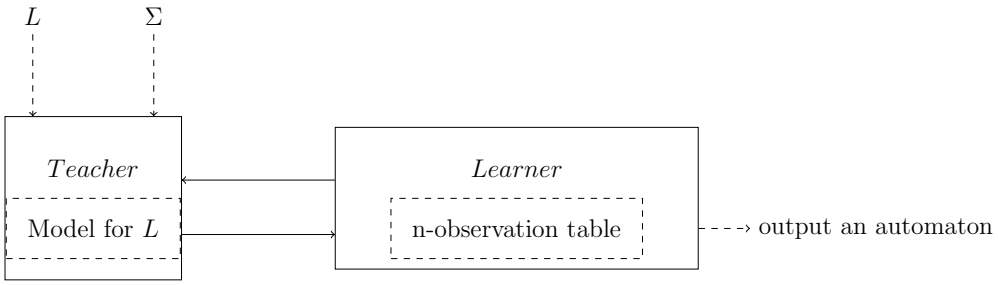


Figure 4.2: The schema of data calculations

The Teacher is given (a representation of) the language L and the finite alphabet Σ as input by the user. As we will see, for evaluation, the input for L is either a nominal regular expression (in normal form) or a nominal automaton. In the former case, the Teacher component computes an automaton accepting L. The Teacher passes the finite alphabet Σ to the Learner. On the other hand, in practice, the languages are stored in different models with respect to the certain situation. For example, applying for the software testing, the software architecture is the target and it is provided by a huge amount of testing data. The Teacher formulates the data into a proper model which could answer the two types of queries [47].

The Learner firstly receives the finite alphabet Σ from the Teacher and sets some local variables. More precisely, the Learner creates the data structure for the n-observation table and initially sets to 0 the parameter n for nested binders. Further operations are based on the n-observation table. To achieve the functionality of completing the table, the Learner calculates the data from the entries of the table, transfers the data to the Teacher and then fills the table with the responses of the Teacher. To verify when the table is closed and consistent, the Learner invokes operations on n-observation table (that are described later). When the table is closed and consistent, the Learner produces a minimal automaton associated to the table, transfers the data about the automaton to the Teacher and then gets a response.

Data interactions implement that the Teacher teaches the Learner what is right and what is wrong. The data transferred from the Learner to the Teacher implements the queries in our algorithm. There are two types of queries: membership queries and equivalence queries. In the **ALeLaB**, the queries' types are distinguished by the data structures and methods. Each membership query is performed in sequence. Each equivalence query is a data set representing an automaton. The data transferred from the Teacher to the Learner implements the answers to the queries in the algorithm. In the **ALeLaB**, all the answers are in the representation way of strings, but returned by different methods.

4.2 Technical Specifications

This section presents a detailed overview of the technical specifications. We first state the formats for special symbols, inputs and output. We then define the data structures for alphabets and languages. Finally, we explain how to construct the n-observation table and the automaton.

Symbols formats We need to consider how to implement all the symbols easy-understanding by the machine, not only letters and names, but also operators. As in Table 4.1, we have their formats for the implementation.

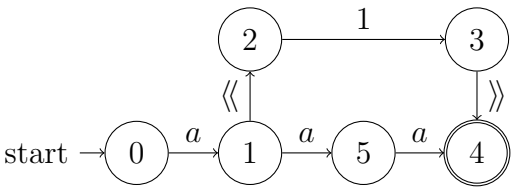
- The empty string is represented as 0 in inputs and during the process, in order to users input easily.
- The representation of the letters in the implementation is different in fonts from the ones in theory because the font of the implementation is the default font of the most machine systems.
- The names are represented in a same font in all situations. As the outputs are the automata, we decide to use a unique representation of binders.
- The brackets $<, >$ can be typed directly by keyboard, so the users can use the **ALeLaB** easily without changing input methods and **ALeLaB** need no actions about transferring formats. To avoid the confusion of the meaning of the symbol ".", we ignore the declaration of names, such as "1.", and assume that names are declared from 1 in ascending order.
- When the use inputs expressions, the operators are needed. The representations of the operators are the same.

Input formats. Our tool **ALeLaB** allows two types of inputs. One is to enter the canonical expression for the target language, as defined in Definition 3.1.1. Note that, the representation of symbols in expressions should be formatted as in Table 4.1. Another type of input is through providing a file whose content is a textual

	In theory	In inputs	In process	In outputs
empty string	ϵ	0	0	N/A
Letters	$a, b, c, \ldots$	a,b,c,...	a,b,c,...	a,b,c,...
Names	$1, 2, 3, \ldots$	1,2,3,...	1,2,3,...	1,2,3,...
Binders in words	$\langle\langle 1.\_ \rangle\rangle$	$< \_ >$	$< \_ >$	N/A
Binders in expressions	$\langle 1.\_ \rangle$	$< \_ >$	$< \_ >$	N/A
Binders in automata	$\langle\langle\ \rangle\rangle$	$< >$	$< >$	$< >$
concatenation	ingored	ingored	.	N/A
union	$+$	$+$	$+$	N/A
Kleene-star	$*$	$*$	$*$	N/A

Table 4.1: The representations of symbols

representation of an automaton accepting the target language. We illustrate this with the example in Figure 4.3.



(a) A nominal automaton

```
initial 0
final 4
0 a 1
1 < 2
2 1 3
3 > 4
1 a 5
5 a 4
```

(b) Textual representation

Figure 4.3: Inputting L as automaton

Suppose the input language is the one accepted by the automaton depicted on the left-hand side of Figure 4.3. The input to **ALeLaB** will then consist of a file containing the text on the right-hand side of Figure 4.3. Each line of the text described an element of the automaton:

- the first two lines specify initial and final state (if the automaton has more than one final state, the second line lists all of them separated with a blank space)
- each of the remaining lines specifies one of the transitions of the automaton; for instance, the third line corresponds to the transition from the initial state and the forth one to the allocation transition from state '1' to state '3' of the automaton on the left-hand side of Figure 4.3.

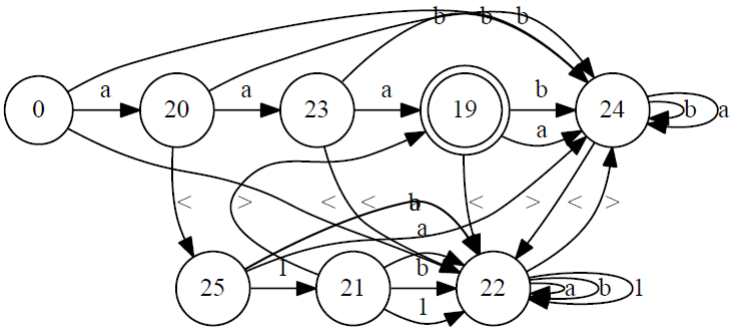


Figure 4.4: The graph of the output

The automaton in the input file is minimised to make it deterministic before starting the learning process; this makes the language equivalence check more efficient.

Output formats. When the Teacher replies 'yes' to an equivalence query, the Learner knows that the current automaton accepts the language given as input to the Teacher. This ends the learning process and the Learner outputs the current automaton. We rely on the Graphviz library [35] to render the output as DOT file. The graph shows a minimal nominal automaton as in Figure 4.4. For example, the output graph produced for the example in Figure 4.3 is reported in Figure 4.4. The nodes of the automaton are layered according to the nesting of the binders. In our example (cf. Figure 4.6) the layer containing state "0" is the first layer corresponding to the first layer while state "25" is on the second layer corresponding to the second layer. In order to obtain such layering, **ALeLaB** exploits a core feature of nominal automata, namely the nesting of binders. This feature allows to incrementally determine the alphabet to use at each layer. As said, note that the nodes at each layer form a classical finite-state automaton (once allocation and deallocation transitions are ignored) on an alphabet that depends on the nesting of binders. In our example, the states on the first layer correspond to those states that are "outside" any binder, namely the states whose alphabet is just the finite alphabet Σ. Instead, the states on the second layer (that is states "21", "22", and "25") are those reached after an allocation transition; for such states the alphabet is Σ augmented with a new name, which in the canonical representation, is the number 1. In fact, e.g., state "21" has transitions labelled by 'a', 'b', and '1' to state "22".

This is realised through the data structures that we now describe. Firstly, the alphabet for the given language is divided and stored into two sets which are instances of `Set <String>`; the first set is for the finite letters and another is for names. The binder symbols $\langle$ and $\rangle$ are not set as operators directly. Binders comes in pairs that determine their scope. Thus, a method `RegularExpression.AddConcat(String)` is defined for computing the scope of binders in order to generate machine-readable expressions. The method brackets binders' lexemes and then regenerates the expressions with the concatenation. The lexeme $\langle$. is surrounded by opening round brackets and the symbol $\rangle$ is surrounded by closing brackets. This is necessary in order to treat symbols $\langle$ and $\rangle$ as normal letters. For example, a canonical expression $a\langle 1\rangle$ is transformed as $a.\ (\langle .(\ 1.\ )\rangle)$.

The implementation of the n-observation tables is a matrix for storing the information about the relationships between the strings and the input language. Namely, the elements of the first row and the first column are the strings. Other elements of the matrix are storing the information about the relationship. The relationships are denoted by the elements of an enumerator list. The enumerator list is corresponding with the definitions of $\{0, 1, P, \perp\}$.

The Automata are implemented as data sets. The data set contains some data sets which implementing the states and transitions. Each of states and transitions is an individual instance of a data set. States contain the information about its identification. Transitions contain the information about from which state, to which state, and with which label.

4.3 Components Implementation

The architecture described previously has been implemented in Java. In this section, we describe the main features of the implementation. We refer to the UML diagram in Figure 4.5 to discuss our implementation.

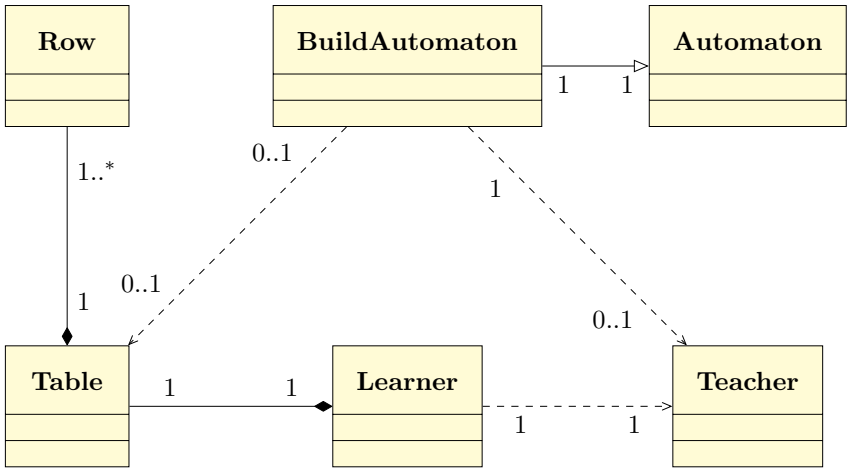


Figure 4.5: Conceptual schema of the main classes

When we presented the architecture of **ALeLaB**, we saw that are the main components the Teacher and the Learner. Hence, the core of **ALeLaB**includes a `Teacher` class and a `Learner` class. Also, we associate the `Table` class with `Learner`. We will first focus our attention on how automata are represented and then on the most important part of **ALeLaB**: the Learner realised in the `Learner` class. Then, we will consider to the classes associated to `Learner` class, namely the class `Table` and the class `Teacher`.

Learner is implemented by the main `Learner` class whose interface is described by the following UML diagram:

Learner
- alphabet: Set<String> - table: Table ...
+ Learner(teacher: Teacher): void + processMembershipQueriesForTables(learner: Table, teacher: Teacher): void - makeTableClosedAndConsistent (table: Table): void - ensureConsistency (t: Table): Table - closeTable (t: Table): Table ...

This class invokes the methods in the `Teacher` class and maintains the table used in the learning process. The constructor method of the class `Learner` re-

quires an object `teacher` of class `Teacher`. In fact, the constructor calls the method `teacher.getAlphabet` to initialise the attribute `alphabet` with a `Set <String>` object representing the finite alphabet Σ. Then, the constructor of `table` declares the attribute `table` as an object of class `Table` initialised using the value of `alphabet`. After the initialisation phase, the constructor starts calling the following methods to realise the learning process of our algorithm:

- method `processMembershipQueriesForTables` achieves the data interactions about membership queries (cf. Figure 4.2): composing the membership query from the current table `table`, transferring queries to the `teacher`, and then updating the table `table` according to the answers to the queries;
- method `makeTableClosedAndConsistent` checks the table is closed and consistent using the methods `ensureConsistency` and `closeTable`;
- when the current table is closed and consistent, the constructor of `table` retrieves the automaton `automaton` from attribute `table` as an object of the `Automaton` class. The object `automaton` is transferred to the `teacher` to check if it accepts the input language.

The equivalence query is realised by invoking method `teacher.checkAutomaton` on the object `automaton` which, if `automaton` does not accept the input language, returns a couterexample. The constructor of `Learner` modifies the table `table` according to this counterexample and continues with the learning process. If the invocation to `teacher.checkAutomaton` is successful, the process terminates and outputs a dot graph file presenting the `automaton` object.

As said, the class `Learner` has an associated object of class `Table` which manages the data of an n-observation table and has the following interface: The attribute `setA` stores the information of the finite alphabet and will be declared when the `Learner` class calls the method `setAlphabet(alphabet)`. The attribute `freshName` is to store the names dynamically discovered during the learning process. This attribute is initially set to `null` and expanded when the method `addFreshName` is called.

The attribute `rowS` and attribute `rowSA` are the data of rows of a table. The

Table
- setA: Set<String> - freshName: Set<String> - rowS: List<Row> - rowSA: List<Row> - setE: List<String> ...
+ Table(): void + setAlphabet(alphabet: Set<String>): void + addFreshName(name: String): void + extendByCE(counterexample: String) + printTable(): void ...

attribute `setE` is the titles of the columns of a table. A table would be output by calling the method `printTable()`.

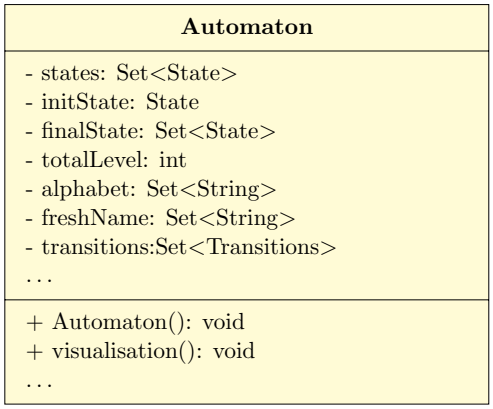
Teacher The class `Teacher` has the following interface:

Teacher
+ Strategy: enum - strategy: Strategy = Strategy.VERTICAL - selectedCounterexample: Set<String> - teacherAutomaton: Automaton ...
+ Teacher(choice: boolean, alphabet: Set<String>, strategy: Strategy, language: String): void + getAnswer(word: String): String + getAlphabet(): Set<String> + checkAutomaton(a: Automaton): String - selectCE(learner: Automaton,counterexampleLength:int,maxLength:int): String - verticalStrategy(wordlist: Set<String>): String - horizontalStrategy(wordlist: Set<String>): String ...

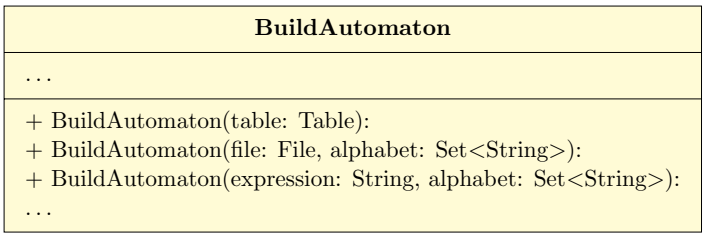
This class computes the given language and generates the machine-readable data. The core of class includes the method `getAnswer`, the method `checkAutomaton`, the method `verticalStrategy` and the method `horizontalStrategy`. The main aim of our recent experiments is to check the learner algorithm correction and the strategy algorithm performance, so the types of inputs are designed in an easy way to comprehend and judge what is the language. In the practical experiments, we prefer to the data structure which can store the raw data directly, such as sets. We will discuss it in the section 5.5. The boolean parameter `choice` used in the constructor marks the type of input used to represent the language to be learnt and it is used to decide how to handle the parameter `language`, which represents the input language. The data of the `language` will be computed and restored into the `teacherAutomaton`

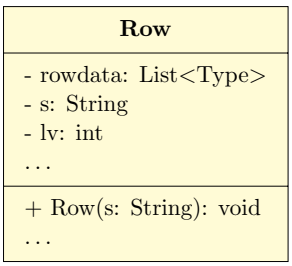
for answering the queries. Certainly, the data type `Automaton` is designed for the current stage of theoretical experiments, providing the completed language of infinite elements. The possible application scenario will be shown in section 5.5. The membership queries handler `getAnswer` receives a string `word` representing a legal word and returns the answer to the membership query about such word. The equivalence queries handler `checkAutomaton` receives the `Automaton` object `a` computed by the `Learner`'s constructor as described above and returns the answer to the equivalence query about this object. If the Learner's automaton is not correct, the method `selectCE` will be called to finding a proper counterexample under the strategy. As mentioned in the previous section, the Teacher selects a random counterexample according to a specified strategy. We will discuss the different strategies later; for the moment, it is sufficient to bear in mind that the strategy is set in the attribute `strategy` by the the constructor method of `Teacher`. When the equivalence check fails, a counterexample is selected according to the attribute `strategy`. As we will see, there are two strategies rendered by the methods `verticalStrategy` and `horizontalStrategy`. The selected counterexample is stored into the attribute `selectedCounterexample`, in case of being selected twice. Before replying a counterexample, the counterexample is checked not in `selectedCounterexample`, so that the reply ensure the learning process learning useful information. Further, `selectedCounterexample` makes the process efficient.

Automata To implement nominal automata, **ALeLaB** features the class `Automaton`, which relies on an auxiliary class `State` to represent the states (not representd in Figure 4.5). The class `Automaton` has the following interface:



The class `Automaton` has a class attribute `totalLevel` recording the total number of layers in an automaton. The class `State` (interface in Appendix A.1) features an object attribute `level` to store the information about at which level the state is in the automaton. This attribute is used to handle transitions with binders and names. If the value of `level` is 0 then, by construction, states have no deallocation transitions. If `level == totalLevel`, the state has no allocating transition. The class `BuildAutomaton` extends the class `Automaton`, used to construct the automata. The class `BuildAutomaton` provides three constructor methods to implement building automata from an n-observation table, an extra file, and an expression respectively. The interface of the class `BuildAutomaton` is as follows. Building an automaton





from an n-observation table is implemented by the constructor method of the class `BuildAutomaton` requiring an object `table` of class `Table`. Firstly, the attributes `setA` and `freshname` in `Table` class are assigned to the attributes `Alphabet` and `freshName` in `Automaton` class. The variable `maxLevel` in `Table` class identifies the `totalLevel` in `Automaton` class. Then, each row in `table` is used to identify a state of the automaton by the attribute `rowdata`. The interface of `Row` class is as above. The attribute `rowdata` stores the information about the relationships between the strings and the input language. The variable `level` in `State` class is assigned as the variable `lv` in `Row` class. The initial state is identified by the row with `s=`ϵ. The final states are identified after computing `lv` and the first content of `rowdata`. As Definition 2.4.7, a state is final when `lv=0` and the first content is `State.Type.FINAL`.

Next, the transitions of the automaton are computed by the attribute `s` in `table` with the attributes `Alphabet` and `freshName`.

Building an automaton from an extra file is implemented by the constructor method of the class `BuildAutomaton` requiring an object `file` of class `File` and a data set `alphabet`. Firstly, the parameter `alphabet` is assigned to the attribute `alphabet` in `Automaton` class. Then, the initial state and the final states are declared by computing the first two lines of `file`. As the example of Figure 4.3, the symbol "0" after "initial" identifies the initial state with `id=0`, and the final state is identified with `id=4`. Then, the other states and transitions are produced by the rest lines of `file`. The rest lines can be divided into three part by the spaces. The first part and the third part identify two states respectively, and the second part is the transition label from the first state to the second state. Meanwhile, the information is computed for declaring the attribute `freshName`. Moreover, this constructor would produce more states and transitions with `alphabet` if the file does not contain fully information about a deterministic automaton. Finally, the automaton is modified as a minimal automaton.

Building an automaton from an expression is implemented by the constructor method of the class `BuildAutomaton` requiring a String `expression` and a data set `alphabet`. The progress of this constructor is similar to the algorithm for converting the regular expressions to the minimal regular automata as explained in Section 2.2. The differences are extra operations for binders. The first operation is an additional step in generating machine-readable expressions for computing the scope of binders. The second one is to generate the `totalLevel` and the elements of the `freshName` by counting the nested binders. Besides, the transitions of binders computes the attribute `lv` of the states connected by the transitions. In the case of an allocated transition, the `lv` of the state which the transition is towards is 1 bigger than the one which the transition is from. In the case of a deallocated transition, the `lv` of the state which the transition is towards is 1 smaller than the one which the transition is from.

Other essential classes are shown in Appendix A.

4.4 Strategies for Selecting Counterexamples

Counterexamples play a decisive role to the efficiency of the learning algorithm. Compared with original algorithm of Angluin, the counterexamples of our algorithm require to extend the Learner's alphabet to account for allocation transitions. We have identified two specific strategies that we introduce in this section.

The two strategies deal differently with allocation and deallocation of names. The first strategy is to give lower priority to counterexamples that require allocating new names; basically, the Teacher has a bias for counterexamples with minimal number of names. We call this the *horizontal* strategy since this strategy teaches the learner the structure of the automaton layer-wise, namely before learning a layer of the automaton, the Learner has to acquire the information of the lower layers. In the second strategy, dubbed *vertical* strategy, the Teacher has the opposite bias and gives priority to counterexamples with maximal number of names. Dually to the horizontal strategy, in the vertical strategy the Learner acquires first the information about as many names as possible, and then it learns about the structure of each layer.

We give a pictorial representation of the two strategies with an example. Consider the nominal automaton in Figure 4.6 where the dashed boxes highlight the two layers.

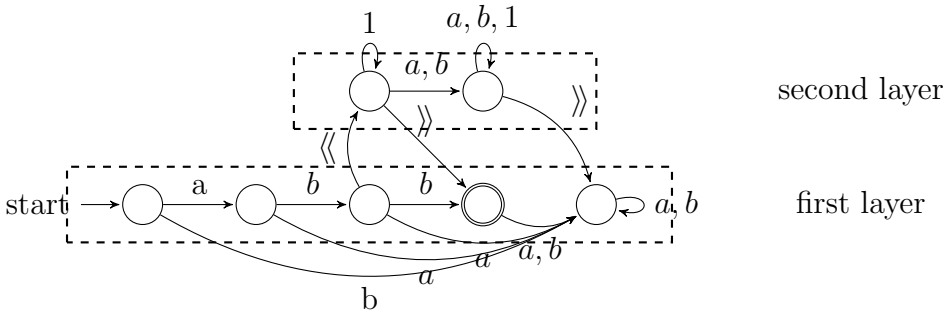


Figure 4.6: The layers of an automaton.

Formally, a layer consists of the automaton with states using the same amount of freshly allocated names and the transitions among such states. For instance, the first layer in Figure 4.6 includes all the states where no names are allocated, while the second layer has all states reachable after one allocation transition has been

used to account for a fresh name. Observe that, by definition, inter-layers transitions can only be allocation or deallocation transitions. Hence, a nominal automaton may be thought of as several stacked classical finite automata connected through allocation and deallocation transitions (where the automaton at the i-th layer has alphabet $\Sigma \cup \underline{n}$). As declared in Figure 3.1, the Learner does not ask for allocating a name unless the Teacher replies a counterexample containing a name. Image that the Teacher never reply a counterexample containing a name, the Learner only can concrete transitions within the first layer as Figure 4.6. The strategies allow the counterexamples affect the direction of the Learner's automaton completion.

As the Figure 4.7, the Teacher first finds a set of counterexamples which are in the certain length of *counterexampleLength*. The *counterexampleLength* would be different under the strategy. Recently, the *counterexampleLength* is the number of the states of Learner's automaton when the strategy is horizontal, otherwise, it is the number of the states of the Teacher's automaton. We give a bound *maxLength* to avoid the infinite loop and output alarms. Then, the proper counterexample will be filtered from the set *availableCEset* under the strategy.

```
1: selectCE(learner, counterexampleLength, maxLength):
2: ...
3: initialisation availableCEset = null
4: while availableCEset is empty and counterexampleLength < maxLength do
5:     stored the counterexamples of the length counterexampleLength
6:     counterexampleLength++
7: end while
8: if strategy is HORIZONTAL then
9:     select a one of availableCEset under horizontal strategy
10:    if strategy is VERTICAL then
11:        select a one of availableCEset under vertical strategy
12:    end if
13: end if
14: ...
```

Figure 4.7: The algorithm of finding counterexamples.

Let us first discuss the *horizontal* strategy. The idea is to select the counterexample allowing the table to construct each layer of an automaton in ascending order. That is, the selected counterexamples have nested fresh names as less as possible. The Learner uses these counterexample to expand the table first before

starting to expand the alphabet. With this strategy, the expansion of the alphabet is "slower" compared to the completion of a layer; as a result the Learner needs more times to learn the maximal amount of names needed in the language. As Figure 4.8 shows, the satisfied counterexamples are selected from the list *list* and stored into the list *filtered*. The *list* is a finite list of string in a certain length `counterexampleLength`. The attribute `counterexampleLength` is computed from the states numbers of the Learner's automaton and the Teacher's automaton. The Teacher receives the Learner's automaton and then the loop filters the words whose depth is smallest (recall that $\|w\|$ is the depth of legal strings, see page 32). In a word, the counterexample prefer to cover all the state in the first layer and spread the second layer in the Figure 4.6.

Input: a list of counterexamples *list*.
Output: a list of words *filtered*.
1: **if** *list* is not empty **then**
2: **for** each word w in *list* **do**
3: **if** $\|w\|$ is smallest than other words' in *list* **then**
4: add w into *filtered*
5: **end if**
6: **end for**
7: **end if**

Figure 4.8: The algorithm of horizontal strategy.

For instance, let us consider the automaton in Figure 4.6. Suppose that the list of counterexamples contain $w_1 = abbbbb$ and $w_2 = abb<1>$. Then the horizontal strategy will filter w_1 but not w_2. If instead the list was made of w_2 and $w_3 = abb<11>$ then both w_2 and w_3 would be filtered and one of them randomly chosen are the returned counterexample.

We now consider the *vertical* strategy. In some sense this strategy is dual to the horizontal strategy. The idea is to select the counterexample allowing the table to construct each layer of an automaton in descending order. The Learner completes the highest layer first. This is pictorially represented by the red arrow in Figure 4.9, which suggests a possible "trajectory" for exploring the states of the two layers of

our running example.

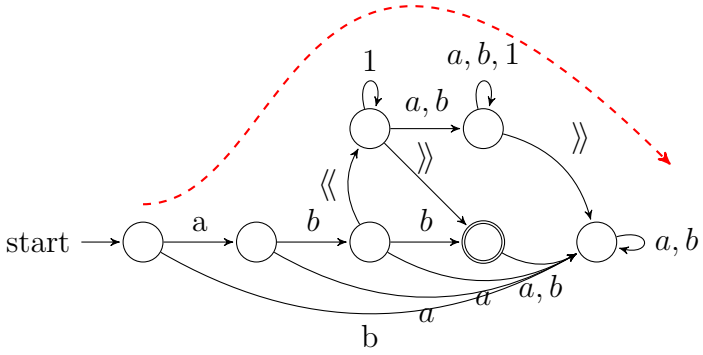


Figure 4.9: The example of the completion under the vertical strategy.

Basically, the counterexample leads the Learner to acquire information on the highest layer and than about the lower ones. Thus, the counterexamples under the vertical strategy have as many as possible nested fresh names. The best-case scenario is that the Learner learns all names with the first counterexample. Unlike in the algorithm of the horizontal strategy, the words filtered out have the biggest depth, as in Figure 4.10.

Input: a list of counterexamples *list*.

Output: a list of words *filtered*.

```
1: if list is not empty then
2:     for each word w in list do
3:         if ‖w‖ is biggest than other words' in list then
4:             add w into filtered
5:         end if
6:     end for
7: end if
```

Figure 4.10: Vertical strategy.

Both strategies are traversing all the counterexamples in the *list*, filtering the ones satisfied the conditions, and adding them into a new set. Note that there could

be more than one filtered counterexample satisfied the strategy. In current version of **ALeLaB**, we allow the Teacher to selecting one from *filtered* randomly.

In Example 4.4.1, different strategies select different counterexamples in the same set of counterexamples.

Example 4.4.1. Let {*abbb*, *baab*, *ab*<1>*b*<*a*>, <<*ab*>>} be a set of counterexamples, the Teacher selects one of them as reply to the Learner.

- Under horizontal strategy, the Teacher would prefer to reply *abbb* or *baab*.
- Under vertical strategy, the Teacher would prefer to reply <<*ab*>>.

We guess that these two strategies affect the efficiency of the learning progress. We make experiments and analyse them in Chapter 5.

4.5 Testing ALeLaB

Testing is important part of software development. Since we have a crystal clear idea about the each functionality of the components of **ALeLaB**, we use the white-box testing. We design the tests with the tool Junit. The Junit allows we test single method separately and counts the coverage in an easy way. We make sure that the tests cover the 100 percent of the methods, and in further with a high condition combination coverage. This section shows the main tests on the functionalities of **ALeLaB**.

Firstly, we discuss how we tested the input interface. Then, we focus on the tests of the functionalities of main components separately. Finally, we present the testing phases of the strategies for selecting counterexamples.

4.5.1 Testing the input interface

Section 4.2 introduces the data specifications and the input formats. To support diagnosis, **ALeLaB** validates inputs and alerts the user when invalid or illegal data are used. First, we test the functionalities to validate inputs. The input interface of

ALeLaB is a command line interface. Figure 4.11 shows the snapshot of a typical session in eclipse platform; the line after the ":" prompt the user for some input.

```
Before starting learning process, please input the target language and an initial alphabet.
Please enter the finite initial alphabet (separate by a space):
a b
Please choose how to input the language:
 1. enter a canonical expression.
 2. enter a file path.
 you choose 1 or 2:
1
Please enter a canonical expression:
a(<1>+ab)
Please select a strategy for counterexamples:
 1. HORIZONTAL.
 2. VERTICAL.
 you choose 1 or 2:
1
```

Figure 4.11: The example of user interface.

Following Figure 4.11, the user is first asked to enter the finite alphabet (letters 'a' and 'b' in example); this information is used to initialise the parameter `alphabet` in the constructor of the class `Teacher`. Then, the user is asked to decide the format to represent the language; this input is used to set the parameter `choice` in the constructor of the class `Teacher`. Depending on the user's preference, the next input is a representation of the language (a nominal regular expression in the example of Figure 4.11) and finally, the user choose a strategy for counterexamples; these inputs are respectively assigned to the parameter `language` and the parameter `strategy` in the constructor of the `Teacher` class.

A simple validation of the inputs `choice` and `strategy` is given in Table 4.2 that show how exceptions are raised on invalid values.

`choice`	`strategy`	Expected results	Results
1	1	pass	no messages for errors
2	3	An exception	An exception of "unknown strategy"
3	N/A	An exception	An exception of "not provided"

Table 4.2: An example of test results of the input interface.

The first row of the table is a valid input; the result has no messages for errors as expected. The second row tests the interface when the user enters an invalid value;

in this case, **ALeLaB** pops out an error message. Likewise, when an invalid value is entered for the `choice` parameter.

Once the input data is validated, **ALeLaB** produces an instance `teacher` of class `Teacher`. We now discuss the testing of the object `teacher(...)`. The object `teacher` sets up the finite alphabet and builds an automaton for the language in input using the parameters `choice`, `alphabet`, and `language` and using the value of `strategy`. Following the session of Figure 4.11, **ALeLaB** executes the following assignment:

```
teacher = new Teacher(choice, alphabet, Strategy.HORIZONTAL, language);
```

(where `s = Strategy.HORIZONTAL`) to set up an object `teacher`. And then, we assert the results. The attribute `teacher.teacherAutomaton` is the ideal result of tests at this stage. The method

```
teacher.getAutomaton().visualisation();
```

produces a human-friendly representation of `teacherAutomaton`, the automaton that the `teacher` object uses to internally represent the input language. We tested this functionality in the following cases.

Canonical Expression. If inputting a language as a canonical expression, we design the test cases depending on the induction of the number of operators in the expressions. The first case is that the language contain a letter which is not contained in the finite alphabet. This is an illegal situation. **ALeLaB** alerts the user as per Table 4.3.

`alphabet`	`language`	Expected results	Results
a	b	an exception	an exception for "unknown letters"
a, b	a	an exception	an exception for "unknown letters"
a	1	an exception	an exception for "unknown letters"

Table 4.3: An example of test results of calculating inputs (part 1).

The functionalities validating the inputs also check that no illegal symbols are used in the inputs for the alphabet or the language. For instance, in this version of **ALeLaB**,

- the symbol "," is not allowed on strings in input, or
- numbers are only permitted in binders of regular expression (since they represent names).

We tested that this sort of invalid data are flagged by **ALeLaB** with proper messages to the user.

Other basic tests are shown in Table 4.4. The firs row is the case that the language is an empty language. The second is that the language only contains an empty string. The last one is that the language only contains any one string of length 1.

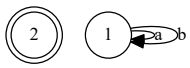
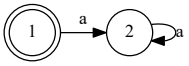
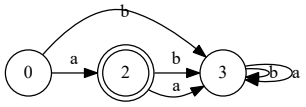
`alphabet`	`language`	Expected results	Results
a b		pass	
a	0	pass	
a b	a	pass	

Table 4.4: An example of test results of calculating inputs (part 2).

Running the tests about the input information as first two columns in Table 4.4, **ALeLaB** generate instances of `Teacher` class as expected. Then, we call the method `visualisation` to view the graph of the `teacherAutomaton` as the last column in Table 4.4. To compare the language and the graph of the result in each row, we can see the success of functionalities in these cases intuitively. **ALeLaB** constructs the minimal automata accepting the input languages in the basic cases.

Next, we test the cases of one or more operations in expressions as in Table 4.5. We also measure the graph results of the `teacherAutomaton` with the input lan-

guages. The first three rows show the one operation case of the concatenation, the union and the Kleene-star. The fourth row shows the case of one pair of binders in an expression.

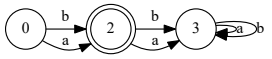
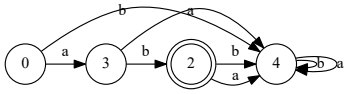
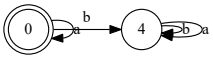
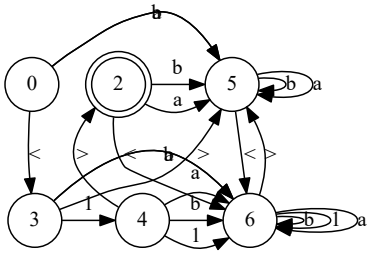
`alphabet`	`language`	Expected results	Results
a b	a+b	pass	
a b	ab	pass	
a b	a*	pass	
a b	< 1 >	pass	
a b	a(< 1+a*b< 2 >>)	pass	see Appendix A.3

Table 4.5: An example of test results of calculating inputs (part 3).

Further, the last row shows the case of an expression mixing the operations and binders. Constructing the expression of more than one operations need to be careful with the order of these operations. The last result in Table 4.5 shows **ALeLaB** produces the operations and binders in a correct order.

Descriptive Text File. Another way of inputting languages is to provide a descriptive text file as introduced on page 48. **ALeLaB** allows the input files describing a deterministic automaton or a non-deterministic automaton. As in Table 4.6, the first row shows the contents of the two descriptive text files. These two files describe

the automata accepting the same language which represented by an expression a+b. One file describes the non-deterministic automaton accepting the language. One file describes the deterministic automaton accepting the language.

From the column of the results in Table 4.6, we can see **ALeLaB** produces correct automata accepting the input languages. Furthermore, **ALeLaB** produces a same automaton to the different descriptive data of the same language.

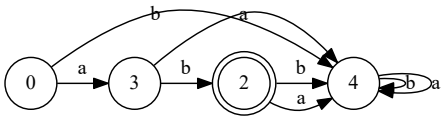
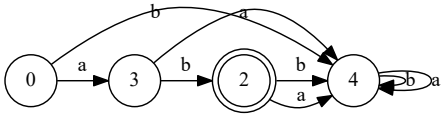
Files	Expected results	Results
initial 0 final 2 0 a 2 0 b 2	pass	
initial 0 final 2 0 a 2 0 b 2 2 a 3 2 b 3 3 a 3 3 b 3	pass	

Table 4.6: An example of test results of calculating inputs (part 4).

4.5.2 Data interaction testing

The Learner learns correctly depending on the answers of the queries. So, **ALeLaB** should ensure the Teacher generates the answer correctly. In this section, we test the functionalities of generating the two kinds of answers.

Membership queries The answers to membership queries are calculated in the method `getAnswer(word)`. We design Junit testing cases in the following template:

```
@Before
public void setUp() throws Exception {
    alphabet.add("a");
    alphabet.add("b");
    String testCRE="a<1>";
```

```
    teacher1=new Teacher(Teacher.FromRE, alphabet, Strategy.Horizontal,
        testCRE);
    @Test
    public void testgetAnswer() {
        String word="";
        assertEquals(teacher1.getAnswer(word), State.Type.FINAL);
    }
}
```

We change the value of the variable `word` and the second parameter of the method `assertEquals` to test the functionality of the method `getAnswer`. Given the assignment `String testCRE="a<1>"`, we show some test results in Table 4.7.

`word`	second parameter	Expected results	Results
a< 1 >	`State.Type.FINAL`	pass	pass
a	`State.Type.PREFIX`	pass	pass
b	`State.Type.SINK`	pass	pass

Table 4.7: An example of test results of the method `getAnswer`.

The first column is the value of the variable `word`. The second column is the value of the second parameter of the method `assertEquals`. The third column is the expected results of running the test unit. The last column is the results of the test unit. For example of the first row, the word a< 1 > is in the given language. So, the expected result is "pass". And the real result is "pass" so that the functionality of the case about an accepted word is achieved. Certainly, we change the value of the variable `test CRE` to test the method under the different languages. We get results as expected.

Equivalence queries The answers to membership queries are calculated in the method `checkAutomaton(automaton)`. Similar to the test on membership queries, we design a new template of test units:

```
@Test
public void testcheckAutomaton() {
    String testlanguage="";
```

```
    Automaton automaton=new BuildAutomaton(testlanguage,alphabet);
    assertEquals(teacher1.checkAutomaton(automaton), null);
}
```

In this case, we change the value of the variable `testlanguage` to generate different automata. Then, we assert the results of the method `checkAutomaton`. If the `automaton` equals to the `teacher1.teacherAutomaton`, the result of the method `checkAutomaton` is `null`. Otherwise, the result is a string. We fix the expected result of the assertion as `null`. When the automata are not equal, the test fails. Given that `String testCRE="a<1>";`, we show some test results in Table 4.8.

`testlanguage`	Expected results	Results
a< 1 >	pass	pass
a	fail	fail

Table 4.8: An example of test results of the method `checkAutomaton`.

The first row shows the case of the equivalence. Two automata are both accepting the same language. And, the test passes as expected. The second row is that two automata are accepting two different languages. The result of the method `checkAutomaton` is not `null` so that the test fails. But, the failure is what we want. Thus, the functionality is achieved.

4.5.3 Testing strategies

To test the different strategies for returning counterexamples, we need the methods `checkAutomaton` and `setStrategy`. Given a parameter `automaton` and the instance `teacher`, we set `teacher.strategy` by invoking the method `setStrategy` and call the method `checkAutomaton` to see the results under different strategies. Accordingly, we set up tests as done in the following example:

```
...
        String testlanguage="a";
        String testCRE="a+a<1>+aaa";
        Automaton automaton=new BuildAutomaton(testlanguage,alphabet);
```

```
        teacher1=new Teacher(Teacher.FromRE, alphabet, Strategy.
            HORIZONTAL, testCRE);
        String result1=teacher1.checkAutomaton(automaton);
        teacher1.setStrategy(Strategy.Vertical);
        String result2=teacher1.checkAutomaton(automaton);
...
}
```

The results are shown in Table 4.9 and show that different strategies yield different counterexamples. As expected, the result under horizontal strategy have less binders than the result under vertical strategy. Certainly, one example is not enough to show the functionalities. Thus, we produce more test cases by changing the the variable `testlanguage` and `testCRE`.

Strategy	Results
Horizontal	aaa
Vertical	a< 1 >

Table 4.9: An example of test results of `checkAutomaton` under different strategies.

Chapter 5

Experiments

In Chapter 3, we analysed the termination of our learning progress. As for Angluin's $\mathsf{L}^\star$ algorithm, a main factor for the convergence of our learning process is the determination of counterexamples. We described two strategies for selecting counterexamples in Chapter 4. Due to possibly infinite number of candidate counterexamples in both strategies, the selection of the counterexamples in is non-deterministic. Therefore, we designed some experiments in order to empirically evaluate the effectiveness of our implementation. For this evaluation we considered two factors affecting the efficiency of **ALeLaB**: the number of fresh name in counterexamples and the length of counterexamples. In our experiments we measured the execution time, the number of iterations, and number of membership queries.

5.1 Experimental Settings

Our experiments are designed to address the following research questions:

1. What is the impact of different alphabets on the effectiveness of each strategy?
2. What is the impact of each strategy on different operators?
3. What is the impact on learning a language of using different strategies?

According the analysis in Section 3.4, it is clear that the complexity of the learning process is attributed to the cardinality of the alphabet, the length of the

counterexamples and the number of the states of the Teacher's automata. Among the three parameters, the first one and the last one are associated directly with the given language, and the length of the counterexample is mainly corresponding to the algorithm of finding counterexamples. Thus, we have to classify the languages, the alphabet and the algorithm of finding counterexamples if we want to analyse the efficiency of our implementation.

As Gruber and Holzer [23] develop the relationship among the descriptional complexity of the regular languages, the expression size and the connectivity of digraphs. It provides an approach to classify the scale of the languages. The total number of occurrences of letters in the alphabet in the languages or the expressions, *alphabetic width*, is a basic notion. They denote the basic notion by $alph(r)$ for a regular expression, and $alph(L)$ for a regular language. In further, they discuss the lower bounds on alphabetic width of language operations. In a sum, we can learn that the size or scalability of a regular language or a regular expression is on the size of the alphabet, the alphabetic width and the alphabetic width of the operations. Thus, we could classify our experiments from

- The *size* or *alphabetic width* of the given languages,
- the *star height* of the given languages,
- the size of the alphabet and/or the names of the given languages,
- the alphabetic width of the language operations.

Besides, the strategies are our main aim. Each class of experiments are including the comparison about the different strategies. Furthermore, we found the lack of our implementation in computing a large scale language and made an improvement to overcome it. We use the "P" symbols in finding counterexamples and have a noticeable speedup in large scale experiments.

A good implementation is to take less time and space. As an output, we have to take the run-time and memory into account. However, the implementation combines the Learner's process and the Teacher's process. The best is to counter them separately. From the theoretical explanation, we know that the Learner produces a

huge number of the membership queries and the Teacher takes most time on finding a proper counterexample. Thus, the benchmarks are considered from

- the number of the membership queries,
- the number of the equivalence queries,
- the total run-time,
- (optional) the time of Learner/Teacher process.

In our implementation, the counterexamples are the decisive variable to the number of instructions which the program executes for producing the next equivalence query. Namely, different counterexamples would lead to a different number of instructions or memory usage in order to update n-observation table and hence to produce membership or equivalence queries. The execution time is an obvious benchmark against which to determine the time-efficiency of the implementation. For space-efficiency, our experiments measure the number of the equivalence queries and the number of the membership queries as benchmarks. Note that this boils down to consider the size of n-observation tables. The number of membership queries corresponds to the size of the n-observation tables: the bigger the size of the n-observation table, the higher the memory usage.

For an empirical analysis, the choice of input instances is of paramount importance. In our case, reviewing the description of the strategies to select counterexamples, we determine as important factors for the input instances are the number of the nested binders/names in a word distinguishes the strategies. The words are limited by the input languages and alphabets.

5.1.1 Varying the finite alphabet

In our first round of experiments we consider our first research question: how alphabets influence the efficiency of the two strategies. In the experiments, we fix the input language and the strategy. This amounts to the following kind of experimental setup:

```
 @Before
 public void setUp() throws Exception {
        strategy= Strategy.Horizontal;
        language="ab<1>";
...
```

The example above describes an experimental setup where the horizontal strategy is used on the language $ab\langle 1\rangle$.

Experiments are encoded as objects where size of the input alphabet is increases by adding extra elements:

```
@Test
public void experiment1(){
...
        alphabet.add("a");
        alphabet.add("b");
...
}
@Test
public void experiment2(){
...
        alphabet.add("a");
        alphabet.add("b");
        alphabet.add("c");
...
}
```

The experimental data collected are the number of equivalence queries, the number of equivalence queries, and the execution time of the instances of the `Learner` class.

5.1.2 Varying operators

Another round of experiments considers the structure of the language. Our implementation exploits the syntactic structure of nominal regular expressions to manipulate languages. For instance, we need to determine the equivalence of a regular expressions representing the input language and an automaton. Therefore, the implementation efficiency depends on the operators used in the regular expressions

describing the input languages. Thus, we fix `choice=Teacher.FromRE` as a condition. Then, the experimental setup can be specified as follows:

```
@Before
public void setUp() throws Exception {
        strategy= Strategy.HORIZONTAL;
        alphabet=alphabet1;
...
```

In this set of experiments we focus particularly on the concatenation operator. More precisely, maintaining fixed the strategy and the alphabet, we vary the input language by concatenating nominal regular expressions. For instance, in the experimental set up above we run our experiments on languages a, aba, b^*, $a\langle 1\rangle$, and $a\langle 1\rangle b^* a\langle 1\rangle$.

Additionally, another set of experiments considers the binder operator. This time we vary the input language by increasing the nesting structure of binders while maintaining fixed the strategy and the alphabet. For instance, we consider the input languages ab^*, $\langle ab^*\rangle$, $\langle\langle\langle ab^*\rangle\rangle\rangle$, and $\langle\langle\langle\langle\langle ab^*\rangle\rangle\rangle\rangle\rangle$.

Finally, we considered similar experiments for the union operator. Again in the same experimental set up as above, we vary the language by composing them with $\_+\_$ operator. For example, we consider the input languages a, $a+b$, and $a+b+\langle 1\rangle$.

In this set of experiments, we want also to figure out the performance of **ALeLaB** with the larger and complex languages. Therefore, we measured the efficiency of the strategies on the regular nominal expressions obtained by shuffling all the operators.

5.1.3 Varying strategies

The comparison on the different strategies is attained by instantiating teacher objects that follow different strategies. For example, the tests sets up with

```
@Experiment
experimentCompareStrategy(){
        teacher1=new Teacher(Teacher.FromRE, alphabet, Strategy.
            HORIZONTAL, language);
    ...
```

```
        teacher2=new Teacher(Teacher.FromRE, alphabet, Strategy.VERTICAL,
            language);
    ...
}
```

collects data associated with the instances of `teacher1` following the horizontal strategy and `teacher2` for the vertical one.

5.1.4 With "P" symbols

We realised the lack of our implementation on computing the large scale examples when we got the reviews after submitting the paper [56]. Before the paper, the "P" symbols are used to optimise the n-observation table reducing the numbers of the meaningless membership queries. When we found the lack is caused by the amounts of meaningless operations in finding counterexamples, we wondered whether the "P" symbols is useful. With attempts, we found the proper position of the "P" symbols in the statements of finding counterexamples. Thus, we produce new tests using "P" symbols and compare with the previous tests.

5.2 Experimental Results Without "P" Symbols

Our experiments aim at illustrating the efficiency of the implementation on the languages and the strategies for counterexamples. We have investigated the tests of the different solutions. Each solution have in common that the benchmarks are the same. They are performed using the benchmarks a selection of inputs values and evaluation values. For each test, we provide the finite alphabet (Alphabet), the input language (Expressions), the alphabetic width of the language (Width), the number of the states of the automaton (State), the selected strategy for counterexamples (Strategy), the time in milliseconds of the process (Time), the number of the equivalence queries (#EQ) and the number of the membership queries (#MQ) during the learning process. The number of the equivalence queries and the membership queries have impacts on both the time complexity and space complexity. Moreover, we also consider the finite input alphabet (Alphabet) where appropriate.

The values of these parameters is determined by taking the average of the values measured on 20 runs of the experiments. In the figures, we omit the data of fixed parameters.

5.2.1 Data: Varying the finite alphabet

In this section we illustrate the performance of tests of varying the finite alphabet. For this reason, it is sufficient to test the Time, #EQ, and #MQ side with the different finite alphabet. In the remainder, we briefly introduce the considered scenarios in the following Tables: the same given languages, the increasing size of the alphabets and the two strategies.

Alphabet	Strategy	Time	#EQ	#MQ
a b	HORIZONTAL	168	2	97
a b c	HORIZONTAL	167	2	126
a b c d	HORIZONTAL	166	2	144
a b	VERTICAL	163	2	99
a b c	VERTICAL	164	2	122
a b c d	VERTICAL	167	2	141

Table 5.1: Varying the alphabet on $\langle ab^* \rangle$

Alphabet	Strategy	Time	#EQ	#MQ
a b	HORIZONTAL	4138	5	1860
a b c	HORIZONTAL	28002	5	2051
a b c d	HORIZONTAL	156685	5	2282
a b	VERTICAL	3986	4	1940
a b c	VERTICAL	37563	4	2458
a b c d	VERTICAL	146477	4	2627

Table 5.2: Varying the alphabet on $a(\langle 1 + \langle 2 + ab^* \rangle \rangle + aa)$

In total 600 times tests cases were run and the distribution of the tests is depicted in Table 5.1 and Table 5.2 The last two columns report the values of #EQ

and #MQ respectively. The figures on the former clearly show that varying the alphabet has no impact on the equivalence queries (as expected since the decision to ask for an equivalence query depends only on the consistency and closedness of the n-observation table). The situation is different for number of the membership queries, #MQ. We can observe that the size of alphabet affect the learning progress in the value of #MQ. Augmenting the alphabet requires yield an increase on the membership queries. For non-trivial language this significantly impacts on the execution time as reported in Table 5.2; in fact, there is a blow up in the execution time whenever a new element is added to the alphabet.

In conclusion, the size of the input alphabet impacts on the time complexity and the space complexity. The reason for the inefficiency is due to the fact that **ALeLaB** has to build deterministic automata when minimising which grow due to the extra (immaterial) transitions covering the extra letters in the alphabet (which do not play any role in the language).

5.2.2 Data: Varying operators

We now consider our second experimental setting were we vary the input language. As discussed in Section 5.1.2, we have investigated the tests of the *star height* of given languages, the size of the name of the given languages and the alphabetic width of the language operations. For this reason, we classified 5 classes of experiments:

- the alphabetic width of the language operations on concatenation,
- the *star height* of the given languages,
- the size of the name of the given languages,
- the alphabetic width of the language operations on union,
- the alphabetic width of the language operations on mixing,

All the tests are fixed the input alphabet. Then, we compare and analyse the data on benchmarks.

In total 500 times test cases were run for the concatenation case. Table 5.3 shows the typical experimental results.

Expression	Strategy	Time	#EQ	#MQ
a	HORIZONTAL	162	1	15
aba	HORIZONTAL	165	2	41
abababaaba	HORIZONTAL	196	2	1133
a	VERTICAL	162	1	15
aba	VERTICAL	166	2	41
abababaaba	VERTICAL	188	2	1133

Table 5.3: Experimental results for concatenation on both strategies.

Our experiments confirm that concatenation increases linearly execution time of both strategies. The effect on the equivalence queries is negligible, while for membership queries the situation is different. Let us focus on the last column reporting the results for #MQ.

For both the horizontal and the vertical concatenation may significantly impact on membership queries. Unsurprisingly, execution time follows the trend of #EQ and #MQ. Comparing the results for column Time, they are increasing while the number of the concatenation operators is increasing under a same strategy. Comparing Time for the two strategies, we see that the execution time is similar.

Next class, the *star height* of the given languages is the sufficient variable side with the same alphabet and the same size of the name. In total over 600 times test cases were run and the results of which are in Table 5.4. Every two rows are a group for comparison.

Expression	Strategy	Time	#EQ	#MQ
b	HORIZONTAL	162	1	15
b*	HORIZONTAL	239	1	8
a< 1 >	HORIZONTAL	187	2	167
a<1*>	HORIZONTAL	169	2	100
a< 1 >ba< 1 >	HORIZONTAL	312	2	1042
a< 1 >b*a< 1 >	HORIZONTAL	228	2	816
b	VERTICAL	162	1	15
b*	VERTICAL	243	1	8
a< 1 >	VERTICAL	178	2	167
a<1*>	VERTICAL	173	2	105
a< 1 >ba< 1 >	VERTICAL	294	2	1042
a< 1 >b*a< 1 >	VERTICAL	230	2	814

Table 5.4: Focusing on the Kleene-star

Here every even row is obtained by adding the star operator in some subexpression of the previous row. For instance, the fourth row adds the Kleene-star operator in the scope of the binder of the expressions on the third row. Starting from the first row and comparing every two consecutive rows of Table 5.4, we see that the values of columns Time and #MQ clearly indicate that the Kleene-star operator generally requires less resources. Contrasting the two different strategy we see that there is no significant impact in respect to the Kleene-star. This is somehow expected since the Kleene-star can introduce loops only "layer-wise" given the structure of nominal automata. In other words, a loop cannot involve states on different layers that would involve allocation and deallocation transitions.

The third class of experiments focus on the size of the names. We now consider experiments where the binding structure of expressions increases. In total over 500 times test cases were run and the distribution of the tests is depicted in Table 5.5.

Expression	Strategy	Time	#EQ	#MQ
ab	HORIZONTAL	164	2	35
<ab>	HORIZONTAL	172	2	167
<<<ab*>>>	HORIZONTAL	2050	2	513
a<a<b*<1>>>	HORIZONTAL	56666	2	871
a<a<b*<a>>>	HORIZONTAL	62866	2	885
ab	VERTICAL	168	2	35
<ab>	VERTICAL	173	2	167
<<<ab*>>>	VERTICAL	2065	2	470
a<a<b*<1>>>	VERTICAL	58030	2	799
a<a<b*<a>>>	VERTICAL	73330	2	863

Table 5.5: Focusing on binders

Looking at the column on #EQ, we observe that the nested binders do not have a significant affect on the number of the equivalence queries. Similar to previous experiments' results, the nested binders have impacts on the figures on column Time and #MQ. Comparing the results for the different strategies, the execution time under vertical strategy is more than the one under horizontal strategy in a larger expression (rows four and nine in Table 5.5).

We can observe that there is a huge difference between the execution time of rows 3 and 4. This can be explained by noticing the following facts. For the expression on row three, at the highest level the automaton uses an alphabet of five symbols (the letters a and b plus the three freshly generate names) and the languages uses two such letters (namely a and b). For the expression on row four, at the highest level the automaton uses an alphabet of five symbols as before, but the languages uses only one of such letters (namely the first freshly allocated name). Hence we have a phenomenon similar to the one highlighted in Table 5.2.

In conclusion, both two strategies have no significant affect on the number of equivalence queries. But in terms of the execution time and the number of membership queries, the vertical strategy takes less resources in average than the horizontal strategy.

In a sequence, we now focus on the union operators in expressions. Thus, we specified 10 kind of test blocks have been used in over 500 times test running for the increasing union operators. Then, we compare and analyse the data on benchmarks. Table 5.3 shows a group of the interesting experimental results on a fixed alphabet $\{a, b, c, 1\}$.

Expression	Strategy	Time	#EQ	#MQ
a	HORIZONTAL	161	1	21
a+b	HORIZONTAL	162	1	21
a+b+c	HORIZONTAL	167	1	21
$< 1 >$	HORIZONTAL	185	2	105
a+b+$< 1 >$	HORIZONTAL	202	2	128
a	VERTICAL	174	1	21
a+b	VERTICAL	166	1	21
a+b+c	VERTICAL	165	1	21
$< 1 >$	VERTICAL	176	2	105
a+b+$< 1 >$	VERTICAL	179	2	128

Table 5.6: Expanding expressions by union

The figures of expressions a, a+b, and a+b+c are extremely similar whatever the strategy is. If only looking at these figures, it seems that the expressions with different union operators have same results. The binders bring something different in the figures on Time. They are in line with expectations. The time of the expressions with binders is more than those with no binders. However, the differences, in terms of the number of union operators, are not clear.

Finally, we want to go further with the experiments. A alphabetic width of the mixed operators is considered. Since the Kleene-star is a special case of the operators, we did not take it into account. For this reason, the class of test cases is mixing the union, concatenation and binders. As in Table 5.7, the distribution of the test cases over the alphabet $\{a, b, 1\}$ is depicted.

Expression	Strategy	Time	#EQ	#MQ
a+b	HORIZONTAL	162	1	21
a+aa+bb+b	HORIZONTAL	170	2	197
a+b+< 1 >	HORIZONTAL	202	2	128
aa+<1a+bb+<a>>	HORIZONTAL	20756	4	1186
a+b	VERTICAL	166	1	21
a+aa+bb+b	VERTICAL	172	2	173
a+b+< 1 >	VERTICAL	179	2	128
aa+<1a+bb+<a>>	VERTICAL	27562	4	1773

Table 5.7: Mixed operators

The main difference from the previous results is in column #EQ. We can see the number of equivalence queries increases as the number of union operators in expressions increases. In this case, the space complexity is represented in both the numbers of the equivalence queries and the membership queries. Comparing the figures on columns Time, #EQ, and #MQ, in terms of strategies, we observe that the horizontal strategy takes less resources than the vertical strategy on the expressions with increasing number of the union operators.

5.3 Experimental Results With "P" Symbols

The previous set of experiments are designed to analyse the effectiveness of strategies among the simple and small scale languages. Actually, we tried more larger scale test cases but those tests cases ran over 2 days without any foreseeable finish point. Hence, we made attempts to improve the implementation. Recently, we have found that the "P" symbols used in finding counterexamples can rapidly reduce the number of the operations and the time.

In this section, we present the typical experimental results with using the "P" Symbols.

First of all, we show the speedup of the improved experiments. The typical example is the one we used in the presentation of the ICE 2019 conference. Given

language $L = (li_1\langle n.(n\ readfeed\ +\ updateProfile\ (li_2\langle m.(m\ isvalid\ update\ +\ n\ delete)lo_2\rangle)^*)^*lo_1\rangle)^*$, if we use the approach without using "P" Symbols, the test gets stuck over 2 days without any new progress. However, when we implement the test case with the "P" Symbols, the correct result is output in 1.628 seconds. More benchmarks are taken into account

- the total closedness rounds: 4
- the total consistency rounds rounds: 8
- the total membership queries rounds: 13811
- the equivalence queries rounds: 5

We present a reasonable comparison between the two approaches in Figure 5.1. The considered scenarios of the comparison is

- the language is $a(\langle 1 + \langle 2 + ab^*\rangle\rangle + aa)$,
- all test cases were both compared under the same stratetgy and the same alphabet,
- the data of test cases were the average data over 20 times,
- the first case is without the "P" Symbols and the second is with the "P" Symbols.

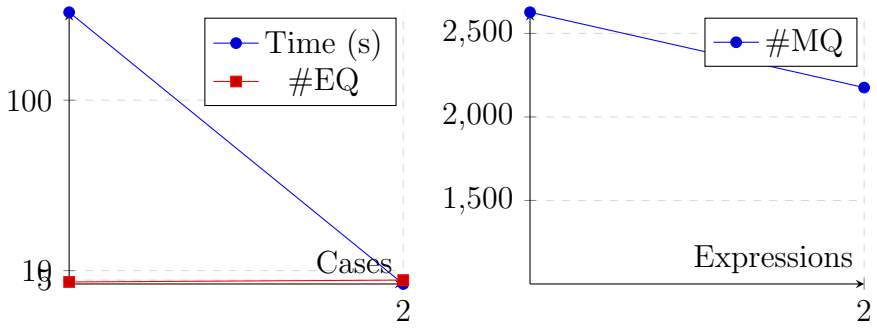


Figure 5.1: The comparison between two approaches.

The line of the Time in Figure 5.1 shows the rapid reduction from case 1 to case 2. However, the number of the equivalence queries increases and the number of the

membership queries reduces slowly. It proof our guess that the most of execution time is on finding counterexamples. And, the membership queries and the equivalence queries are more reasonable benchmarks for the learning algorithm.

Since the experiments in previous section have investigated the scale factors separately, the experiments with the "P" Symbols focus on the performance of difference strategies on the increasing scale scenarios. In the following, we present recent research data in a graphic way.

We have not make a standard of the scale of the nominal regular languages with binders. We have suspected that a stable factor would be the number of the states of the minimal automaton associated with the given language, considering that the Learner always output a minimal automaton and the automaton is unique. So, we designed the test cases on the baseline of the number of the states.

Recently, over 100 kinds of test cases were developed for the strategies on increasing scales from 7 states to 62 states. A interesting and typical set of performances is depicted in Figure 5.2. We fixed the scenarios as

- size of the finite of the alphabet $\{a, b\}$,
- languages is tested under both two strategies,
- each test case is run at least 20 times.

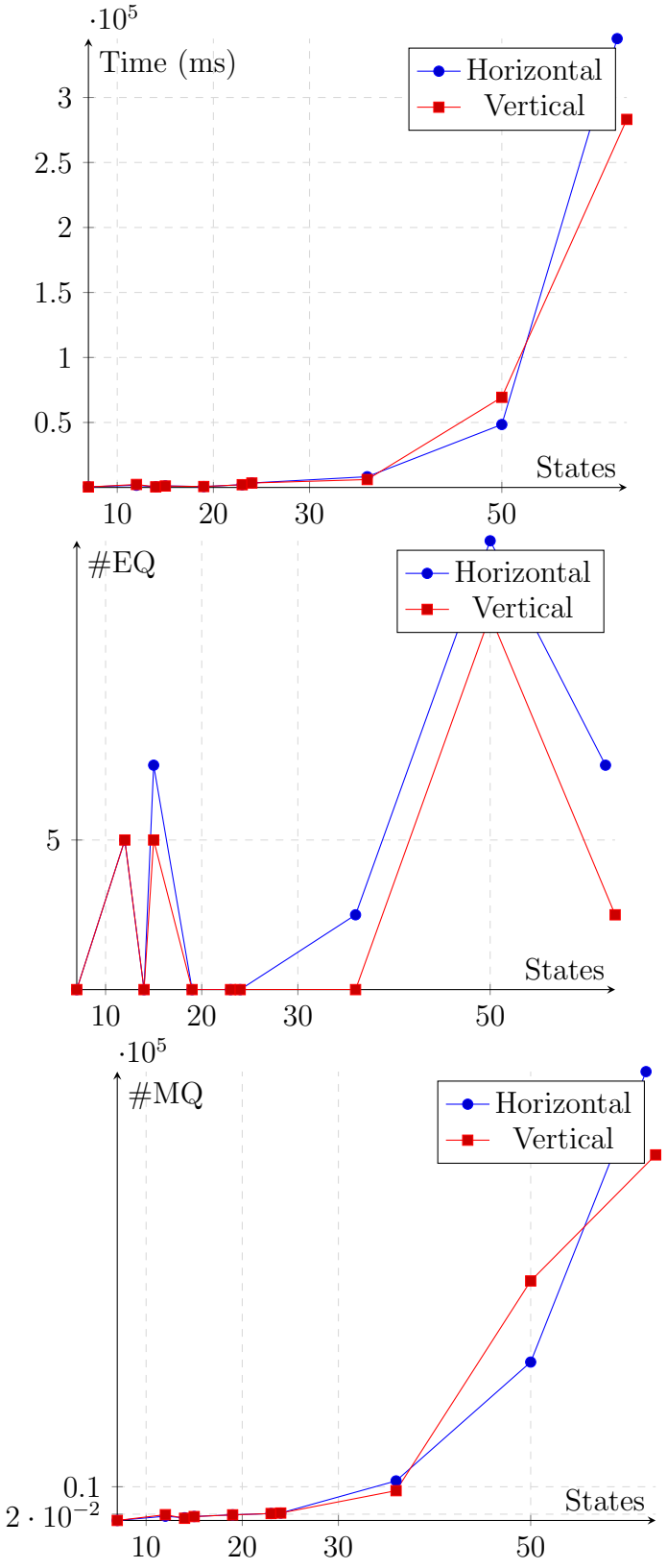


Figure 5.2: The comparison between two approaches on the increasing number of the states.

Looking at the plots of Time and #MQ, they rise exponentially as expected.

However, the trend of #EQ has curves and turns. The curves and turns are caused by the combination of the operators of the languages. For example, the languages $L_1 = a\langle a\langle b^*\langle a\rangle\rangle\rangle$ and $L_2 = a + ba + \langle 1\rangle + \langle\langle 2\rangle\rangle + \langle a\langle 1\rangle\rangle$, L_1 has 14 states and L_2 has 15 states. Under horizontal strategy, Learning L_1 needs 3 equivalence queries and Learning L_2 needs 6 equivalence queries. Currently, we did not find out what affects it clearly because we have enough time to design the experiments and filter out the classification rule. It would be interesting direction of optimising the strategies in the future work.

Comparing the data between strategies, it still have no clear bound on the superiority for most languages. As the case of #EQ, it needs more meticulous conditions on large Quantity of experiments. In the future, we prefer to study on this part and try to mix the strategy when we have some ideas about their fields of advantages.

5.4 Discussion

All experimental cases were repeated over 20 times to take the randomness of the counterexamples satisfied the selected strategies int account, and the results were taken the average values into account. The experiments are designed by induction to analyse the effectiveness of strategies more reasonably from simple cases to complex cases.

In general, we presented some typical experimental results performing the efficiency of the two strategies under several situations in small scale. We can summarise clearly that the vertical strategy takes less resources when the expressions have more concatenation operators, and the horizontal strategy is better to handle with the expressions with more union operators. Further, neither two strategies have no apparent advantages as expressions have the Kleene-star operators. In terms of nested binders, the vertical strategy has an advantage in consuming resources.

However, in the extreme case, with very large sizes, the preference of the strategies cannot be decided easily. For example, in terms of the larger expression in section 5.3, the figures on the #EQ, and the #MQ represent the turns and crosses. Sometime, the vertical strategy need more space for the larger number of queries,

but the consuming time is less, the horizontal strategy takes more time with less queries. And vice versa.

In conclusion, the two strategies have their own advantages in current version. Further, the use of the two strategies can be optimised. In fact, instead of randomly choosing a counterexample, we could apply heuristics to weight the counterexample and choose according to their weights. Hopefully, good heuristics will select counterexamples that make the algorithm to converge more quickly. An approach on adding conditions for counterexamples has been investigated for $\mathsf{L}^\star$ in [2]. We discuss in Chapter 6 how to extend these approaches to nominal regular languages. Also, we could study conditions to predict when one the strategies could outperform the other so that we use mixed strategy. All these optimisations are left for future investigation and discussed in Chapter 6.

5.5 Case Study Blueprint

With enough theoretical experiments, we would think about apply the algorithm for a real application. The verification of protocols and systems are interesting areas [16, 17]. In this section, we state an rough idea about the application of our algorithm on model testing. Everything are just out of theoretical and practical attempts.

Our idea is to extend the approach of inferring models by [55]. Here we use a small application scenario in [55] to explain where our algorithm could be used. The scenario is "Assuming a model with capturing the order in which the methods *recurse* and *basecase*, for each method we print out its name *recurse* and *basecase*, along with any associated data values *depth for recurse*". Running this on four inputs (depth=3,2,5, and 0) gives rise to the traces as Figure 5.3. The raw data can be divided into two parts: input and output. The input are 3,2,5, and 0. And, the output are those in Figure 5.3. Furthermore, the 3,2,5, and 0 are variables, and the recurse and the basecase are constants. The variables are mapping to the names in our algorithm, and constants are the finite alphabet. The operations of input and output are in pared, so they are associated to the binders. We can translate the

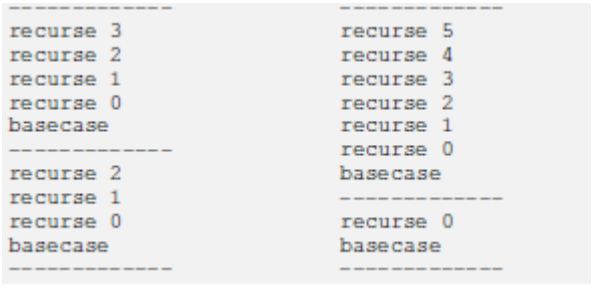


Figure 5.3: The output traces.

four pieces of data into a nominal form as

- $I_1 < 3.\ recurse\ 3\ recurse\ 2\ recurse\ 1\ recurse\ 0\ basecase\ > O_1$
- $I_1 < 2.\ recurse\ 2\ recurse\ 1\ recurse\ 0\ basecase\ > O_1$
- $I_1 < 5.\ recurse\ 5\ recurse\ 4\ recurse\ 3\ recurse\ 2\ recurse\ 1\ recurse\ 0\ basecase\ > O_1$
- $I_1 < 0.\ recurse\ 0\ basecase\ > O_1$

Thus, the data for the Teacher has an outline. Next step is to think about how to store and represent these data as a Teacher. Roughly, we consider that a set is a good data structure and each piece of the data is an element of the set. More precisely, the names should be defined properly. For now, we have no idea about that.

Then, the Learner can get progress with the finite alphabet until the Learner get an automaton containing all the four pieces of data. In this stage, we can not assert the automaton is correct. We suggest an additional step: to do more model tests on the target model and the Learner's automaton, and to collect them to produce an counterexample, then the Learner can keep learning unless there is no counterexamples.

The core techniques of applications of our algorithm are the translation of the raw data, and the additional test data for the equivalence. We have a plan to implement this idea in the following work.

Chapter 6

Conclusions and Future Work

This chapter collects final remarks on this dissertation and discussing some potential research directions.

6.1 Conclusions

The learning algorithm $\mathsf{L}^\star$ was introduced more than thirty years ago and has been intensively extended to many types of models in following years. This algorithm continues to attract the attention of many researchers [2, 52, 39].

In this dissertation, we have designed a learning algorithm for a class of languages over infinite alphabet; more precisely, we have considered nominal regular languages with binders [31, 33]. We have tackled the finite representations of the alphabets, words and automata for retaining the basic scheme and ideas of $\mathsf{L}^\star$. Hence, we revised and added definitions for the nominal words and automata. Further, accounting for names and the allocation and deallocation operations, we revised the data structures and notions in $\mathsf{L}^\star$. Accordingly, we have proposed the learning algorithm, $\mathsf{nL}^\star$, to stress the progress of learning a nominal language with binders. We have proved the correctness and analysed the complexities of $\mathsf{nL}^\star$.

We have developed a implementation of $\mathsf{nL}^\star$ in Java, named **ALeLaB**. This implementation allowed us to study how $\mathsf{nL}^\star$ behaves under different strategies to generate counterexamples.

Indeed, the experiments have been mainly designed for the impact of the two strategies. The benchmarks included the data of input parameters. Besides, we also have taken the behaviour of $\mathsf{nL}^\star$ into account. The analysis of experimental data shows that the advantages of two strategies are various in terms of varying operators mixing with binders.

6.2 Future Work

There are several directions to advance our work. Firstly, one can improve the implementation and extend the case studies for $\mathsf{nL}^\star$. In particular, it would be interesting to analyse $\mathsf{nL}^\star$ using *mixed strategies* to generate counterexamples. Another, more theoretical direction, would be the extension of our algorithm to the non-deterministic case.

Improving ALeLaB Although the current implementation achieved all the functionalities for the learning process of $\mathsf{nL}^\star$, several improvements are possible. The current implementation requires some manual inputs (as shown in Section 4.5.1). Improving on the usability of the tool would be necessary to apply to verification. Also, developing a user-friendly graphical user interface is left for future work. It would indeed be interesting to analyses some case studies. Since the regular expressions with binders could be use for verification, testing and analysis of programming languages [32], we first plan to apply our algorithm to model testing, which comes from the idea of inferring models by [55].

Optimisation of the learning algorithm The different strategies for counterexamples are an attempt to optimisation the learning algorithm. And the evaluation of the experiments data provides the potential for optimisation. In the future, a new strategy balancing the advantages of current two strategies is a desirable setting. As we saw, the effectiveness of $\mathsf{nL}^\star$ and its implementation **ALeLaB** may depend on the algebraic structure of the regular expressions representing the input language. In this respect we envisage two possible interesting directions. One is to identify heuristics based on the algebraic structures that the teacher could use to

generate “better” counterexamples, that is counterexamples that allow the learner to learn “more quickly”. Another direction is to identify classes of nominal regular expressions that make the learning process quicker.

Study on learning non-deterministic automata Like in the classical theory of formal languages, non-deterministic automata are more expressive than deterministic ones. Non-deterministic nominal automata have exponentially less operations with respect to transitions. In the current version of our implementation, we provide one way to input descriptive file about non-deterministic nominal automata. **ALeLaB** can do operations with less input data. This is a good start to the research of an extension of the learning algorithm for non-deterministic cases. Namely, the algorithm can support more situations which have less certain data, and the algorithm would reduce internal instructions with producing non-deterministic models.

Appendix A

Additional Figures

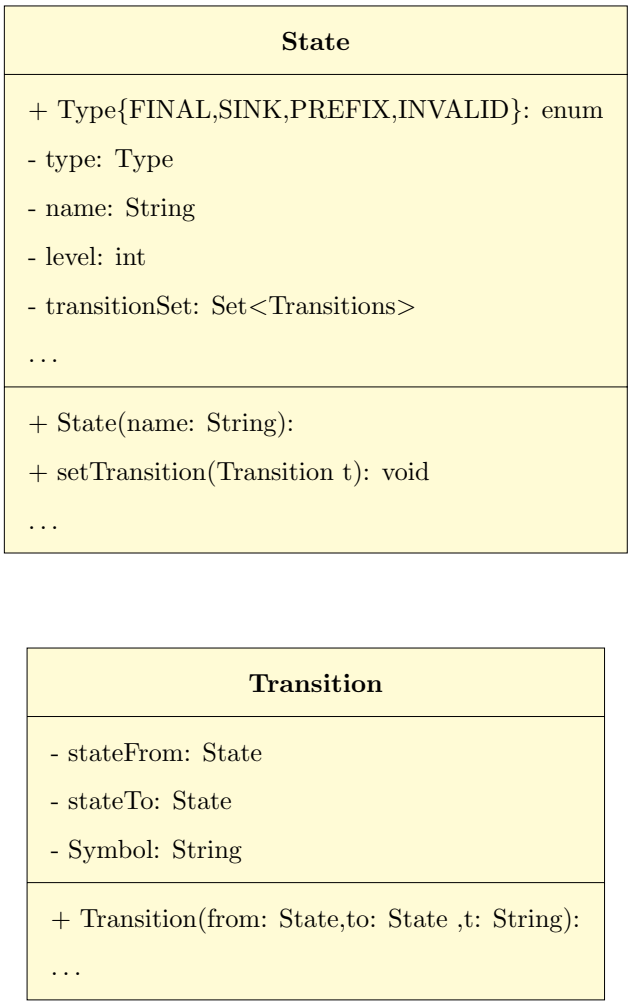


Figure A.1: Essential classes associated to `Automaton` class.

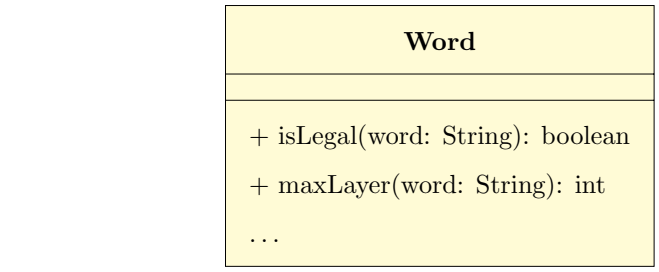


Figure A.2: `Word` class.

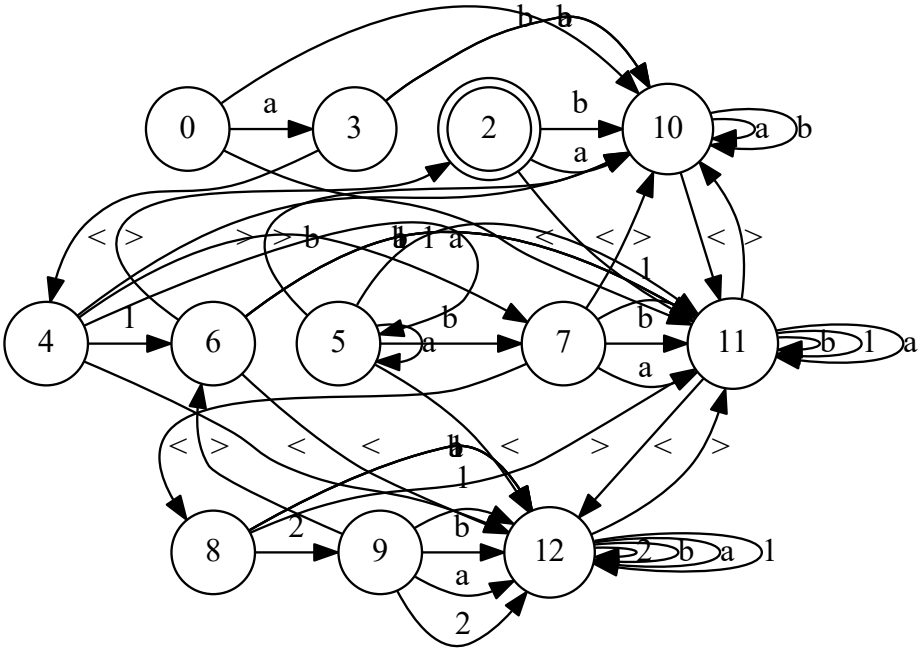


Figure A.3: The graph for the last result in Table 4.5.

Bibliography

[1] Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[2] Dana Angluin and Tyler Dohrn. The Power of Random Counterexamples. In *International Conference on Algorithmic Learning Theory, ALT 2017, 15-17 October 2017, Kyoto University, Kyoto, Japan*, pages 452–465, 2017.

[3] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. Local policies for resource usage analysis. *ACM Trans. Program. Lang. Syst.*, 31(6), 2009.

[4] Mikołaj Bojańczyk, Bartek Klin, and Slawomir Lasota. Automata theory in nominal sets. *Logical Methods in Computer Science*, 10(3), 2014.

[5] Mikolaj Boja'nczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-variable logic on words with data. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 7–16, 2006.

[6] Benedikt Bollig, Peter Habermehl, Martin Leucker, and Benjamin Monmege. A Fresh Approach to Learning Register Automata. In *Developments in Language Theory - 17th International Conference, DLT2013, Marne-la-Vallée, France, June 18-21, 2013. Proceedings*, pages 118–130, 2013.

[7] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. Learning Communicating Automata from MSCs. *IEEE Trans. Software Eng.*, 36(3):390–408, 2010.

[8] Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *JACM*, 30(2):323–342, 1983.

[9] J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Proc. Sympos. Math. Theory of Automata (New York, 1962)*, pages 529–561. Polytechnic Press of Polytechnic Inst. of Brooklyn, Brooklyn, N.Y., 1963.

[10] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Active learning for extended finite state machines. *Formal Asp. Comput.*, 28(2):233–263, 2016.

[11] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972.

[12] Pierpaolo Degano, Gian Luigi Ferrari, and Gianluca Mezzetti. Nominal automata for resource usage control. In *Implementation and Application of Automata - 17th International Conference, CIAA 2012, Porto, Portugal, July 17-20, 2012. Proceedings*, pages 125–137, 2012.

[13] Pierpaolo Degano, Gian Luigi Ferrari, and Gianluca Mezzetti. Towards nominal context-free model-checking. In *Implementation and Application of Automata - 18th International Conference, CIAA 2013, Halifax, NS, Canada, July 16-19, 2013. Proceedings*, pages 109–121, 2013.

[14] Stéphane Demri and Ranko Lazic. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Log.*, 10(3):16:1–16:30, 2009.

[15] Jan Dijkstra, H.J.P. Timmermans, and Joran Jessurun. A multi-agent cellular automata system for visualising simulated pedestrian activity. In *Proceedings of ACRI*, pages 29–36, 01 2000.

[16] Gianluigi Ferrari, Giovanni Ferro, Stefania Gnesi, Ugo Montanari, Marco Pistore, and Gioia Ristori. An automata based verification environment for mobile processes. In Ed Brinksma, editor, *TACAS*, volume 1217 of *LNCS*, pages 275–289. Springer, April 1997.

[17] Gianluigi Ferrari, Stefania Gnesi, Ugo Montanari, Marco Pistore, and Gioia Ristori. Verifying Mobile Processes in the HAL Environment. In *Proc. 10th International Computer Aided Verification Conference*, pages 511–515, 1998.

[18] Gianluigi Ferrari, Ugo Montanari, and Marco Pistore. Minimizing transition systems for name passing calculi: A co-algebraic formulation. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures*, pages 129–143, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[19] Murdoch J. Gabbay. *A Theory of Inductive Definitions with alpha-Equivalence.* phdthesis, University of Cambridge, UK, March 2001.

[20] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax involving binders. In Giuseppe Longo, editor, *LICS*, pages 214–224, Trento, Italy, July 1999. IEEE.

[21] Murdoch J. Gabbay and Andrew M. Pitts. A New Approach to Abstract Syntax with Variable Binding. *J. of Formal Aspects of Computing*, 13(3-5):341–363, July 2002.

[22] Fabio Gadducci, Marino Miculan, and Ugo Montanari. About permutation algebras, (pre)sheaves and named sets. *Higher-Order and Symbolic Computation*, 19(2-3):283–304, 2006.

[23] Hermann Gruber and Markus Holzer. Finite automata, digraph connectivity, and regular expression size. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, pages 39–50, 2008.

[24] Serena Hamilton, A.J. Jakeman, and John P. Norton. Artificial intelligence techniques: An introduction to their use for modelling environmental systems. *Mathematics and Computers in Simulation*, 78:379–400, 07 2008.

[25] John Hopcroft. *An $n \log n$ algorithm for minimizing states in a finite automaton.* Academic Press, New York, 1971.

[26] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1):60–65, March 2001.

[27] Bart Jacobs and Alexandra Silva. Automata learning: A categorical perspective. In *Horizons of the Mind. A Tribute to Prakash Panangaden*, pages 384–406. Springer, 2014.

[28] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.

[29] Michael Kaminski and Tony Tan. Regular expressions for languages over infinite alphabets. *Fundam. Inform.*, 69(3):301–318, 2006.

[30] Dexter Kozen, Konstantinos Mamouras, Daniela Petrisan, and Alexandra Silva. Nominal kleene coalgebra. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, pages 286–298, 2015.

[31] Alexander Kurz, Tomoyuki Suzuki, and Emilio Tuosto. A characterisation of languages on infinite alphabets with nominal regular expressions. In *Theoretical Computer Science - 7th IFIP TC 1/WG 2.2 International Conference, TCS 2012, Amsterdam, The Netherlands, September 26-28, 2012. Proceedings*, pages 193–208, 2012.

[32] Alexander Kurz, Tomoyuki Suzuki, and Emilio Tuosto. On Nominal Regular Languages with Binders. In Lars Birkedal, editor, *Foundations of Software Science and Computational Structures*, pages 255–269, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[33] Alexander Kurz, Tomoyuki Suzuki, and Emilio Tuosto. Nominal Regular Expressions for Languages over Infinite Alphabets. Extended Abstract. *CoRR*, abs/1310.7093, 2013.

[34] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer,

editors, *Computer Aided Verification*, pages 585–591, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[35] maintained by Laurent Blume/John Ellson. Graphviz. `https://http://graphviz.org/`. Accessed: 2019-01-29.

[36] R. McNaughton and H. Yamada. Regular Expressions and State Graphs for Automata. *IRE Transactions on Electronic Computers*, EC-9(1):39–47, March 1960.

[37] Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[38] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i and ii. *Inf. and Comp.*, 100(1), 1992.

[39] Joshua Moerman, Matteo Sammartino, Alexandra Silva, Bartek Klin, and Michal Szynwelski. Learning nominal automata. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 613–625, 2017.

[40] Ugo Montanari and Marco Pistore. π-Calculus, Structured Coalgebras, and Minimal HD-Automata. In Mogens Nielsen and Branislav Roman, editors, *MFCS*, volume 1983 of *LNCS*. Springer, 2000.

[41] Edward F. Moore. Gedanken-experiments on sequential machines. In *Automata studies*, Annals of mathematics studies, no. 34, pages 129–153. Princeton University Press, Princeton, N. J., 1956.

[42] A. Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–541, apr 1958.

[43] Frank Neven, Thomas Schwentick, and Victor Vianu. Towards regular languages over infinite alphabets. In *Mathematical Foundations of Computer Science 2001, 26th International Symposium, MFCS 2001 Marianske Lazne, Czech Republic, August 27-31, 2001, Proceedings*, pages 560–572, 2001.

[44] Oliver Niese. *An integrated approach to testing complex systems.* PhD thesis, Technical University of Dortmund, Germany, 2003.

[45] Gertjan van Noord. Treatment of Epsilon Moves in Subset Construction. *Computational Linguistics*, 26(1):61–76, 2000.

[46] Corina S. Pasareanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M. Cobleigh, and Howard Barringer. Learning to divide and conquer: applying the $\mathsf{L}^\star$ algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32(3):175–205, 2008.

[47] Doron A. Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225–246, 2002.

[48] Marco Pistore. *History Dependent Automata.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 1999.

[49] Andrew M. Pitts. Names and Symmetry in Computer Science (Invited Tutorial). In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pages 21–22, 2015.

[50] M. O. Rabin and D. Scott. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*, 3(2):114–125, April 1959.

[51] Davide Sangiorgi and David Walker. *The π-Calculus: a Theory of Mobile Processes.* Cambridge University Press, 2002.

[52] Lutz Schröder, Dexter Kozen, Stefan Milius, and Thorsten Wißmann. Nominal automata with name binding. In *CoRR*, volume abs/1603.01455, 2016.

[53] Luc Segoufin. Automata and logics for words and trees over an infinite alphabet. In *International Workshop on Computer Science Logic*, pages 41–57. Springer, 2006.

[54] Ken Thompson. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[55] Neil Walkinshaw and Mathew Hall. Inferring Computational State Machine Models from Program Executions. In *2016 IEEE International Conference on*

Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016, pages 122–132, 2016.

[56] Yi Xiao and Emilio Tuosto. On learning nominal automata with binders. In *Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20-21 June 2019*, pages 137–155, 2019.