Active Learning Techniques for Pomset Recognizers

Adrien Pommellet, Amazigh Amrane, Edgar Delaporte
(supervisor: Amazigh Amrane)

January 2025

Pomsets are a promising formalism for concurrent programs based on partially ordered sets. Among this class, series-parallel pomsets admit a convenient linear representation and can be recognized by simple algebraic structures known as pomset recognizers. Active learning consists in inferring a formal model of a recognizable language by asking membership and equivalence queries to a minimally adequate teacher (MAT). We improve existing learning algorithms for pomset recognizers by

1. introducing a new counter-example analysis procedure that is in the best case scenario exponentially more efficient than existing methods
2. adapting the state-of-the-art L^λ algorithm to minimize the impact of exceedingly verbose counter-examples and remove redundant queries
3. designing a suitable finite test suite that ensures general equivalence between two pomset recognizers by extending the well-known W-method.

This document contains the theoretical sections of [1], which contains the proofs of all the claims presented here.

Keywords
active learning, concurrency, pomsets

Laboratoire de Recherche de l'EPITA
14-16, rue Voltaire
94270 Le Kremlin-Bicêtre CEDEX
France

Copying this document

Contents

1 Introduction

Finite state automata are a straightforward model for terminating sequential systems. Runs are implicitly described by a *total* order relation: an execution is merely an ordered, linear sequence of events. However, richer structures may be needed for concurrent programs. Indeed, two threads may be action in parallel, neither of them preceding nor following the other. In this case, runs may be modelled using a *partial* order relation: concurrent events cannot be relatively ordered.

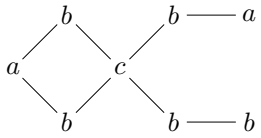

Figure 1: The series-parallel pomset $a(b \parallel b)c(ba \parallel bb)$.

Instead of linear traces, *pomsets* [2] (partially ordered multisets) represent executions of a parallel program. We consider in this article the class of *series-parallel* pomsets that can be linearly described using letters as well as the sequential $\cdot$ and parallel $\parallel$ composition operators. Figure 1 displays the Hasse diagram of the pomset $a(b \parallel b)c(ba \parallel bb)$. Were we to consider the interleaving semantics on parallel threads, series-parallel pomsets are an exponentially more succinct description than finite words: given n distinct letters $a_1, \ldots, a_n$ in an alphabet Σ, each one labelling a different thread, a single pomset $a_1 \parallel \ldots \parallel a_n$ describes the $n!$ possible linearized traces of the interwoven threads.

Active learning algorithms consist in inferring a formal model of a black-box system that can be dynamically queried. Under *the Minimally Adequate Teacher* (MAT) framework, interactions with the black-box system are twofold: *membership queries* that consist in asking whether a given trace can be generated by the system, and *equivalence queries* to determine whether a given automaton accurately represents all executions of the system, yielding a counter-example if the answer is negative.

One of the earliest active learning algorithms is Angluin's L^* [3] for rational languages. It infers the Myhill-Nerode equivalence classes of the target language by maintaining a set of *representatives* of these classes as well as a set of *distinguishers* that separate them. Van Heerdt et al. extended L^* to recognizable pomset languages, and further provided a translation of pomset recognizers to pomset automata.

However, various improvements have been brought to L^* over the years. The length m of a counter-example returned by the MAT being arbitrarily long, it may end up dominating the learning process; Rivest and Schapire [4] therefore introduced an algorithm that infers a new equivalence class in $\mathcal{O}(\log(m))$ membership queries. Moreover, the use of equivalence queries makes little practical sense as it implies that the MAT knows the very formal model of the system we are trying to infer; Chow [5] proved that a finite suite could subsume equivalence of finite automata, provided a bound on the size of the target model is known beforehand. Finally, new algorithms such as TTT [6], $L^\#$ [7], or L^λ [8] have been shown to significantly reduce the number of queries needed compared to L^*.

The point of the article is adapt and extend these improvements to the active learning of pomset recognizers. Our new contributions are the following:

- We introduce a new counter-example handling algorithm that extends the sets of representatives and distinguishers. Its complexity depends on the depth of the counter-example's syntactic tree, rather than its number of nodes.
- We extend Howar et al.'s L^λ algorithm [8] to recognizable pomset languages. L^λ has

been proven to be competitive with state-of-the-art active learning algorithms for rational languages and maintains a prefix (resp. suffix) closed set of representatives (resp. distinguishers), further reducing the influence of the counter-example's maximal length.

- We make use of redundancy-free discrimination trees [9], in the sense that only membership queries that contribute to the distinction of states have to be performed.
- In a similar fashion to Chow's W-method [5], we design a finite test suite that can conditionally replace equivalence queries.

2 Preliminary Definitions

2.1 Series-parallel pomsets

We consider a non-empty finite set Σ of *letters* (or *labels*) called the *alphabet*.

Definition 1 (Poset) *A partially ordered set or* poset $(A, <, \ell)$ *consists of a finite set* A *(called the* carrier*), a strict partial order* $<$ *on* A*, and a labelling map* $\ell : A \to \Sigma$*.*

Two posets $(A, <_A, \ell_A)$ and $(B, <_B, \ell_B)$ are said to be isomorphic if there exists a bijection between A and B preserving ordering and labelling.

Definition 2 (Pomset [10]) *A partially ordered multiset or* pomset *is an equivalence class for the isomorphism relation on posets.*

For convenience's sake, we will treat a representative of such a pomset as if it were the entire class. The empty pomset is denoted ε, and the singleton pomset labelled by $a \in \Sigma$. We say that a pomset $\mathfrak{B}$ is a subpomset of $\mathfrak{A}$ if $\mathfrak{B}$ can be embedded in $\mathfrak{A}$, that is, if there exists an injection from B to A preserving labelling and ordering.

Pomsets can be composed sequentially (in a similar fashion to words) or in parallel fashion. Let $\mathfrak{A} = (A, <_A, \ell_A)$ and $\mathfrak{B} = (B, <_B, \ell_B)$ be two pomsets such that A and B are disjoint (an assumption that applies to the rest of this article).

1. Their *parallel composition* $\mathfrak{A} \parallel \mathfrak{B}$ is $(A \cup B, <_A \cup <_B, \ell_B \cup \ell_B)$. The two pomsets are juxtaposed but cannot be compared.
2. Their *sequential composition* (or concatenation) $\mathfrak{A} \cdot \mathfrak{B}$ is $(A \cup B, <_A \cup <_B \cup (A \times B), \ell_A \cup \ell_B)$. Every element of B is greater than every element of A.

Parallel composition is associative and commutative; sequential composition is merely associative. Both operations share ε as neutral element.

Definition 3 (Series-parallel pomsets) *The set* $\mathrm{SP}(\Sigma)$ *of series-parallel pomsets is the smallest set of pomsets containing* $\{\varepsilon\}$ *and* Σ *closed under sequential and parallel composition.*

A pomset $\mathfrak{A}$ is said to be *N-free* if the pomset $(\{x_1, x_2, x_3, x_4\}, \{x_1 < x_2, x_3 < x_2, x_3 < x_4\}, \ell)$ for some labelling ℓ is not a subpomset of $\mathfrak{A}$. Intuitively, the pattern shown in Figure 2 does not appear in the pomset's Hasse diagram. It is well-known that series-parallel pomsets coincide with N-free pomsets [11].

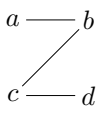

Figure 2: A N pattern.

We denote $\mathrm{SP}^+(\Sigma) = \mathrm{SP}(\Sigma) \setminus \{\varepsilon\}$. For the remainder of this paper, we will refer to series-parallel pomsets as merely pomsets.

The set of *syntactic terms* (or simply terms) ST_Σ over Σ is the set of full binary trees whose inner nodes are labelled by operators in $\{\cdot, \|\}$ and whose leaves are labelled by $\Sigma \cup \{\varepsilon\}$. We associate each term with its isomorphic *linear description* obtained by performing a prefix traversal. We may omit the symbol $\cdot$, consider that $\cdot$ has priority over $\|$, and assume left associativity for syntactic purposes. Intuitively, a term is merely a way to describe a pomset: as an example, $a(b \parallel b)c(ba \parallel bb)$ represents the pomset of Figure 1.

Due to neutrality, associativity and commutativity properties, several terms may describe the same pomset: as an example, $a \parallel b = b \parallel a = a \cdot \varepsilon \parallel b$. Each pomset $w \in \mathrm{SP}(\Sigma)$ may therefore be associated with a set $\mathsf{ST}(w) \subseteq \mathsf{ST}_\Sigma$ of syntactically different but semantically equivalent terms whose interpretation in $\mathrm{SP}(\Sigma)$ is w. The *depth* $\delta(w)$ of w is the minimum of the tree depth function on $\mathsf{ST}(w)$. A term in $\mathsf{ST}(w)$ is said to be *canonical* if its depth is minimal and, assuming $w \neq \varepsilon$, it has no leaf labelled by ε. Note that canonical terms are not unique: $\mathsf{ST}(w)$ may contain more than one canonical term.

If $w \notin \Sigma \cup \{\varepsilon\}$, given a canonical term of w such that its root is labelled by $\circ \in \{\cdot, \|\}$ and its left (resp. right) subtree is a term representing pomset z_1 (resp. z_2), $z = z_1 \circ z_2$ is called a *canonical decomposition* of z. Obviously, $\delta(z_1) < \delta(z)$ and $\delta(z_2) < \delta(z)$. Figure 3 displays a canonical term of $bc \parallel a$ of depth 2.

The use of terms is a consequence of our counter-example handling algorithm and test suite for equivalence queries that manipulate syntactic trees. Nevertheless, most concepts and algorithms on pomsets outlined in this article still remain term-agnostic.

2.2 Pomset recognizers

Definition 4 (Bimonoids [12]) *A* bimonoid $(M, \odot, \oplus, e)$ *is a set* M *equipped with two internal associative operations* $\odot$ *and* $\oplus$, $\oplus$ *being commutative as well, and a neutral element* e *common to* $\odot$ *and* $\oplus$.

These constraints define a *variety* of bimonoids, that is, a class of algebraic structures satisfying the same behaviour (as defined by various equations encoding associativity, commutativity, etc.). Note that there is no distributivity property. The set $\mathrm{SP}(\Sigma)$ endowed with $\cdot$, $\|$ and the neutral element ε is a bimonoid.

A set A *generates* a bimonoid $(M, \odot, \oplus, e)$ if $A \subseteq M$ and any element of $M \setminus \{e\}$ can be obtained by inductively applying $\odot$ and $\oplus$ to A. Moreover, A *freely* generates M if any element of $M \setminus \{e\}$ admits a unique (up to commutativity and associativity) decomposition according to $A \setminus \{e\}$, $\odot$, and $\oplus$.

Theorem 1 (Freeness of $\mathrm{SP}(\Sigma)$ [12]) $(\mathrm{SP}(\Sigma), \cdot, \|, \varepsilon)$ *is freely generated by* Σ *in the variety of bimonoids.*

In particular, all the ε-free terms of the same pomset are equivalent up to commutativity and associativity: as an example, the terms $a \parallel bc$ and $bc \parallel a$ describe the same pomset, and no other ε-free term exists.

We rely on bimonoids to recognize languages of SP-pomsets. As is customary, we define homomorphisms of bimonoids as mappings between two bimonoids preserving identity and both internal operations. Note that, $\mathrm{SP}(\Sigma)$ being freely generated by Σ, any function $i \colon \Sigma \to M$ for some bimonoid $(M, \odot, \oplus, e)$ can be (inductively) extended in a unique way to a bimonoid homomorphism $i^\sharp : \mathrm{SP}(\Sigma) \to M$ so that for all $a \in \Sigma$, $i^\sharp(a) = i(a)$ and $i^\sharp(\varepsilon) = e$. This leads to the following definition of a pomset recognizer [13].

Definition 5 (Pomset recognizer) *The tuple $\mathcal{R} = (M, \odot, \oplus, e, i, F)$ is said to be a* pomset recognizer *(PR) on Σ if $(M, \odot, \oplus, e)$ is a finite bimonoid, $i \colon \Sigma \to M$, and $F \subseteq M$. The carrier M is also called the set of* states *of $\mathcal{R}$. The language of $\mathcal{R}$, denoted $\mathcal{L}(\mathcal{R})$, is the set $\{w \in \mathrm{SP}(\Sigma) \mid i^\sharp(w) \in F\}$. Finally, we introduce the predicate $\mathcal{R}(u) =$ "$i^\sharp(w) \in F$".*

PRs act as bottom-up deterministic finite tree automata on terms: each letter in Σ has an image in a set of states M, that we combine by using the images $\odot$ and $\oplus$ of the operators $\cdot$ and $\|$. Due to the freeness of $\mathrm{SP}(\Sigma)$ and $i^\sharp$ being a homomorphism, we can apply PRs to pomsets, as $i^\sharp(t)$ always return the same result, regardless of the term $t \in \mathsf{ST}(w)$ chosen.

Definition 6 (Recognizable pomset languages) *A set (or* language*) $L \subseteq \mathrm{SP}(\Sigma)$ is said to be* recognizable *if there exists a PR $\mathcal{R}$ such that $L = \mathcal{L}(\mathcal{R})$.*

Example 7 *Let L be the language containing singleton c and every pomset $(a \parallel bu)$ where $u \in L$, i.e. $L = \{c, a \parallel (bc), a \parallel (b(a \parallel (bc))), \dots\}$. This language is accepted by the PR $\mathcal{R} = (M, \odot, \oplus, e, i, F)$ where $M = \{r_a, r_b, r_c, r_{bc}, r_0, e\}$, $i(x) = r_x$ for $x \in \{a, b, c\}$, $F = \{r_c\}$, and $\oplus$ and $\odot$ are such that $r_b \odot r_c = r_{bc}$, $r_a \oplus r_{bc} = r_c$, e is the neutral element for both operations, and all the other possible products return r_0.*

Definition 8 (Equivalence) *Two PRs $\mathcal{R}_1$ and $\mathcal{R}_2$ on a common alphabet Σ are* equivalent *if $\mathcal{L}(\mathcal{R}_1) = \mathcal{L}(\mathcal{R}_2)$.*

We define the set of *evaluation trees* $\mathsf{ET}_{\mathcal{R}}(w)$ of a pomset w in a PR $\mathcal{R} = (M, \odot, \oplus, e, i, F)$ by relabelling the nodes of w's terms in $\mathsf{ST}(w)$ with states of $\mathcal{R}$ inductively:

- If node ι is labelled with $x \in \Sigma \cup \{\varepsilon\}$ in $\mathsf{ST}(w)$, then we relabel it with $i^\sharp_{\mathcal{R}}(x)$ instead.
- If node ι is labelled with $\circ \in \{\cdot, \|\}$ and its left (resp. right) subtree represents pomset w_1 (resp. w_2) and $m_1 = i^\sharp(w_1)$ (resp. $m_2 = i^\sharp(w_2)$), then we relabel it with state $i^\sharp_{\mathcal{R}}(m_1 \circ_{\mathcal{R}} m_2)$ instead for the appropriate $\circ_{\mathcal{R}} \in \{\odot, \oplus\}$.

Figure 4 displays an evaluation tree expliciting the computation performed by pomset recognizer $\mathcal{R}$ of Example 7 on pomset $bc \parallel a$.

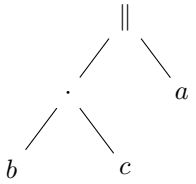

Figure 3: A canonical term of $bc \parallel a$.

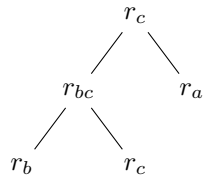

Figure 4: An evaluation tree of $bc \parallel a$.

2.3 Contexts

Definition 9 (Multi-contexts) *For $m \in \mathbb{N}^*$, let $\Xi = \{\Box_1, \dots, \Box_m\}$ be a set of m distinct letters such that $\Xi \cap \Sigma = \emptyset$. The set of m-*contexts $\mathrm{C}_m(\Sigma)$ *is the subset of $\mathrm{SP}(\Sigma \cup \Xi)$ of pomsets containing exactly one element labelled by $\Box_j$ for all $j \in \{1, \dots, m\}$.*

Given $c \in \mathrm{C}_m(\Sigma)$ and $w_1, \dots, w_m \in \mathrm{SP}(\Sigma)$, we denote by $c[w_1, \dots, w_m]$ the pomset where $\Box_j$ has been replaced by w_j. Intuitively, a m-context is a pomset pattern featuring $\Box_j$ placeholder symbols that can be replaced by pomsets. We write $\mathrm{SP}(\Sigma) = \mathrm{C}_0(\Sigma)$.

We simply call 1-contexts *contexts*, and always denote their placeholder symbol $\Box$. Pomset contexts are called $\Box$-terms in [14, 15, 16, 17], but the symbol ξ is used instead of $\Box$. Given $c_1, c_2 \in \mathrm{C}_1(\Sigma)$, $c_1[c_2] \in \mathrm{C}_1(\Sigma)$ stands for the context obtained by replacing $\Box$ with c_2 in c_1. c_2 is then said to be a *subcontext* of $c_1[c_2]$. For $w \in \mathrm{SP}(\Sigma)$, a *split* of w is a pair $(c, z) \in \mathrm{C}_1(\Sigma) \times \mathrm{SP}(\Sigma)$ such that $w = c[z]$. Note that z is a subpomset of w. Finally, given $C \subseteq \mathrm{C}_1(\Sigma)$ and $A \subseteq \mathrm{SP}(\Sigma) \cup \mathrm{C}_1(\Sigma)$, we define the set $C[A] = \{c[z] \mid c \in C, z \in A\}$.

Given a pomset recognizer $\mathcal{R} = (M, \odot, \oplus, e, i, F)$, note that $\mathcal{R}(w_1) = \mathcal{R}(w_2)$ does not imply that for all $c \in \mathrm{C}_1(\Sigma)$, $\mathcal{R}(c[w_1]) = \mathcal{R}(c[w_2])$. Indeed, it might be that $i^\sharp(w_1), i^\sharp(w_2) \in F$ but $i^\sharp(w_1) \neq i^\sharp(w_2)$: w_1 and w_2 lead to different accepting states, thus potentially yielding a different result whenever inserted in c then evaluated in $\mathcal{R}$. As a consequence, $i^\sharp(c[w_1])$ may differ from $i^\sharp(c[w_2])$. However, the following result holds:

Lemma 10 (Freeness of PRs [13, Lem. 29]) *For all $w_1, w_2 \in \mathrm{SP}(\Sigma)$, if $i^\sharp(w_1) = i^\sharp(w_2)$, then for all $c \in \mathrm{C}_1(\Sigma)$, $i^\sharp(c[w_1]) = i^\sharp(c[w_2])$.*

2.4 A Myhill-Nerode theorem

Let $L \subseteq \mathrm{SP}(\Sigma)$ and $u, v \in \mathrm{SP}(\Sigma)$. $u \sim_L v$ if for all $c \in \mathrm{C}_1(\Sigma)$, $c[u] \in L \iff c[v] \in L$. The relation $\sim_L$ is an equivalence relation; we say that it is a *congruence* relation on $\mathrm{SP}(\Sigma)$ if it is preserved by $\cdot$ and $\|$. $[w]_{\sim_L}$ stands for the equivalence class of w in the quotient space $\mathrm{SP}(\Sigma)/\sim_L$ of $\mathrm{SP}(\Sigma)$ w.r.t. $\sim_L$. It induces a syntactic homomorphism $\mathrm{SP}(\Sigma) \to \mathrm{SP}(\Sigma)_{/\sim_L}$ and there exists a Myhill-Nerode characterization of recognizable languages of $\mathrm{SP}(\Sigma)$:

Theorem 2 (Characterizing recognizable languages [16]) *L is recognizable if and only if $\sim_L$ is a congruence relation of finite index.*

Let $w_1, w_2 \in \mathrm{SP}(\Sigma)$. Given a pomset language L, we say that $c \in \mathrm{C}_1(\Sigma)$ is a *distinguishing context* in L for w_1 and w_2 if $c[w_1] \in L \iff c[w_2] \notin L$, that is, one of $c[w_1]$ and $c[w_2]$ is in L while the other is not. If we assume L is recognized by a pomset recognizer $\mathcal{R}$, this necessarily implies that $m_1 = i^\sharp(c[w_1]) \neq m_2 = i^\sharp(c[w_2])$: one state must be in F while the other is not. We then say that c *distinguishes* the states m_1 and m_2. If there is no such c, we say that m_1 (resp. w_1) and m_2 (resp. w_2) are *indistinguishable*.

Definition 11 (Reachable and minimal pomset recognizers) *A pomset recognizer $\mathcal{R} = (M, \odot, \oplus, 1, i, F)$ is said to be* reachable *if, for all $m \in M$, there exists $w \in \mathrm{SP}(\Sigma)$ such that $i^\sharp(w) = m$; w is said to be an* access sequence *of m.*

Moreover, it is minimal *if it is reachable and for all $w_1, w_2 \in \mathrm{SP}(\Sigma)$ such that $i^\sharp(w_1) \neq i^\sharp(w_2)$, there exists $c \in \mathrm{C}_1(\Sigma)$ such that $\mathcal{R}(c[w_1]) \neq \mathcal{R}(c[w_2])$.*

Intuitively, $\mathcal{R}$ is minimal if any pair of states in M can always be distinguished by some context. If L is recognizable, $\sim_L$ induces an obvious minimal recognizer $\mathcal{R}_L = (\mathrm{SP}(\Sigma)_{/\sim_L}, \cdot, \| , [\varepsilon]_{\sim_L}, i_L, F_L)$ such that $\forall a \in \Sigma$, $i_L(a) = [a]_{\sim_L}$ and $F_L = \{[w]_{\sim_L} \in \mathrm{SP}(\Sigma)/\sim_L \mid w \in L\}$.

3 Implementation Details

3.1 Pomsets as AST

Abstract syntax trees (AST) are data-structures that store a synthaxic expression directly in revser polish notation (RPN). They are able to store a category of n elements in $2n - 1$ nodes [18]. We can use said structure to represent Pomsets. In such implementation, internal nodes

represent operations in $\{\bullet, ||\}$ and leaves represent the elements of the pomset (see figure 5). The RPN notation of the pomset can thus be retrieved with a post-order traversal.

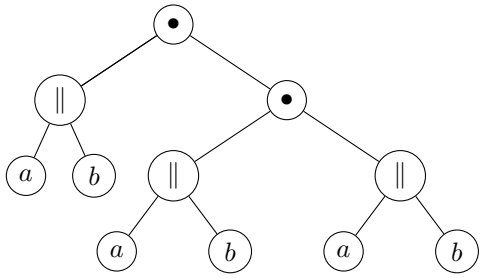

Figure 5: Abstract syntax tree modelling the pomset $p = (a||b) \bullet (a||b) \bullet (a||b)$.

Computing concatenation (resp. parallel product) on AST representing pomsets is trivial. Indeed, for two AST g_1 and g_2 it suffices to create a new AST g_3 with its root being a node with $e = \bullet$ (resp. $||$) and its children being g_1 and g_2 (see algorithm 1). Therefore computing both operations in constant time regardless of the size of the operands.
In a similar manner, substitution $g_1 \ o_\xi \ g_2$ can be done by replacing the leaf.ves holding the letter ξ in g_1 by g_2, resolving in $O(n)$ in the worst case.

Algorithm 1 PomsetComposition($p_1, p_2, \circ$)

1: **if** p_1 is empty **and** p_2 is empty **then**
2: **return** a new empty Pomset
3: **if** p_1 is empty **then**
4: **return** p_2
5: **if** p_2 is empty **then**
6: **return** p_1
7: $res \leftarrow$ a new BinaryTree with $\circ$ as the value
8: $res.left \leftarrow p_1.tree$
9: $res.right \leftarrow p_2.tree$
10: $res \leftarrow$ stack_right$(res, \circ)$
11: **return** res

In order to always have two equivalent pomsets be the exact same we can canonize their AST by recursively stacking all sequences of operations to the right using right bintree rotation (see line 10 of algorithm 1). This allows us to compare two pomsets by literally comparing their AST.

The AST form of a Pomset can be constructed from its infix notation by using the Shunting-Yard algorithm [19].

3.2 Pomset Recognizers

Pomset Recognizers can be represented as bottom-up deterministic finite tree automata using two Cayley tables to store the behavior of the transition functions $\odot$ and $\oplus$. Algorithms for all decidable problems on such machines have been described in [20].

3.3 Membership query

In order to compute the membership query of a Pomset p represented as an AST to a Pomset Recognizer $\mathcal{R}$, we need to parse the bintree using the internal functions of $\mathcal{R}$'s bimonoid. This results to a single element of the bimonoid, as described in algorithm 2. For a pomset of size n, this algorithm performs always exactly $2n - 2$ recursive calls.

Then, we can compute the membership query of a pomset by checking if the element computed by algorithm 2 is in the set of final element of the Pomset recogniser, as described in algorithm 3.

Algorithm 2 process_ast(p)

if $p == null$ **then**
  return $\mathcal{R}.unit$
if p is a leaf **then**
  return $\mathcal{R}.i(p.data)$
else
  $l = process\_ast(p.left\_child)$
  $r = process\_ast(p.right\_child)$
  if $data\ is$ • **then**
    return $\mathcal{R}.\odot(l, r)$
  else
    return $\mathcal{R}.\oplus(l, r)$

Algorithm 3 recognise(p)

if $\mathcal{R}$.F is empty **then**
  return $false$
$res = process\_ast(p)$
for state in $\mathcal{R}$.F **do**
  if $res = state$ **then**
    return true
return false

4 Active Learning

4.1 The active learning framework

Consider a recognizable pomset language L on an alphabet Σ. Let $\mathcal{M}$ be a minimal PR called the *model* such that $\mathcal{L}(\mathcal{M}) = L$. *Active learning* is a cooperative game between a learner and a *minimally adequate teacher* (MAT). It consists for the learner in computing a minimal pomset recognizer for L by asking two types of queries on L to the MAT:

Membership queries. Given $w \in \mathrm{SP}(\Sigma)$, does $w \in L$, i.e. what is $\mathcal{M}(w)$?

Equivalence queries Given a pomset recognizer $\mathcal{H}$ (called the hypothesis) on Σ, does $\mathcal{L}(\mathcal{H}) = L = \mathcal{L}(\mathcal{M})$? If it does not, return a *counter-example* $w \in \mathrm{SP}(\Sigma)$ such that $\mathcal{H}(w) \neq \mathcal{M}(w)$.

The ability to infer the model $\mathcal{M}$ stems from Theorem 2. Active learning algorithms compute an *under-approximation* $\sim_{\mathcal{H}}$ of $\sim_L = \sim_{\mathcal{M}}$ such that $w_1 \not\sim_{\mathcal{H}} w_2 \implies w_1 \not\sim_L w_2$. To do so, they maintain a finite set S of pomsets and a finite set $\mathcal{C}$ of contexts. Each pair of elements of S is distinguished by at least one element of $\mathcal{C}$, thus bearing witness to the existence of at least $|S|$ equivalence classes of $\sim_L$, the set S being their representatives.

Obviously, $w_1 \sim_{\mathcal{H}} w_2 \implies w_1 \sim_L w_2$ may not hold if the hypothesis is too coarse, thus, $\mathcal{H}$ may have to be refined several times. Nevertheless, each refinement increases the number of equivalences classes of $\sim_L$ distinguished by $\sim_{\mathcal{H}}$, until the classes of $\sim_{\mathcal{H}}$ are exactly the classes of $\sim_L$, at which point $\sim_{\mathcal{H}} = \sim_{\mathcal{M}} = \sim_L$ and $\mathcal{L}(\mathcal{H}) = \mathcal{L}(\mathcal{M}) = L$.

4.2 Common structures and invariants

Data structures.

We maintain a finite set S of pomsets called the set of *representatives* or *access sequences*, meant to store the representatives of $\sim_L$'s equivalence classes. By design, S will be closed by the subpomset operation and contain the empty pomset ε. We also introduce the *frontier* set $S^+ = (\Sigma \cup \{u \circ v \mid \circ \in \{\cdot, \|\}, u, v \in S\}) \setminus S$ that contains combinations of elements of S and single letters: its purpose is to infer the internal operations $\cdot_{\mathcal{M}}$ and $\|_{\mathcal{M}}$ of the model.

A *pack of components* $\mathcal{B} = \{B_1, \ldots, B_m\}$ partitions $S \cup S^+$ in such a manner each component contains at least one $s \in S$. For $s \in S \cup S^+$, $\mathcal{B}_s$ stands for the only component of $\mathcal{B}$ s belongs to. Given $B \in \mathcal{B}$, $\alpha_{\mathcal{B}}(B) = S \cap B$ is called the set of *access sequences* of B. For $s \in S \cup S^+$, we define $\alpha_{\mathcal{B}}(s) = \alpha_{\mathcal{B}}(\mathcal{B}_s)$. $\mathcal{B}$ under-approximates the classes of $\sim_{\mathcal{M}}$.

Finally, we maintain a *discrimination tree* $\mathcal{D}$: it is a full binary tree, its inner nodes being labelled by contexts in $\mathrm{C}_1(\Sigma)$, and its leaves, either unlabelled or labelled by a component of $\mathcal{B}$ in such a fashion $\mathcal{D}$'s set of labelled leaves is in bijection with $\mathcal{B}$. In particular, $\mathcal{D}$'s root is labelled by $\Box$. The labels of $\mathcal{D}$'s inner nodes form a set of contexts $\mathcal{C}$. Given $B \in \mathcal{B}$, $\mathcal{C}_B$ is defined as the set of contexts that appear along the branch that runs from the root of $\mathcal{D}$ to the leaf labelled by B. In particular, note that for all $B \in \mathcal{B}$, $\Box \in \mathcal{C}_B$. $\mathcal{D}$'s use is to posit which class of $\mathcal{B}$ a pomset belongs to.

Operations and invariants.

For any pomset $w \in \mathrm{SP}(\Sigma)$, we define the *sifting* operation of w through $\mathcal{D}$: starting at the root of $\mathcal{D}$, at every node labelled by a context c of $\mathcal{D}$ we branch to the right (resp. left) child if $c[w] \in L$ (resp. $c[w] \notin L$). We iterate this procedure until a leaf is reached: the matching component $B \in \mathcal{B}$ is the result of the sifting operation. We define $\mathcal{D}(w) = B$. Note that $\mathcal{D}(w)$ may be undefined if w is sifted into an unlabelled leaf. Thus, $\mathcal{D}$ can be viewed as a partial function $\mathrm{SP}(\Sigma) \to \mathcal{B}$. Sifting requires a number of membership queries bounded by the height of $\mathcal{D}$. Intuitively, the discrimination tree is used to classify pomsets: pomsets that behave similarly w.r.t. the finite set of distinguishing contexts $\mathcal{C}_B$ are lumped into the same component $B \in \mathcal{B}$.

Property 1 *By design, the learning algorithm maintains the following invariants:*

1. *For any $s \in S \cup S^+$, $\mathcal{D}(s) = \mathcal{B}_s$.*
2. *For any $B \in \mathcal{B}$, $s_1, s_2 \in B$, $c \in C_B$, $\mathcal{M}(c[s_1]) = \mathcal{M}(c[s_2])$.*
3. *Let $B_1, B_2 \in \mathcal{B}$ such that $B_1 \neq B_2$; then for any $s_1 \in B_1$, $s_2 \in B_2$, there exists $c \in \mathcal{C}_{B_1} \cap \mathcal{C}_{B_2}$ such that $\mathcal{M}(c[s_1]) \neq \mathcal{M}(c[s_2])$; c labels B_1 and B_2's deepest common ancestor in $\mathcal{D}$.*

4. *Let $\sim_{\mathcal{B}}$ be the equivalence relation on $S \cup S^+$ inferred from the partition $\mathcal{B}$. Then it is an under-approximation of $\sim_L$ on $S \cup S^+$: $\forall s_1, s_2 \in S \cup S^+$, $s_1 \not\sim_{\mathcal{B}} s_2 \implies s_1 \not\sim_L s_2$. As a consequence, $|\mathcal{B}| \leq |\mathrm{SP}(\Sigma)/\sim_L|$.*

For $w \in \mathrm{SP}(\Sigma)$, if $\mathcal{D}(w)$ is defined, we write $\mathcal{B}_w = \mathcal{D}(w)$. Thanks to Invariant 1, this notation doesn't invalidate the previous notation $\mathcal{B}_s$ for $s \in S \cup S^+$.

4.3 Building the hypothesis

Properties of the partition.

$\mathcal{B}$ is said to be *consistent* if for any $B_1, B_2 \in \mathcal{B}$, $u_1, v_1 \in \alpha_{\mathcal{B}}(B_1)$, $u_2, v_2 \in \alpha_{\mathcal{B}}(B_2)$, and $\circ \in \{\cdot, \|\}$, $u_1 \circ u_2$ and $v_1 \circ v_2$ belong to the same component of $\mathcal{B}$. Intuitively, no matter the representatives of B_1 and B_2 we consider, their composition will belong to the same component. Moreover, $\mathcal{B}$ is $\circ$-*associative* for $\circ \in \{\cdot, \|\}$ if for any $s_1, s_2, s_3 \in S$ and $s_l \in \alpha_{\mathcal{B}}(s_1 \circ s_2)$, $s_r \in \alpha_{\mathcal{B}}(s_2 \circ s_3)$, $\mathcal{B}_{s_l \circ s_3} = \mathcal{B}_{s_1 \circ s_r}$. Finally, $\mathcal{B}$ is said to be *sharp* if for any $B \in \mathcal{B}$, $|S \cap B| = 1$.

Thus, we can extend the operators $\cdot$ and $\|$ to components of $\mathcal{B}$, and the resulting laws will be internal and associative. Finally, for any $B \in \mathcal{B}$, since $\Box \in \mathcal{C}_B$ and for any $u \in B$, $\Box[u] = u$, $\mathcal{M}$ is constant on B: this shared value is written $\mathcal{M}(B)$.

Defining the hypothesis.

If $\mathcal{B}$ is *consistent*, $\cdot$-*associative*, and $\|$-*associative*, then we design the hypothesis $\mathcal{H} = (H, \cdot_{\mathcal{H}}, \|_{\mathcal{H}}, e_{\mathcal{H}}, i_{\mathcal{H}}, F_{\mathcal{H}})$ as follows:

- $H = \mathcal{B}$. $\mathcal{H}$'s states are the postulated equivalence classes of $\sim_L$.
- Given $u, v \in S$, since $\mathcal{B}$ is *consistent*, we can define $\mathcal{B}_u \cdot_{\mathcal{H}} \mathcal{B}_v = \mathcal{B}_{u \cdot v}$ (resp. $\mathcal{B}_u \|_{\mathcal{H}} \mathcal{B}_v = \mathcal{B}_{u \| v}$). We use S^+ and $\mathcal{B}$ to build $\mathcal{H}$'s internal operations.
- $e_{\mathcal{H}} = \mathcal{B}_\varepsilon$. The neutral element is the class of the empty pomset.
- Given $a \in \Sigma$, $i_{\mathcal{H}}(a) = \mathcal{B}_a$. We rely on $\Sigma \subseteq S \cup S^+$ to build a pomset homomorphism.
- $F_{\mathcal{H}} = \{B \in \mathcal{B} \mid \mathcal{M}(B) = 1\}$. A component is accepting if its members are accepted by $\mathcal{M}$. $F_{\mathcal{H}}$ corresponds to the leaves of $\mathcal{D}$ belonging to its right subtree.

For $w \in \mathrm{SP}(\Sigma)$, we define the component $\mathcal{B}_w = i^\sharp_{\mathcal{H}}(w)$ w evaluates to in $\mathcal{H}$ and its set of access sequences $\alpha_{\mathcal{H}}(w) = \alpha_{\mathcal{B}}(\mathcal{B}_w)$. As proven later in Lemma 15, this notation is compatible with the earlier definition of $\mathcal{B}_w$ for $w \in S \cup S^+$. By design of $\mathcal{H}$, freeness of pomset recognizers, and consistency of $\mathcal{B}$, the hypothesis handles pomsets and their access sequences similarly:

Property 2 (Substitution by access sequences) $\forall c \in \mathrm{C}_1(\Sigma), \forall w \in \mathrm{SP}(\Sigma), \forall p \in \alpha_{\mathcal{H}}(w), \mathcal{H}(c[w]) = \mathcal{H}(c[p])$ *and* $i^\sharp_{\mathcal{H}}(c[w]) = i^\sharp_{\mathcal{H}}(c[p])$.

Compatibility of the hypothesis.

Given a set $X \subseteq \mathrm{SP}(\Sigma)$ of pomsets, hypothesis $\mathcal{H}$ is X-*compatible* if for any $w \in X$, $\mathcal{H}(w) = \mathcal{M}(w)$. $\mathcal{H}$ is said to be *compatible* with $\mathcal{B}$ if it is compatible with $\bigcup_{B \in \mathcal{B}} \{c[s] \mid s \in B, c \in \mathcal{C}_B\}$.

Active learning algorithms such as TTT [6], $L^\#$ [7], or van Heerdt et al.'s adaptation of L^* [13] to pomset recognizers may not always immediately result in a compatible hypothesis. However,

incompatibilities provide a 'free' counter-example $c[s]$ such that $\mathcal{H}(c[s]) \neq \mathcal{M}(c[s])$ without requiring an extra membership or equivalence query. We should therefore guarantee that $\mathcal{H}$ is compatible with $\mathcal{B}$ before submitting an equivalence query.

4.4 Handling counter-examples

The Rivest-Schapire counter-example handling method on finite words consists in studying all the possible splits $w = u \cdot a \cdot v$, $a \in \Sigma$, of a counter-example $w \in \Sigma^+$, then trying to find one such that the hypothesis and the model agree on the input $p' \cdot v$ where $p' \in \alpha_{\mathcal{H}}(u \cdot a)$ but disagree on $p \cdot a \cdot v$ where $p \in \alpha_{\mathcal{H}}(u)$, thus proving that the successor of state $\mathcal{B}_p$ in the hypothesis has been incorrectly identified as $\mathcal{B}_{p'}$, suffix v being witness to this error.

Intuitively, replacing a prefix u of w by its access sequence p is akin to feeding u to the hypothesis, then letting either the model or the hypothesis handle the rest of the computation, iterating on all possible splits until the algorithm witnesses the model and hypothesis no longer being in agreement. This change of behaviour, called a *breaking point*, yields a distinguishing suffix and a further refinement of the partition $\mathcal{B}$ and its matching hypothesis.

Extending breaking points to pomsets is non-trivial due to the branching nature of terms. Assume that $\mathcal{B}$ is a consistent, associative partition from which a hypothesis $\mathcal{H}$ is inferred. We define breaking points w.r.t. canonical decomposition:

Definition 12 (Agreement) *Given $c \in \mathrm{C}_1(\Sigma)$ and $z \in \mathrm{SP}(\Sigma)$, we define the* agreement *predicate $\mathcal{A}(c, z) =$ "$\forall p \in \alpha_{\mathcal{H}}(z), \mathcal{H}(c[p]) = \mathcal{M}(c[p])$".*

Definition 13 (Breaking point) *Given a counter-example $w \in \mathrm{SP}(\Sigma)^+$ such that $\mathcal{H}(w) \neq \mathcal{M}(w)$, and a split (c, z) of w such that $\mathcal{A}(c, z) = 1$, a (left)* breaking point *is either:*

- *the pair (c, z) if $z \in \Sigma$;*
- *a quadruplet $(c, \circ, z_1, z_2)$ where $\circ \in \{\cdot, \|\}$, $z_1, z_2 \in \mathrm{SP}^+(\Sigma)$, $z = z_1 \circ z_2$ is a canonical decomposition of z, and $\mathcal{A}(c[\Box \circ z_2], z_1) = 0$.*

We seek a split (c, z) of w such that there exists $p \in S^+ \cap \mathcal{B}_z$, for any $p' \in \alpha_{\mathcal{H}}(z)$, $\mathcal{M}(c[p']) \neq \mathcal{M}(c[p])$ as it guarantees that p belongs to another class of $\sim_L$ than the current elements of S, resulting in a refinement of $\mathcal{B}_z$. Unlike the Rivest-Schapire decomposition, we may however not directly be able to infer a distinguishing context from every breaking point. Indeed, given a breaking point $(c, \circ, z_1, z_2)$, while c distinguishes $p_1 \circ z_2$ for some $p_1 \in \alpha_{\mathcal{H}}(z_1)$ from any $p' \in \alpha_{\mathcal{H}}(z)$, $p_1 \circ z_2$ may not belong to S^+. Nevertheless, Algorithm 8 can alter an original counter-example w until a breaking point can be used to infer a new class and a refinement. We call such a breaking point *effective*.

5 Adapting the L^λ Algorithm

We detail here the various components of the L^λ active learning algorithm, some of them being somewhat data agnostic, others being peculiar to pomset languages.

5.1 Expanding components

Algorithm 4 inserts a new pomset w belonging to the frontier S^+ or equal to ε into the set S of access sequences then updates S^+ by exploring w's successors (i.e. the pomsets that we can

build by combining w with another element of S) and using $\mathcal{D}$ to sift them into the existing partition $\mathcal{B}$.

Algorithm 4 `Expand`(w) where $w \in S^+$ or $w = \varepsilon$ if $\mathcal{B} = \emptyset$

1: $S \leftarrow S \cup \{w\}$
2: **for** $p \in \{w\} \cup \{p' \circ w, w \circ p' \mid \circ \in \{\cdot, \|\}, p' \in S\} \cup \Sigma$ **do**
3: **if** p does not belong to any class of $\mathcal{B}$ **then**
4: $B \leftarrow \mathcal{D}(p)$
5: **if** B is defined **then**
6: $B \leftarrow B \cup \{p\}$
7: **else**
8: $B_p \leftarrow \{p\}$
9: $\mathcal{B} \leftarrow \mathcal{B} \cup \{B_p\}$
10: `UpdateTreeLeaf`$(\mathcal{D}, p, B_p)$
11: `Expand`(p)

By design, building a PR requires sorting the letters of Σ into $\mathcal{B}$, hence Line 2, despite elements of Σ not being successors of w. However, we only insert them once, during the very first call. Expanding a pomset may result in a new class being created if the leaf p was sifted into is unlabelled. Indeed, $\mathcal{D}$ initially consists of a root labelled by the identity context $\Box$ and two children that have yet to be labelled by components of $\mathcal{B}$ due to $S \cup S^+$ being empty. Either leaf may even end up not being labelled at all if the PR is trivial (i.e. has language $\mathrm{SP}(\Sigma)$ or $\emptyset$). Thus, if a pomset p is sifted into an unlabelled leaf, Lines 7 to 11 result in a new class B_p being created and $\mathcal{D}$ being updated by labelling said leaf with B_p.

5.2 Refining components

Algorithm 5 refines a component B into two new components B_0 and B_1, assuming a context c distinguishes two access sequences of B. S, $\mathcal{B}$ and $\mathcal{D}$ are updated accordingly. Lines 5 and 6 guarantee that the new components have at least one access sequence in S. Line 4 consists in replacing leaf B of the discrimination tree $\mathcal{D}$ with an inner node labelled by c whose left (resp. right) child is a new leaf labelled by B_0 (resp. B_1).

Algorithm 5 `Refine`(B, c) where $B \in \mathcal{B}$, $c \in \mathrm{C}_1(\Sigma)$, and $\exists z_1, z_2 \in B, \mathcal{M}(c[z_1]) \neq \mathcal{M}(c[z_2])$

1: $B_0 \leftarrow \{w \in B \mid \mathcal{M}(c[w]) = 0\}$
2: $B_1 \leftarrow \{w \in B \mid \mathcal{M}(c[w]) = 1\}$
3: $\mathcal{B} \leftarrow (\mathcal{B} \setminus \{B\}) \cup \{B_0, B_1\}$
4: `RefineTree`$(\mathcal{D}, B, c, B_0, B_1)$
5: **if** $S \cap B_0 = \emptyset$ **then** `Expand`(p_0) for some $p_0 \in B_0$
6: **if** $S \cap B_1 = \emptyset$ **then** `Expand`(p_1) for some $p_1 \in B_1$

Algorithm 6 refines partition $\mathcal{B}$ whenever it encounters a consistency issue, e.g. class B contains two representatives p_1 and p_2 such that $p_1 \circ p$ and $p_2 \circ p$ in $S \cup S^+$ do not belong to the same class. This inconsistency yields a context $c[\Box \circ p]$ that distinguishes p_1 and p_2, where $c \in \mathcal{C}$ is the label of the deepest common ancestor in $\mathcal{D}$ of $p_1 \circ p$ and $p_2 \circ p$. This algorithm returns Boolean $\top$ if and only if $\mathcal{B}$ was already consistent in the first place. It could also be that $p \circ p_1$ and $p \circ p_2$

do not belong to the same class, resulting in a distinguishing context $c[p \circ \Box]$: we omit this case here for brevity's sake.

Algorithm 6 `MakeConsistent()`

1: `already_consistent` $\leftarrow \top$
2: **while** $\exists \circ \in \{\cdot, \|\}, \exists B \in \mathcal{B}, \exists p_1, p_2 \in \alpha_{\mathcal{B}}(B), \exists p \in S, \mathcal{D}(p_1 \circ p) \neq \mathcal{D}(p_2 \circ p)$ **do**
3: Let $c \in \mathcal{C}$ be such that $\mathcal{M}(c[p_1 \circ p]) \neq \mathcal{M}(c[p_2 \circ p])$.
4: `Refine`$(B, c[\Box \circ p])$
5: `already_consistent` $\leftarrow \bot$
6: **return** `already_consistent`

Algorithm 7 refines partition $\mathcal{B}$ whenever it encounters an associativity issue: if $(s_1 \circ s_2) \circ s_3$ and $s_l \circ s_3$ do not behave similarly, as witnessed by the deepest common ancestor $c \in \mathcal{C}$ in $\mathcal{D}$ of $s_1 \circ s_r$ and $s_l \circ s_3$, since $s_1 \circ s_2 \in S^+$ and $s_l \in S$ both belong to a same class B, the context $c[\Box \circ s_3]$ refines B. A similar test is performed to detect right associativity issues. This algorithm returns Boolean $\top$ if and only if $\mathcal{B}$ was already associative in the first place.

Algorithm 7 `MakeAssoc()`

1: `already_assoc` $\leftarrow \top$
2: **while** $\exists \circ \in \{\cdot, \|\}, \exists s_1, s_2, s_3 \in S, \exists s_l \in \alpha_{\mathcal{B}}(s_1 \circ s_2), \exists s_r \in \alpha_{\mathcal{B}}(s_2 \circ s_3), \mathcal{D}(s_1 \circ s_r) \neq \mathcal{D}(s_l \circ s_3)$ **do**
3: Let $c \in \mathcal{C}$ be such that $\mathcal{M}(c[s_1 \circ s_r]) \neq \mathcal{M}(c[s_l \circ s_3])$.
4: Let $p \in \alpha_{\mathcal{B}}(s_1 \circ s_2 \circ s_3)$.
5: `query` $\leftarrow \mathcal{M}(c[p])$
6: **if** $\mathcal{M}(c[s_l \circ s_3]) \neq$ `query` **then**
7: `Refine`$(\mathcal{B}_{s_1 \circ s_2}, c[\Box \circ s_3])$
8: **else**
9: `Refine`$(\mathcal{B}_{s_2 \circ s_3}, c[s_1 \circ \Box])$
10: `already_assoc` $\leftarrow \bot$
11: **return** `already_assoc`

5.3 Using counter-examples to identify new components

Algorithm 8 is an important contribution as it differs from existing counter-example handling algorithms on finite words and pomsets. Its arguments are a context c and a pomset z such that $w = c[z]$ is a counter-example. It returns a context c' and a pomset p belonging to the frontier such that c' distinguishes p from all the existing access sequences of $\mathcal{B}_p$.

- Line 1 handles the base case: if z is a letter, we can trivially infer a distinguishing context and a new representative. Property 18 guarantees that, by the time the algorithm reaches a leaf, it is indeed a breaking point.
- Lines 3 to 4 consist in inductively finding a breaking point along the leftmost branch of a canonical term t of w. By Property 18, such a breaking point always exists. Figure 6 displays how the algorithm explores $t \in \mathsf{ST}(c[z])$: if ι is the insertion point of z in t, let μ and ν be the children of ι. The exponent of each node stands for the local value of predicate $\mathcal{A}(c^x, z^x)$ for $x \in \{\iota, \mu, \nu\}$, where (c^x, z^x) stands for the split of w induced by node x. Here,

Algorithm 8 `FindEBP`(c, z) where $c \in \mathrm{C}_1(\Sigma)$, $z \in \mathrm{SP}^+(\Sigma)$, $\mathcal{H}(c[z]) \neq \mathcal{M}(c[z])$ and $\mathcal{A}(c, z) = 1$

1: **if** $z \in \Sigma$ **then return** (c, z)
2: **else if** $z = z_1 \circ z_2$ is a canonical decomposition of z **then**
3: **if** $\mathcal{A}(c[\Box \circ z_2], z_1)$ **then**
4: **return** `FindEBP`$(c[\Box \circ z_2], z_1)$
5: **else**
6: Let $p_1 \in \alpha_{\mathcal{H}}(z_1)$ be such that $\mathcal{H}(c[p_1 \circ z_2]) \neq \mathcal{M}(c[p_1 \circ z_2])$.
7: **if** $\mathcal{A}(c[p_1 \circ \Box], z_2)$ **then**
8: **return** `FindEBP`$(c[p_1 \circ \Box], z_2)$
9: **else**
10: Let $p_2 \in \alpha_{\mathcal{H}}(z_2)$ be such that $\mathcal{H}(c[p_1 \circ p_2]) \neq \mathcal{M}(c[p_1 \circ p_2])$.
11: **return** $(c, p_1 \circ p_2)$

$(c, \circ, z_1, z_2)$ is a breaking point. If it is not the case, the algorithm inductively explores the (purple) leftmost sub-branch rooted in μ instead.

- Lines 6 to 7 determine whether a refinement can be inferred from this breaking point. Our intuition is that witnessing a conflict when the left branch is replaced by an access sequence p_1 is not enough; we need to check if feeding both branches to the hypothesis still result in a conflict.
  - If it does (Lines 10 to 11), then c distinguishes $p_1 \circ p_2$ for some $p_1 \in \alpha_{\mathcal{H}}(z_1)$, $p_2 \in \alpha_{\mathcal{H}}(z_2)$ from any access sequence in $\alpha_{\mathcal{H}}(p_1 \circ p_2)$. Then the algorithm returns c and $p_1 \circ p_2$.
  - Otherwise, the algorithm no longer explores the leftmost branch rooted in ι. Line 8 instead replaces z_1 with an access sequence p_1 and restarts the exploration process from ν, as shown by Figure 7. The pre-condition is respected, as $c[p_1 \circ z_2]$ is still a counter-example due to $\mathcal{A}(c^\mu_w, z^\mu_w) = 0$ by definition of breaking points.

By Theorem 5, Algorithm 8 ends and does return a context c' that distinguishes a representative p of a new component from the access sequences of its previous component. Due to each inductive call descending deeper into the term, we can intuit that Algorithm 8 performs at most $\mathcal{O}(\delta(z))$ inductive calls.

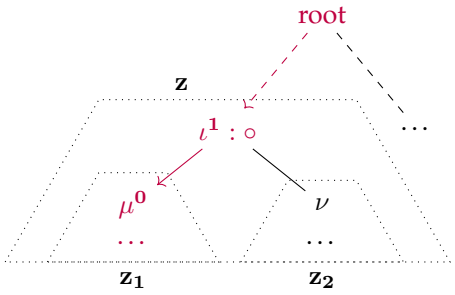

Figure 6: A breaking point along a branch of a counter-example $c[z]$.

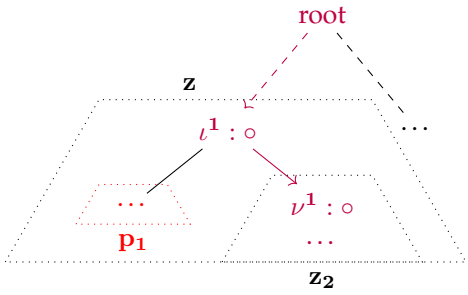

Figure 7: Replacing the left branch by its access sequence and switching to its sibling.

5.4 Inducing a refinement

Algorithm 9 characterizes L^λ: it relies on Algorithm 8 to find a representative of a new component p and a matching distinguishing context c, but does **not** use c to immediately refine $\mathcal{B}_p$. Indeed, c is of arbitrary size, being inferred from an arbitrarily long counter-example; an overly large context in $\mathcal{C}$ would weight down future membership queries.

Instead, it merely expands p (Line 5) and relies only on Algorithms 6 and 7 to refine $\mathcal{B}$ and $\mathcal{H}$ (Lines 6 and 7), therefore keeping $\mathcal{C}$ closed by Property 17. Fixing a consistency defect may result in an associativity defect appearing and vice versa, hence the loop.

Should a lack of associativity or consistency defects prevents a new component for p from being refined, Algorithm 9 also adds $c[p]$ and $c[p']$ to a counter-example pool $\mathcal{E}$ it maintains (Line 4) for any representative $p' \in \alpha_{\mathcal{H}}(p)$ in order to guarantee that p and p' are eventually distinguished. By Theorem 6, this loop eventually depletes $\mathcal{E}$ and ends.

Algorithm 9 `HandleCE`(w) where $w \in \mathrm{SP}^+(\Sigma)$ is such that $\mathcal{H}(w) \neq \mathcal{M}(w)$

1: $\mathcal{E} \leftarrow \{w\}$
2: **while** $\exists u \in \mathcal{E}, \mathcal{M}(u) \neq \mathcal{H}(u)$ **do**
3: $\quad (c, p) \leftarrow$ `FindEBP`$(\square, u)$
4: $\quad \mathcal{E} \leftarrow \mathcal{E} \cup \{c[p]\} \cup \{c[p'] \mid p' \in \alpha_{\mathcal{H}}(p)\}$
5: $\quad$ `Expand`(p)
6: $\quad$ **repeat**
7: $\quad$ **until** `MakeConsistent`$() \wedge$ `MakeAssoc`$()$
8: $\quad \mathcal{H} \leftarrow$ `BuildHypothesis`$(S, \mathcal{B})$

5.5 The main loop

Algorithm 10 first initializes the pack of components $\mathcal{B}$ and the hypothesis $\mathcal{H}$ by expanding the empty pomset ε. There are no consistency and associativity defects to fix that early due to the first iteration of $\mathcal{B}$ having at most two classes.

It then submits $\mathcal{H}$ to the teacher. If the equivalence query returns a counter-example w, it then proceeds to apply Algorithm 9 to identify new components and refine $\mathcal{H}$ accordingly. Otherwise, a model $\mathcal{H}$ equivalent to $\mathcal{M}$ has been learnt and the algorithm returns $\mathcal{H}$.

Lines 6 and 7 guarantee that $\mathcal{H}$ is compatible before submitting an equivalence query. Note that this compatibility test is free (although the counter-example handling is obviously not) due to the membership query $\mathcal{M}(c[s])$ having already been performed during either the sifting of s through $\mathcal{D}$ or the refinement of $\mathcal{B}_s$.

Algorithm 10 `Learn`()

$S, \mathcal{B}, \mathcal{D} \leftarrow \emptyset, \emptyset,$ `Tree`$(\square)$
`Expand`(ε)
$\mathcal{H} \leftarrow$ `BuildHypothesis`$(S, \mathcal{B})$
while $\exists w \in \mathrm{SP}(\Sigma), \mathcal{H}(w) \neq \mathcal{M}(w)$ **do**
  `HandleCE`(w)
  while $\exists B \in \mathcal{B}, \exists s \in B, \exists c \in \mathcal{C}_B, \mathcal{H}(c[s]) \neq \mathcal{M}(c[s])$ **do**
    `HandleCE`$(c[s])$
return $\mathcal{H}$

5.6 Using counter-examples to identify new components

We assume in this section that $\mathcal{B}$ is a consistent, associative partition from which a hypothesis $\mathcal{H}$ is inferred. We first prove that a breaking point can always be found on the leftmost branch of every term of a counter-example, that is, the branch Algorithm 8 focuses on. This branch choice is arbitrary: indeed, the proof of Property 18 can be applied to any branch. This breaking point may not be effective, i.e. resulting in a new component being discovered.

Property 3 *Given a counter-example $w \in \mathrm{SP}^+(\Sigma)$ such that $\mathcal{H}(w) \neq \mathcal{M}(w)$, given a split (c, z) of w such that $\mathcal{A}(c, z) = 1$, there exists a breaking point (c', z') or $(c', \circ, z_1, z_2)$ such that c is a subcontext of c' and either $w = c'[z']$ or $w = c'[z_1 \circ z_2]$.*

The following theorem is one of our main results: it states that from a counter-example, by looking for breaking points, we can find a representative p of a new component that is distinguished by a context c' from any other representative p' of its current component.

Theorem 3 (Correction and termination of Algorithm 8) *Given $c \in \mathrm{C}_1(\Sigma)$ and $z \in \mathrm{SP}^+(\Sigma)$ such that $\mathcal{H}(c[z]) \neq \mathcal{M}(c[z])$ and $\mathcal{A}(c, z) = 1$, `FindEBP`(c, z) terminates and returns a pair $(c', p) \in \mathrm{C}_1(\Sigma) \times S^+$ such that $\forall p' \in \alpha_{\mathcal{H}}(p), \mathcal{M}(c'[p]) \neq \mathcal{M}(c'[p'])$.*

Property 4 *Given a counter-example $w \in \mathrm{SP}(\Sigma)$ such that $\mathcal{H}(w) \neq \mathcal{M}(w)$, a call to `FindEBP`$(\square, w)$ returns a pair $(c', p) \in \mathrm{C}_1(\Sigma) \times S^+$ such that for any $p' \in S$, $p \not\sim_L p'$.*

6 Termination, Correctness, and Complexity

6.1 Properties of the hypothesis

L^λ [8] on finite automata maintains a prefix-closed set of access sequences and a suffix-closed set of distinguishing suffixes. We show that similar results hold on pomsets as well.

Lemma 14 (Closedness of access sequences) *S is subpomset-closed.*

Similarly, $S \cup S^+$ is subpomset-closed. We show below that an associative, consistent, and compatible $(\mathcal{B}, S, \mathcal{D})$ induces a minimal hypothesis $\mathcal{H}$.

Lemma 15 (Reachability of the hypothesis) *An associative, consistent $(\mathcal{B}, S, \mathcal{D})$ induces a hypothesis $\mathcal{H}$ such that: 1. $i^\sharp_{\mathcal{H}}(s) = \mathcal{B}_s$ for all $s \in S \cup S^+$, 2. $\mathcal{H}$ is reachable.*

Lemma 16 (Partial compatibility) *An associative, consistent $(\mathcal{B}, S, \mathcal{D})$ induces a $(S \cup S^+)$-compatible hypothesis $\mathcal{H}$.*

Lemma 17 (Closedness of distinguishing contexts) *$\mathcal{C}$ is such that, for any $c \in \mathcal{C}$, either $c = \Box$ or there exist $c' \in \mathcal{C}$, $s \in S \setminus \{\varepsilon\}$, and $\circ \in \{\cdot, \|\}$ such that $c = c'[\Box \circ s]$ or $c = c'[s \circ \Box]$.*

Theorem 4 (Minimality) *Given a hypothesis $\mathcal{H}$ induced from an associative, consistent $(\mathcal{B}, S, \mathcal{D})$, if $\mathcal{H}$ is compatible, then it is minimal.*

6.2 Using counter-examples to identify new components

We assume in this section that $\mathcal{B}$ is a consistent, associative partition from which a hypothesis $\mathcal{H}$ is inferred. We first prove that a breaking point can always be found on the leftmost branch of every term of a counter-example, that is, the branch Algorithm 8 focuses on. This branch choice is arbitrary: indeed, the proof of lemma 18 can be applied to any branch. This breaking point may not be effective, i.e. resulting in a new component being discovered.

Lemma 18 *Given a counter-example $w \in \mathrm{SP}^+(\Sigma)$ such that $\mathcal{H}(w) \neq \mathcal{M}(w)$, given a split (c, z) of w such that $\mathcal{A}(c, z) = 1$, there exists a breaking point (c', z') or $(c', \circ, z_1, z_2)$ such that c is a subcontext of c' and either $w = c'[z']$ or $w = c'[z_1 \circ z_2]$.*

The following theorem is one of our main results: it states that from a counter-example, by looking for breaking points, we can find a representative p of a new component that is distinguished by a context c' from any other representative p' of its current component.

Theorem 5 (Correction and termination of Algorithm 8) *Given $c \in \mathrm{C}_1(\Sigma)$ and $z \in \mathrm{SP}^+(\Sigma)$ such that $\mathcal{H}(c[z]) \neq \mathcal{M}(c[z])$ and $\mathcal{A}(c, z) = 1$, $\texttt{FindEBP}(c, z)$ terminates and returns a pair $(c', p) \in \mathrm{C}_1(\Sigma) \times S^+$ such that $\forall p' \in \alpha_{\mathcal{H}}(p)$, $\mathcal{M}(c'[p]) \neq \mathcal{M}(c'[p'])$.*

Corollary 19 *Given a counter-example $w \in \mathrm{SP}(\Sigma)$ such that $\mathcal{H}(w) \neq \mathcal{M}(w)$, a call to $\texttt{FindEBP}(\Box, w)$ returns a pair $(c', p) \in \mathrm{C}_1(\Sigma) \times S^+$ such that for any $p' \in S$, $p \not\sim_L p'$.*

6.3 Termination and correction of the refinement process

Lemma 20 *Algorithms 6 and 7 1. terminate and 2. induce a refinement if they return $\bot$.*

Theorem 6 *Algorithm 9 1. terminates and 2. induces a refinement of $\mathcal{B}$.*

A component of $\mathcal{B}$ may at some point feature more than one access sequence if Algorithm 9 inserts a new representative that cannot be immediately separated from its original class by an inconsistency or an associativity defect. Nevertheless, it remains a temporary issue:

Lemma 21 *Algorithm 9 terminates with $\mathcal{B}$ being sharp.*

Theorem 7 (Correctness of L^λ) *Algorithm 10 1. terminates and 2. returns a PR $\mathcal{H}$ such that $\mathcal{L}(\mathcal{H}) = L$.*

6.4 Complexity analysis

Let $|M| = n$ be the size of the target minimal pomset recognizer $\mathcal{M}$, $k = |\Sigma|$ the size of the alphabet, m and d the maximal size and depth of counter-examples returned by the MAT. We conduct a comparative theoretical analysis of query and symbol complexities, comparing our algorithm to the L^* adaptation of [13].

Query complexity

Building the pack of components. Due to each element of S being eventually distinguished from all the others, $|S| \leq n$ and $|S^+| \leq n^2 + k$. In a similar fashion, new elements are only ever

added to $\mathcal{C}$ when they result in a new class being added to $\mathcal{B}$, thus $|\mathcal{C}| \leq n$.

We build the pack of components by sifting every element of S and S^+ through $\mathcal{D}$. The worst-case scenario arises when $\mathcal{D}$ is a linear tree of depth $n-1$: sifting a pomset may then require up to $n-1$ membership requests. Computing $\mathcal{B}$ then results in $\mathcal{O}(n^3 + k \cdot n)$ membership queries. In the best case scenario, $\mathcal{D}$ is a complete binary tree of depth $\lceil \log_2(n) \rceil$: we then perform $\mathcal{O}(\log_2(n) \cdot n^2 + k \cdot \log_2(n))$ membership queries.

Both cases are similar to L^*, whose set of distinguishing contexts is of size $\lceil \log_2(n) \rceil$ in a best case scenario, and n in the worst case scenario. A predictable result, due to the same lemma holding for finite automata.

Handling counter-examples. As a secondary result of the proof of Theorem 5, given a counter-example w, `FindEBP`$(\square, w)$ performs at most d recursive calls. If we assume $\mathcal{B}$ is sharp, each call asks only two membership queries (Lines 3 and 7). In that case, `FindEBP`$(\square, w)$ only requires $\mathcal{O}(d)$ membership queries.

Let us compare `FindEBP` to the function `HCE` outlined in [13] that instead relies on a prefix traversal of a term of a counter-example instead of exploring a single branch on the fly. `HCE` performs at most $\mathcal{O}(m)$ membership queries. Thus, the closer to a perfect binary tree the term of w considered, the more `FindEBP` outperforms `HCE`: the former will perform at most $\mathcal{O}(\log_2(m))$ queries and the latter, $\mathcal{O}(m)$. However, if instead canonical terms are linear trees, then $d = \Theta(m)$ and both algorithms perform $\mathcal{O}(m)$ queries.

In the context of the L^λ algorithm, while executing Algorithm 9, $\mathcal{B}$ may not be sharp and a component may feature up to n access sequences. Computing the agreement predicates on Lines 6 and 10 therefore requires $\mathcal{O}(n)$ membership queries. Thus, `FindEBP`$(\square, w)$ finds a new component in $\mathcal{O}(n \cdot d)$ membership queries in the worst case scenario.

Rivest-Schapire's method [4] for finite words identifies a breaking point in a counter-example w by performing a binary search over the totally ordered set of prefixes of w, achieving logarithmic complexity w.r.t. the length of w. However, it is worth noting that, for series-parallel pomsets, the set of subpomsets of a counter-example w forms a partial order that prevents us from searching for a breaking point dichotomically.

Total number of queries. In both cases, the number of equivalence queries is bounded by n: in the worst case scenario, each counter-example results in only one component being added to $\mathcal{B}$. Finally, our algorithm performs at most $\mathcal{O}(n^3 + k \cdot n + d \cdot n^2)$ membership queries, whereas [13]'s adaptation of L^* performs at most $\mathcal{O}(n^3 + k \cdot n + m \cdot n)$. Were we to replace `HCE` with `FindEBP`, L^* would require $\mathcal{O}(n^3 + k \cdot n + d \cdot n)$ membership queries at most instead.

Theoretically, L^λ's delayed refinements may burden the counter-example handling process and lead to higher query complexity than L^*; it has however been shown by Howar et al. [8] that L^λ is competitive with state-of-the-art active learning algorithms for rational languages and therefore outperforms L^* due to delayed refinements being rare. It remains to be seen if such an observation holds for recognizable pomset languages.

Symbol complexity

Estimating the symbol complexity of an active learning algorithm is of great practical use. Merely bounding the number of queries overlooks the fact that the actual execution time of membership queries depends on the size of the input.

A study of representatives and contexts. Let us estimate the size of S's greatest access sequence: in the worst case scenario, the $i+1$-th representative $s_{i+1} \in S^+$ added to S is the composition of two copies of the current largest element s_i of $S \setminus \{\varepsilon\}$ for $1 \leq i < n$. Thus, $|s_{i+1}| = 2 \cdot |s_i| + 1$ and trivially, $|s_n| = \mathcal{O}(2^n)$. For any $s \in S \cup S^+$, $|s| = \mathcal{O}(2^n)$. An identical lemma holds for [13]'s adaptation of L^*.

The same is not true of $\mathcal{C}$. Let us estimate the size of $\mathcal{C}$'s greatest context: in the worst case scenario, the $i+1$-th context c_{i+1} added to $\mathcal{C}$ is of the form $c_i[\square \circ s]$ where $s \in S$ and c_i is the current largest element of $\mathcal{C}$ for $1 < i < n$, hence $|c_{i+1}| \leq |c_i| + 1 + |s|$. Trivially, $|c_n| = \mathcal{O}(n \cdot 2^n)$. Thus, for any $c \in \mathcal{C}$, $|c| = \mathcal{O}(n \cdot 2^n)$. In L^*'case, distinguishing contexts are directly extracted from the counter-examples and for any $c \in \mathcal{C}$, $|c| = \mathcal{O}(m)$.

Total symbol complexity. In order to build its observation table, L^* thus requires $\mathcal{O}((2^n+m) \cdot n^3 + k \cdot m \cdot n)$ symbols, while L^λ takes $\mathcal{O}(2^n \cdot n^4 + k \cdot 2^n \cdot n^2)$ symbols to build $\mathcal{B}$. Moreover, each membership query performed by `FindEBP` and `HCE` alike takes up to $\mathcal{O}(m+d \cdot 2^n)$ symbols, the worst case scenario being a right linear tree such that d branches have to be replaced by their access sequences. `FindEBP` performs $\mathcal{O}(d \cdot n^2)$ membership queries, whereas `HCE` performs $\mathcal{O}(m \cdot n)$. Thus, the total symbol complexity of L^* is $\mathcal{O}((2^n+m) \cdot n^3 + k \cdot m \cdot n + m \cdot n \cdot (m + d \cdot 2^n))$, and the symbol complexity of L^λ is $\mathcal{O}(2^n \cdot n^4 + k \cdot 2^n \cdot n^2 + d \cdot n^2 \cdot (m + d \cdot 2^n))$.

It is therefore worth pointing out that L^λ's symbol complexity here does not depend on m with the unavoidable exception of the counter-example handling procedure. Due to m being arbitrary large, should $m = \Omega(n \cdot 2^n)$, then the extra symbols carried by the needlessly large distinguishing contexts directly inferred from counter-examples will burden every further membership query performed by L^* to extend its observation table.

7 Example Run

Let's design the Pomset recognizer $\mathcal{R} = (M, \odot, \oplus, e, i, F)$ with $M = \{q_a, q_b, q_1, q_\perp, q_\mathbf{1}\}$, $e = \mathbf{1}$, $F = \{q_1, \mathbf{1}\}$, $\odot$ and $\oplus$ as defined by figure 8. We then have $L(\mathcal{R}) = (a||b)^*$.

$\odot$	q_a	q_b	q_1	$q_\perp$	$\mathbf{1}$
q_a	$q_\perp$	$q_\perp$	$q_\perp$	$q_\perp$	q_a
q_b	$q_\perp$	$q_\perp$	$q_\perp$	$q_\perp$	q_b
q_1	$q_\perp$	$q_\perp$	q_1	$q_\perp$	q_1
$q_\perp$	$q_\perp$	$q_\perp$	$q_\perp$	$q_\perp$	$q_\perp$
$\mathbf{1}$	q_a	q_b	q_1	$q_\perp$	$\mathbf{1}$

$\oplus$	q_a	q_b	q_1	$q_\perp$	$\mathbf{1}$
q_a	$q_\perp$	q_1	$q_\perp$	$q_\perp$	q_a
q_b	q_1	$q_\perp$	$q_\perp$	$q_\perp$	q_b
q_1	$q_\perp$	$q_\perp$	$q_\perp$	$q_\perp$	q_1
$q_\perp$	$q_\perp$	$q_\perp$	$q_\perp$	$q_\perp$	$q_\perp$
$\mathbf{1}$	q_a	q_b	q_1	$q_\perp$	$\mathbf{1}$

Figure 8: Cayley tables of $\odot$ and $\oplus$.

Let's now consider a teacher T holding $\mathcal{R}$ capable of answering membership and equivalence queries on the latter. We will use L^λ to build a Pomset recognizer H equivalent to $\mathcal{R}$ by querying $\mathcal{R}$:

Step 1 The algorithm starts with $S = \{\}$, $\mathcal{B} = \{\}$ and $\mathcal{D} = Tree(\square)$ and begins by expanding ϵ, which leads to the recursive expansion of a. We then have $S = \{\epsilon, a\}$, $\mathcal{B} = \{B_\epsilon = \{\epsilon, a||b\}, B_a =$

$\{a, b, aa, ab, a||a, ...\}\}$ and $\mathcal{D}$ as defined by figure 9. $\mathcal{B}$ is consistent, $\bullet$-associative, $||$-associative and thus induces an hypothesis H_1.

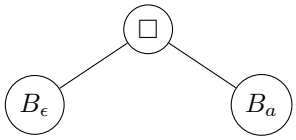

Figure 9: Discrimination tree at Step 1.

Step 2 H_1 is submitted to the teacher, which hands $(a||b)(a||b)(a||b)$ as a counter-example. $FindEBP$ is then called with $c = \square$ and $z = (a||b)(a||b)(a||b)$. A recursive call with $c = \square(a||b)(a||b)$ and $z = (a||b)$ is then performed, which leads to another recursive call with $c = (a||\square)(a||b)(a||b)$ and $z = b$. As z is evaluated by H_1 as a and $(a||a)(a||b)(a||b)$ is not a counter-example while $(a||b)(a||b)(a||b)$ is, we found an effective breaking point at $z = b$ and expand b.

Step 3 After adding b to S, a consistency problem arises as $a \in B_a, b \in B_a, ab \in B_a$ while $a||b \in B_\epsilon$ and $a||ab \in B_a$. This leads to the refinement of $B = B_a$ with $c_1 = \square||b$ which splits B_a into $B_a = \{a\}$ and $B_1 = \{b, ab, aa, a||a, ...\}$. This leads to the expansion of $a||b$. We then have the discrimination tree defined by figure 10.

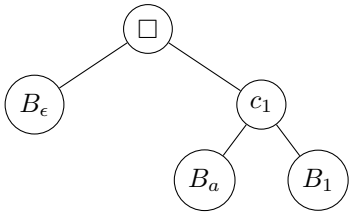

Figure 10: Discrimination tree at Step 3.

Step 4 We have a $||$-associativity defect with $s1 = a, s2 = a, s3 = b, sl = b, sr = \epsilon$. This is remedied by refining B_1 with $c_2 = a||\square$ which splits B_1 into $B_b = \{b\}$ and $B_1 = \{ab, aa, bb, a||a, ...\}$. This leads to the expansion of $a||a$ and gives the discrimination tree given by figure 11.

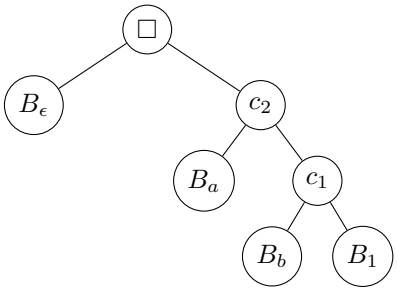

Figure 11: Discrimination tree at Step 4.

Step 5 A consistency default arises as $\epsilon \in B_\epsilon, a||b \in B_\epsilon$ but $e \bullet a \in B_a$ while $(a||b) \bullet a \in B_1$. This is fixed by refining B_ϵ with $c_3 = a||(\Box||b)$ which splits B_ϵ into $B_\epsilon = \{\epsilon\}$ and $B_{a||b} = \{a||b, (a||b)(a||b), ...\}$. We then have the discrimination tree defined by figure 12.

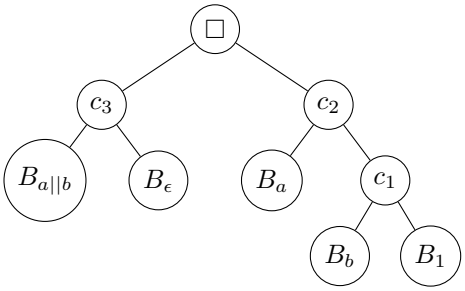

Figure 12: Discrimination tree at Step 5.

Step 6 We have $S = \{\epsilon, a, b, a||b, a||a\}$ and $\mathcal{B} = \{B_\epsilon = \{\epsilon\}, B_{a||b} = \{a||b, (a||b)(a||b), ...\}, B_a = \{a\}, B_b = \{b\}, B_1 = \{aa, bb, ab, a||a, ...\}\}$. Such pack of components if consistent, $\bullet$-associate and $||$-associative and thus induces an hypothesis H_2. Upon comparison of H_2 with the teacher, the equivalence query succeeds and the algorithm terminates.

8 Generating Test Suites for Equivalence Queries

The use of equivalence queries in active learning algorithms is paradoxical: we are striving to infer a formal model from a black box yet our method requires that we compare the hypotheses we submit to the very model we are trying to learn. Practically speaking, we can only rely on membership queries.

We remedy this issue by designing a suitable finite test suite ensuring general equivalence between two pomsets recognizers. Naturally, this is not possible in the general case. However, we assume that the size of the model is bounded w.r.t. to the hypothesis we submit. This test suite extends to recognizable languages of series-parallel pomsets the W-method [5] originally applied to finite state machines accepting words.

Definition 22 (Equivalence on a test suite) *Let $Z \subseteq \mathrm{SP}(\Sigma)$. Two pomset recognizers $\mathcal{R}_1$ and $\mathcal{R}_2$ are said to be Z-*equivalent*, written $\mathcal{H} \equiv_Z \mathcal{M}$, if for any $z \in Z$, $\mathcal{H}(z) = \mathcal{M}(z)$.*

We will in this section consider a hypothesis $\mathcal{H} = (H, \cdot_\mathcal{H}, \|_\mathcal{H}, e_\mathcal{H}, i^\sharp_\mathcal{H}, F_\mathcal{H})$ and a model $\mathcal{M} = (M, \cdot_\mathcal{M}, \|_\mathcal{M}, e_\mathcal{M}, i^\sharp_\mathcal{M}, F_\mathcal{M})$ sharing the same alphabet such that $\mathcal{H}$ and $\mathcal{M}$ are minimal, $|H| = n$, and we know a bound k such that $0 \leq |M| - |H| \leq k$.

8.1 Computing a state cover

The active learning algorithm computes a set of pomsets (the representatives) that reach every state of the hypothesis built, and a set of contexts that distinguish these states. We provide a generic definition of these notions:

Definition 23 (State cover) *A set $P \subseteq \mathrm{SP}(\Sigma)$ is a* state cover *of a reachable pomset recognizer $\mathcal{R} = (R, \odot, \mathbb{1}, e, i, F)$ if $\varepsilon \in P$ and every $r \in R$ admits an access sequence $p \in P$.*

Definition 24 (Characterisation set) *A set of contexts $W \subseteq \mathrm{C}_1(\Sigma)$ is a* characterization set *of a pomset recognizer $\mathcal{R} = (R, \odot, \oplus, e, i, F)$ if $\Box \in W$ and for any $r_1, r_2 \in R$, if r_1 and r_2 are distinguishable, then $\exists c \in W$, $\mathcal{R}(c[r_1]) \neq \mathcal{R}(c[r_2])$.*

The definitions of minimality and characterisation sets result in the following property:

Lemma 25 *Given a minimal pomset recognizer $\mathcal{R} = (R, \odot, \oplus, e, i, F)$, a characterization set W of P, and two states $r_1, r_2 \in R$, if for any $c \in W, \mathcal{R}(c[r_1]) = \mathcal{R}(c[r_2])$, then $r_1 = r_2$.*

The first step of the W-method consists in designing a state cover of the unknown model $\mathcal{M}$ that extends a known state cover of the hypothesis $\mathcal{H}$ by relying on the bound k we postulated and our knowledge that states distinguished in $\mathcal{H}$ are still distinguished in $\mathcal{M}$.

Definition 26 (Extending a state cover.) *Let P be a state cover of $\mathcal{H}$. We introduce the set $L_i^P = \{c[p] \mid m \in \mathbb{N}, c \in \mathrm{C}_m(\Sigma), \delta(c) \leq i, p \in P^m\}$.*

Intuitively, L_i^P consists of all pomsets obtained by inserting access sequences of P in a multi-context of height equal to or smaller than i.

Theorem 8 *Let P be a state cover of $\mathcal{H}$ and W a characterisation set of $\mathcal{H}$ such that $\mathcal{H} \equiv_{W[P]} \mathcal{M}$. Then L_k^P is a state cover of $\mathcal{M}$.*

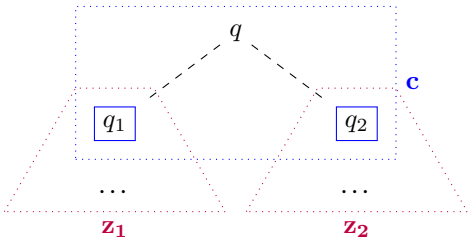

Figure 13: Finding states covered by P in an evaluation tree τ' of w'.

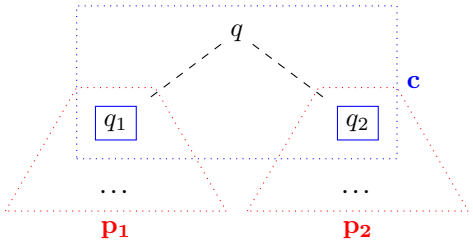

Figure 14: Inserting P's access sequences in τ' to create a new access sequence p.

8.2 Exhaustivity of the test suite

Our goal is to design a *complete* test suite Z, i.e. such that Z-equivalence must imply full equivalence of $\mathcal{H}$ and $\mathcal{M}$, assuming naturally that the hypotheses we have made earlier in this section hold. To do so, we will use a proof inspired by Moerman's [21] that relies on bisimulation. We first define this notion in a similar fashion to finite automata:

Definition 27 (Bisimulation relation) *A* bisimulation relation $\sim$ *between two pomset recognizers $\mathcal{R}_1 = (R_1, \odot_1, \oplus_1, e_1, i_1, F_1)$ and $\mathcal{R}_2 = (R_2, \odot_2, \oplus_2, e_2, i_2, F_2)$ is a binary relation $R_1 \times R_2$ such that:*

1. *$r_1 \sim r_2$ implies that $r_1 \in F_1 \iff r_2 \in F_2$.*
2. *$r_1 \sim r_2$ and $r_1' \sim r_2'$ implies that $r_1 \circ_1 r_2 \sim r_1' \circ_2 r_2'$ for $\circ \in \{\odot, \oplus\}$.*

Lemma 28 *Given a bisimulation relation $\sim$ between $\mathcal{R}_1$ and $\mathcal{R}_2$, if $\mathcal{R}_1$ and $\mathcal{R}_2$ share the same alphabet Σ, the same neutral element e, and for any $x \in \Sigma \cup \{e\}$, $i^\sharp_{\mathcal{R}_1}(x) \sim i^\sharp_{\mathcal{R}_2}(x)$, then $\mathcal{R}_1$ and $\mathcal{R}_2$ are equivalent.*

Our goal is to use P and W to design a test suite Z such that Z-equivalence induces a bisimulation relation.

Theorem 9 *Let P be a state cover of $\mathcal{H}$, W be a characterisation set of $\mathcal{H}$, $L = L_k^P$, $L' = L_{k+1}^P$, and $Z = W[L']$. We introduce the binary relation $\sim$ on $H \times M$:*

$$\sim = \{(i_{\mathcal{H}}^{\sharp}(l), i_{\mathcal{M}}^{\sharp}(l)) \mid l \in L\}$$

If $\mathcal{H}$ and $\mathcal{M}$ are Z-equivalent, then $\sim$ is a bisimulation relation.

9 Generating Pomset Recognizers

In order to test our algorithms, we would need to have access to a wide variety of different Pomset Recognizers. To have a method capable of generating "random" Pomset Recognizers of n states would allow us to employ fuzzing strategies to empirically measure the performances of our version of L^{λ}. The difficulty in generating Pomset Recognisers comes from the associativity and commutativity properties of the operations $\odot$ and $\oplus$. We thus cannot simply fill the operation's Cayley Tables randomly as we need them to respect these invariants.

SAT based methods seem to be the most promising solution. For $\circ = \bullet$, it would imply to start by enforcing $q1 \circ q_2 = q_3$ with $(q_1, q_2, q_3) \in M^3$ and then reduce the associativity's system of equation accordingly. Once the said system is reduced, the operation is repeated until all pairs of states and operations have been assigned a result. Generating the results for $\circ = ||$ can be done using the same strategy, only must commutativity be respected as well as associativity.

The second issue with generating Pomset Recognizer arises when we need to choose whether a state is *final* or *initial* (i.e. labelled by i) or neither. As we cannot be sure that all states can always be reached, using a purely random method of attributing the *final* and *initial* properties to the state can lead to non-reduced (and therefore non-minimal) Pomset Recognizers. This also increases the chances of obtaining a machine that recognizes nothing or everything. The reduced form of any Pomset Recognizer can be found in polynomial time [20], we can thus enforce reduction during our generation, at the cost of trimming some states.

Generating minimal Pomset Recognizers without backtracking seems impossible, and doing so would imply to test an exponential number of possibilities over the number n of desired states. While the aforementioned method does not produce minimal nor reduced machines, minimizing the result allows us to obtain a non-empty minimal Pomset Recognizer most of the time. Repeating this operation numerous times would then permit to generate a benchmark of Pomset Recognizes.

10 Conclusion and Further Developments

In this paper, we have shown that a state-of-the-art active learning algorithm, L^{λ} [8], could be applied to pomset recognizers. It remains to be seen how its compact data structures (discrimination tree, closed sets of representatives and distinguishers) impact query complexity in practice compared to the original adaptation of L^* to pomsets [13]. To this end, we are currently working on an implementation that tackles the following issues:

Representing series-parallel pomsets. We must settle on a canonical representation of pomsets as binary trees that guarantees minimal depth and prevents duplicate queries.

Optimizing counter-example handling. `FindEBP` benefits from the use of canonical terms of minimal depth (ideally, logarithmic w.r.t. the counter-example's size). However, its complexity remains linear if the canonical term is a linear tree. We may therefore try to develop another counter-example handling procedure optimized for linear trees, and dynamically choose a counter-example strategy based on the input term's shape.

Generating a benchmark. We must create a test sample of minimal, reachable pomset recognizers. In that regard, SAT-based approaches look promising.

Testing various scenarios. In order to assess the efficiency of the various techniques discussed in this article, we need to isolate the impact of 1. the discrimination tree $\mathcal{D}$, 2. the new counter-example handling procedure `FindEBP`, and 3. L^λ's lazy refinement.

We plan on adapting various improvements to the W-method, such as the H-method [**?**], to pomset recognizers. We also want to determine whether `FindEBP` can enhance active learning algorithms for tree languages [**?**], which inspired [13]'s algorithm in the first place. We also want to explore the *passive learning* problem for pomset samples: given two non-empty sets $Z^+ \subseteq \mathrm{SP}(\Sigma)$ and $Z^- \subseteq \mathrm{SP}(\Sigma)$ such that $Z^+ \cap Z^- = \emptyset$, find a PR that accepts all the elements of Z^+ and rejects all the elements of Z^-.

Finally, while both van Heerdt's algorithm [13] and ours learn series-parallel pomsets, some use cases such as producer-consumer systems require modelling pomsets that feature N patterns. These scenarios can be effectively formalized using *interval pomsets* [22] and *higher-dimensional automata* (HDA) [23]. The extension of Myhill-Nerode's theorem to languages of HDA [**?**] opens up the possibility of an active learning algorithm for HDA.

Chapter 1

Bibliography

[1] Adrien Pommellet, Amazigh Amrane, and Edgar Delaporte. Active learning techniques for pomset recognizers, 2025. (page 1)

[2] Vaughan R. Pratt. Modeling concurrency with partial orders. *Journal of Parallel Programming*, 15(1):33–71, 1986. (page 4)

[3] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987. (page 4)

[4] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993. (pages 4 and 20)

[5] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng.*, 4(3):178–187, 1978. (pages 4, 5, and 23)

[6] Malte Isberner, Falk Howar, and Bernhard Steffen. The ttt algorithm: A redundancy-free approach to active automata learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 307–322, Cham, 2014. Springer International Publishing. (pages 4 and 12)

[7] Frits Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A new approach for active automata learning based on apartness. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 223–243, Cham, 2022. Springer International Publishing. (pages 4 and 12)

[8] Falk Howar and Bernhard Steffen. Active automata learning as black-box search and lazy partition refinement. In Nils Jansen, Mariëlle Stoelinga, and Petra van den Bos, editors, *A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*, volume 13560 of *Lecture Notes in Computer Science*, pages 321–338. Springer, 2022. (pages 4, 18, 20, and 25)

[9] M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA, 1994. (page 5)

[10] Jay L. Gischer. The equational theory of pomsets. *Theoretical Computer Science*, 61:199–224, 1988. (page 5)

[11] J. Valdes, R. E. Tarjan, and E. L. Lawler. The recognition of series parallel digraphs. *SIAM J. Comput.*, 11:298–313, 1982. (page 5)

[12] S.L. Bloom and Z. Ésik. Free shuffle algebras in language varieties. *Theoretical Computer Science*, 163(1-2):55–98, 1996. (page 6)

[13] Gerco van Heerdt, Tobias Kappé, Jurriaan Rot, and Alexandra Silva. Learning pomset automata. In Stefan Kiefer and Christine Tasson, editors, *Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12650 of *Lecture Notes in Computer Science*, pages 510–530. Springer, 2021. (pages 6, 8, 12, 19, 20, 21, 25, and 26)

[14] K. Lodaya and P. Weil. A Kleene iteration for parallelism. In *Foundations of Software Technology and Theoretical Computer Science*, pages 355–366, 1998. (page 8)

[15] Kamal Lodaya and Pascal Weil. Series-parallel posets: algebra, automata and languages. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 555–565. Springer, 1998. (page 8)

[16] Kamal Lodaya and Pascal Weil. Series–parallel languages and the bounded-width property. *Theoretical Computer Science*, 237(1-2):347–380, 2000. (page 8)

[17] Kamal Lodaya and Pascal Weil. Rationality in algebras with a series operation. *Information and Computation*, 171(2):269–293, 2001. (page 8)

[18] Joel Jones. Abstract syntax tree implementation idioms. *Pattern Languages of Program Design*, 2003. Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003) http://hillside.net/plop/plop2003/papers.html. (page 8)

[19] Mykola Fisun, Hlib Horban, and Kandyba Ihor. Processing of relational algebra expressions by the shunting yard algorithm. In *2019 IEEE 14th International Conference on Computer Sciences and Information Technologies (CSIT)*, volume 3, pages 240–243, 2019. (page 9)

[20] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. 2008. (pages 9 and 25)

[21] Joshua Moerman. *Nominal Techniques and Black Box Testing for Automata Learning*. PhD thesis, Radboud University, 2019. (page 24)

[22] Uli Fahrenberg, Christian Johansen, Georg Struth, and Krzysztof Ziemianski. Posets with interfaces as a model for concurrency. *Inf. Comput.*, 285(Part):104914, 2022. (page 26)

[23] Uli Fahrenberg, Christian Johansen, Georg Struth, and Krzysztof Ziemianski. A kleene theorem for higher-dimensional automata. In Bartek Klin, Slawomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*, volume 243 of *LIPIcs*, pages 29:1–29:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. (page 26)