

Compositional Active Learning of Synchronizing Systems Through Automated Alphabet Refinement

Léo Henry 

Royal Holloway, University of London, UK

Mohammad Reza Mousavi  

King's College London, UK

Thomas Neele   

Eindhoven University of Technology, The Netherlands

Matteo Sammartino  

Royal Holloway, University of London, UK

Abstract

Active automata learning infers automaton models of systems from behavioral observations, a technique successfully applied to a wide range of domains. Compositional approaches for concurrent systems have recently emerged. We take a significant step beyond available results, including those by the authors, and develop a general technique for compositional learning of a synchronizing parallel system with an unknown decomposition. Our approach *automatically refines* the global alphabet into component alphabets while learning the component models. We develop a theoretical treatment of *distributions* of alphabets, i.e., sets of possibly overlapping component alphabets. We characterize *counter-examples* that reveal inconsistencies with global observations, and show how to systematically update the distribution to restore consistency. We present a compositional learning algorithm implementing these ideas, where learning counterexamples precisely correspond to distribution counterexamples under well-defined conditions. We provide an implementation, called COALA, using the state-of-the-art active learning library LEARNLIB. Our experiments show that in more than 630 subject systems, COALA delivers orders of magnitude improvements (up to five orders) in membership queries and in systems with significant concurrency, it also achieves better scalability in the number of equivalence queries.

2012 ACM Subject Classification Theory of computation → Active learning

Keywords and phrases Active learning, Compositional methods, Concurrency theory, Labelled transition systems, Formal methods

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.20

Supplementary Material *Software (Source Code)*: <https://doi.org/10.5281/zenodo.15170685>

Funding *Léo Henry*: EPSRC project Verification of Hardware Concurrency via Model Learning (CLever) – EP/S028641/1.

Mohammad Reza Mousavi: UKRI Trustworthy Autonomous Systems Node in Verifiability – EP/V026801/2; EPSRC project Verified Simulation for Large Quantum Systems (VSL-Q) – EP/Y005244/1; EPSRC project Robust and Reliable Quantum Computing (RoaRQ), Investigation 009 Model-based monitoring and calibration of quantum computations (ModeMCQ) – EP/W032635/1; ITEA/InnovateUK projects GENIUS (600642) and GreenCode (600643).

Thomas Neele: NWO grant VI.Veni.232.224.

Matteo Sammartino: EPSRC project Verification of Hardware Concurrency via Model Learning (CLever) – EP/S028641/1.

Acknowledgements We thank the reviewers for their thorough comments and suggestions, and the authors of [27] for their feedback on an earlier version of this paper.



© Léo Henry, Mohammad Reza Mousavi, Thomas Neele, and Matteo Sammartino; licensed under Creative Commons License CC-BY 4.0

36th International Conference on Concurrency Theory (CONCUR 2025).

Editors: Patricia Bouyer and Jaco van de Pol; Article No. 20; pp. 20:1–20:22



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Automata learning [16] has been successfully applied to learn widely-used protocols such as TCP [12], SSH [13], and QUIC [11], CPU caching policies [38] and finding faults in their black-box actual implementations. Moreover it has been applied to a wide range of applications such as bank cards [1] and biometric passports [2, 28]. There are already accessible expositions of the success stories in this field [20, 36]. However, it is well known that the state-of-the-art automata learning algorithms do not scale beyond systems with more than a few hundreds of input and output symbols in their alphabets [36].

Scalability to larger systems with large input alphabets is required for many real-life systems. This requirement has inspired some recent attempts [25, 31, 27] to come up with compositional approaches to automata learning, to address the scalability issues. Some of the past compositional approaches [31] relied on an a-priori knowledge of the system decomposition, while others [25] considered non-synchronizing state machines or a subset of synchronizing ones to help learning the decomposition [27]. These assumptions need to be relaxed to enable dealing with legacy and black box systems and dealing with synchronizing components. In particular, in the presence of an ever-increasing body of legacy automatically-generated code, architectural discovery is a significant challenge [23, 15, 32]. This is a significant challenge, because refactoring and rejuvenating legacy systems has been posed as a significant application of automata learning [33].

In this paper, we take a significant step beyond the available results and develop a compositional automata learning approach that does not assume any pre-knowledge of the decomposition of the alphabet and allows for an arbitrary general synchronization scheme, common in the theory of automata and process calculi [19]. To this end, we take inspirations from the realizability problem in concurrency theory and use iterative refinements of the alphabet decomposition (called distributions [30]) to arrive in a provably sound decomposition while learning the components' behavior. To our knowledge this is the first result of its kind and the first extension of realizability into the domain of automata learning.

To summarize, the contributions of our paper are listed below:

- We develop a novel theory of system decomposition for LTS synchronization that formally characterizes which alphabet decompositions can accurately model observed behaviors, establishing a theoretical foundation for automated component discovery. Proofs of our theorems are given in the appendix.
- Based on this, we propose a compositional active learning algorithm that dynamically refines component alphabets during the learning process, supporting standard synchronization mechanisms without requiring a priori knowledge of the system's decomposition.
- We implemented our approach as the prototype tool *COALA*, built on the state-of-the-art *LearnLib* framework [22], and evaluated it on over 630 systems from three benchmark sets. Compared to a monolithic approach, *COALA* achieved substantial reductions in queries, with up to five orders of magnitude fewer membership queries and one order fewer equivalence queries across most of our benchmark systems with parallel components, resulting in better overall scalability. The replication package is available at [18].

2 Related work

Realizability of sequential specifications in terms of parallel components has been a long-standing problem in concurrency theory. In the context of Petri nets, this has been pioneered by Ehrenfeucht and Rozenberg [10], followed up by the work of Castellani, Mukund and Thiagarajan [8]. Realizability has been further investigated in other models of concurrency

such as team automata [34], session types [5], communicating automata [17] and labelled transition systems (LTSs) [35]. Related to this line of research is the decomposition of LTSs into prime processes [26]. We are inspired by the work of Mukund [30], characterizing the transition systems that can be synthesized into an equivalent parallel system given a decomposition of their alphabet (called distribution). Mukund explores this characterization for two notions of parallel composition (loosely cooperating- and synchronous parallel composition) and three notions of equivalence (isomorphism, language equivalence, and bisimulation). We base our work on the results of Mukund for loosely cooperating systems and language equivalence. We extend it to define consistency between observations and distributions and refining distributions to reinstate consistency.

Our work integrates two recent approaches on compositional learning: we extend the work on learning synchronous parallel composition of automata [31] by automatically learning the decomposition of the alphabets, through refinement of distributions; moreover, we extend the work on learning interleaving parallel composition of automata [25] by enabling a generic synchronization scheme among components. In parallel to our work, an alternative proposal [27] has appeared to allow for synchronization among interleaving automata; however, the proposed synchronization scheme assumes that whenever two components are not ready to synchronize, e.g., because they produce different outputs on the same input, a special output is produced (or otherwise, the semantic model is output deterministic). We do not assume any such additional information and use a synchronization scheme widely used in the theory of automata and process calculi [19]. Other contributions related to compositional learning include the active learning of *product automata*, a variation of Mealy machine where the output is the combination of outputs of several Mealy machines – as in our case, the component Mealy machines are learned individually [29]; learning of *systems of procedural automata* [14], sets of automata that can call each other in a way similar to procedure calls; learning asynchronously-communicating finite state machines via queries in the form of message sequence charts [6], though using a monolithic approach.

3 Preliminaries

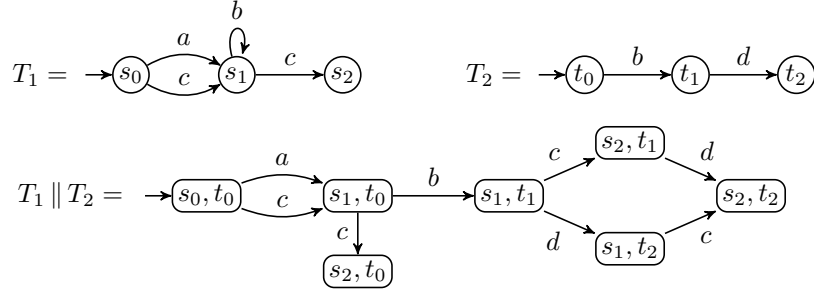
We use Σ to denote a *finite alphabet* of action symbols, and Σ^* to denote the set of finite sequences of symbols in Σ , which we call *traces*; we use $\epsilon \in \Sigma^*$ to denote the empty trace. Given two traces $\sigma_1, \sigma_2 \in \Sigma^*$, we denote their concatenation by $\sigma_1 \cdot \sigma_2$. We refer to the i th element of σ by $\sigma[i]$. The *projection* $\sigma_{\upharpoonright \Sigma'}$ of σ on an alphabet $\Sigma' \subseteq \Sigma$ is the sequence of symbols in σ that are also contained in Σ' : $\epsilon_{\upharpoonright \Sigma'} = \epsilon$ and $\sigma \cdot a_{\upharpoonright \Sigma'} = \sigma_{\upharpoonright \Sigma'} \cdot a$ if $a \in \Sigma'$ and $\sigma_{\upharpoonright \Sigma'}$ otherwise. We generalize this notation to sets (and thus languages), such that $S_{\upharpoonright \Sigma'} = \{\sigma_{\upharpoonright \Sigma'} \mid \sigma \in S\}$. Given a set S , we write $|S|$ for its cardinality. We write $\text{img}(f)$ for the image of a function f .

3.1 Labelled Transition Systems

In this work we represent the state-based behavior of a system as a *labelled transition system*.

► **Definition 3.1** (Labelled Transition System). *A labelled transition system (LTS) is a four-tuple $T = (S, \Sigma, \rightarrow, \hat{s})$, where S is a set of states; Σ is a finite alphabet of actions; $\rightarrow \subseteq S \times \Sigma \times S$ is a transition relation; $\hat{s} \in S$ is an initial state.*

We write in infix notation $s \xrightarrow{a} t$ for $(s, a, t) \in \rightarrow$. We say that an action a is *enabled* in s , written $s \xrightarrow{a}$, if there is t such that $s \xrightarrow{a} t$. The transition relation and the notion of enabled-ness are also extended to traces $\sigma \in \Sigma^*$, yielding $s \xrightarrow{\sigma} t$ and $s \xrightarrow{\sigma}$.



■ **Figure 1** The parallel composition of two LTSs.

► **Definition 3.2** (Language of an LTS). *The language of T is the set of traces enabled from the starting state, formally: $\mathcal{L}(T) = \{\sigma \in \Sigma^* \mid \hat{s} \xrightarrow{\sigma}\}$.*

Note that languages of LTSs are always prefix-closed, because every prefix of an enabled trace is necessarily enabled. LTSs correspond exactly to prefix-closed automata.

The parallel composition of a finite set of LTSs is a product model representing all possible behaviors when the LTSs synchronize on shared actions. Intuitively, an action a can be performed when all LTSs that have a in their alphabet can perform it in their current state. The other LTSs remain idle during the transition.

► **Definition 3.3** (Parallel composition). *Given n LTSs $T_i = (S_i, \Sigma_i, \rightarrow_i, \hat{s}_i)$ for $1 \leq i \leq n$, their parallel composition, denoted $\prod_{i=1}^n T_i$, is an LTS $(S_1 \times \dots \times S_n, \bigcup_{i=1}^n \Sigma_i, \rightarrow, (\hat{s}_1, \dots, \hat{s}_n))$, where the transition relation $\rightarrow$ is given by the following rule:*

$$\frac{\begin{array}{l} s_i \xrightarrow{a} t_i \quad \text{for all } i \text{ such that } a \in \Sigma_i \\ s_j = t_j \quad \text{for all } j \text{ such that } a \notin \Sigma_j \end{array}}{(s_1, \dots, s_n) \xrightarrow{a} (t_1, \dots, t_n)}$$

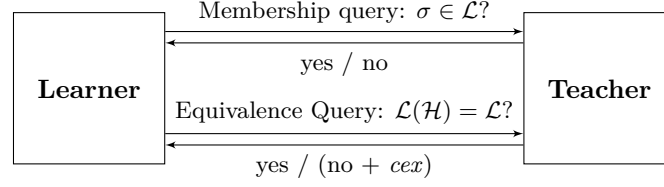
We say that an action a is *local* if there is exactly one i such that $a \in \Sigma_i$; otherwise, it is called *synchronizing*. The parallel composition of LTSs thus forces individual LTSs to cooperate on synchronizing actions; local actions can be performed independently. We typically refer to the LTSs that make up a composite LTS as *components*. For a parallel composition of two LTSs, we use the infix notation, i.e., $T \parallel T'$, when convenient. Synchronization of components corresponds to communication between components in the real world.

We define the corresponding notion for languages on restricted alphabets.

► **Definition 3.4** (Parallel composition of languages). *Given n languages and alphabets $(\mathcal{L}_i, \Sigma_i)$ such that $\mathcal{L}_i \subseteq \Sigma_i^*$ for all $1 \leq i \leq n$, let $\Sigma = \bigcup_{i=1}^n \Sigma_i$. We define $\prod_{i=1}^n (\mathcal{L}_i, \Sigma_i)$ as*

$$\{\sigma \in \Sigma^* \mid \forall 1 \leq i \leq n. \sigma|_{\Sigma_i} \in \mathcal{L}_i\}.$$

► **Example 3.5** (Running example). Consider the LTSs T_1 and T_2 given in Figure 1, with the respective alphabets $\{a, b, c\}$ and $\{b, d\}$. Their parallel composition is depicted at the bottom of Figure 1. Here a , c and d are local actions, whereas b is synchronizing. Note that, although T_2 can perform b from its initial state t_0 , there is no b transition from (s_0, t_0) in $T_1 \parallel T_2$, because b is not enabled in s_0 . Action b can only be performed in $T_1 \parallel T_2$ after T_1 does an a or a c and moves to s_1 , which is captured as the a and c transitions from (s_0, t_0) .



■ **Figure 2** Active automata learning for a target language $\mathcal{L}$.

3.2 Active Automata Learning

In active automata learning [4], a *Learner* infers an automaton model of an unknown language $\mathcal{L}$ by querying a *Teacher*, which knows $\mathcal{L}$ and answers two query types (see Figure 2):

- *Membership queries*: is a trace σ in $\mathcal{L}$? The Teacher replies yes/no.
- *Equivalence queries*: given a *hypothesis* model $\mathcal{H}$, is $\mathcal{L}(\mathcal{H}) = \mathcal{L}$? The Teacher either replies yes or provides a *counter-example* – a trace that is in one language but not in the other.

Algorithms based on this framework – Angluin’s L^* being the classical example – converge to a canonical model (e.g., the minimal DFA) of the target language. In practice, the Teacher is realized as an interface to the System Under Learning (SUL): membership queries become tests on the SUL, and equivalence queries are approximated via systematic testing strategies [7, 24].

During learning, the learner gathers *observations* about the SUL. While these observations are typically organized in a data structure (e.g., a table or a tree), they can be abstractly represented as a partial function mapping traces to their accepted (+) or rejected (−) status.

► **Definition 3.6** (Observation function). *An observation function over Σ is a partial function $\text{Obs} : \Sigma^* \rightarrow \{+, -\}$.*

We write $\text{Dom}(\text{Obs})$ for the domain of Obs and only consider observation functions with a finite domain. We sometimes represent an observation function Obs as the set of pairs $\{(\sigma, \text{Obs}(\sigma)) \mid \sigma \in \text{Dom}(\text{Obs})\}$.

► **Definition 3.7** (Observation function/language agreement). *An observation function Obs agrees with a language $\mathcal{L}$, notation $\mathcal{L} \models \text{Obs}$, whenever $\sigma \in \mathcal{L} \Leftrightarrow \text{Obs}(\sigma) = +$, for all $\sigma \in \text{Dom}(\text{Obs})$.*

To compositionally learn a model formulated as a parallel composition of LTSs $\prod_{i=1}^n T_i$, we define how to derive the *local* observation functions that will be used for the components.

► **Definition 3.8** (Local observation function). *Given a sub-alphabet $\Sigma_i \subseteq \Sigma$, a local observation function $\text{Obs}_{\Sigma_i} : \Sigma_i^* \rightarrow \{+, -\}$ is defined such that $\text{Dom}(\text{Obs}_{\Sigma_i}) = \text{Dom}(\text{Obs})_{\upharpoonright \Sigma_i}$ and $\text{Obs}_{\Sigma_i}(\sigma') = \bigvee_{\{\sigma \in \text{Dom}(\text{Obs}) \wedge \sigma_{\upharpoonright \Sigma_i} = \sigma'\}} \text{Obs}(\sigma)$, for all $\sigma' \in \text{Dom}(\text{Obs}_{\Sigma_i})$.*

This definition is taken to mimic the behavior of parallel composition, i.e., a component T_i accepts σ' if and only if there is some σ such that $\sigma_{\upharpoonright \Sigma_i} = \sigma'$ and $\prod_{i=1}^n T_i$ accepts σ .

► **Example 3.9.** Consider again the LTSs from Figure 1 and suppose we are given the following observation function for $T_1 \parallel T_2$: $\text{Obs} : a \mapsto +; aa \mapsto -; abd \mapsto +$. The local observation functions we obtain for T_1 and T_2 are, respectively:

$$\text{Obs}_{\{a,b,c\}} : a \mapsto +; aa \mapsto -; ab \mapsto + \quad \text{Obs}_{\{b,d\}} : \epsilon \mapsto +; bd \mapsto +.$$

The observation $\text{Obs}(abd) = +$ requires both components to cooperate, hence $\text{Obs}_{\{a,b,c\}}(ab) = +$ and $\text{Obs}_{\{b,d\}}(bd) = +$. We derive $\text{Obs}_{\{b,d\}}(\epsilon) = +$ from $\text{Obs}(a) \vee \text{Obs}(aa) = +$, since the projection of both these traces to $\{b,d\}$ is ϵ .

4 Distributions

In this section, we first discuss how we decompose the global alphabet into a *distribution*, i.e., a set of potentially overlapping local alphabets. We then give some properties of distributions and their relation to observation functions. Based on these, we explain how to extend a distribution to model a given observation function.

4.1 Distributions and Observations

In a model expressed as a parallel composition $\parallel_i T_i$, permuting symbols belonging to different local alphabets does not affect membership of the language. For example, in Figure 1, because $abcd$ is in the language, we directly know that $abdc$ is too, as both c and d are local to different components. To formalize this, we first formally define distributions as follows.

► **Definition 4.1** (Distribution). *A distribution of an alphabet Σ is a set $\Omega = \{\Sigma_1, \dots, \Sigma_n\}$ such that $\bigcup_{i=1}^n \Sigma_i = \Sigma$.*

For the rest of this section, we fix an alphabet Σ , a distribution $\Omega = \{\Sigma_1, \dots, \Sigma_n\}$ of Σ and an observation function Obs over Σ unless otherwise specified.

For a given distribution Ω , we define below the class of languages, called *product languages* over Ω , that can be represented over that distribution.

► **Definition 4.2** (Product language). *$\mathcal{L}$ is a product language over Ω , notation $\Omega \models \mathcal{L}$, iff there exists a family of languages $\{\mathcal{L}_i\}_{1 \leq i \leq n}$, where $\mathcal{L}_i \subseteq \Sigma_i^*$ for all $1 \leq i \leq n$, such that $\mathcal{L} = \parallel_{i=1}^n (\mathcal{L}_i, \Sigma_i)$.*

► **Example 4.3.** In Example 3.5, it is clear by construction that $\{\{a, b, c\}, \{b, d\}\} \models \mathcal{L}(T_1 \parallel T_2)$. However, $\mathcal{L}(T_1 \parallel T_2)$ is not a product language over $\Omega_{\text{singlets}} = \{\{a\}, \{b\}, \{c\}, \{d\}\}$ because any product language over Ω_{singlets} should allow for permuting a and b and thus, would fail to capture the fact that b can only come after one a .

We recall the following key lemma for product languages.

► **Lemma 4.4** ([30], Lemma 5.2). *A language $\mathcal{L}$ is a product language over Ω if and only if $\mathcal{L} = \parallel_{i=1}^n (\mathcal{L}_{\upharpoonright \Sigma_i}, \Sigma_i)$.*

We can now define product observations over Ω , i.e., an observation that can be generated by a product language over Ω .

► **Definition 4.5** (Product observation). *Obs is a product observation over Ω , notation $\Omega \models \text{Obs}$, iff there exists a language $\mathcal{L}$ such that $\Omega \models \mathcal{L}$ and $\mathcal{L} \models \text{Obs}$. We conversely say that Ω models Obs .*

While Definition 4.5 does not prescribe how to find such a distribution given an observation function, it can be used to detect precisely when a current distribution is not consistent with observations and must be updated. This results in the following proposition linking local and global observations for a given distribution: an observation is a product observation over a distribution if and only if its projections according to the distribution hold the same information as the observation itself. The proof follows largely from Lemma 4.4.

► **Proposition 4.6.** *$\Omega \models \text{Obs}$ if and only if for all traces $\sigma \in \text{Dom}(\text{Obs})$ it holds that $\text{Obs}(\sigma) = \bigwedge_{\Sigma_i \in \Omega} \text{Obs}_{\Sigma_i}(\sigma_{\upharpoonright \Sigma_i})$.*

► **Example 4.7.** Following from Example 4.3, consider the following observation function based on $\mathcal{L}(T_1 \parallel T_2)$ from Example 3.5:

$$\text{Obs} : \epsilon \mapsto +; a \mapsto +; ab \mapsto +; b \mapsto -; c \mapsto +; d \mapsto -.$$

Using the above proposition, we can verify that $\Omega_{\text{singles}} = \{\{a\}, \{b\}, \{c\}, \{d\}\} \not\models \text{Obs}$. This is because $\text{Obs}(b) = -$, whereas for all $\Sigma_i \in \Omega_{\text{singles}}$, $\text{Obs}_{\Sigma_i}(b \upharpoonright_{\Sigma_i}) = +$ since $ab \upharpoonright_{\{b\}} = b$ causes $\text{Obs}_{\{b\}}(b) = +$ and for other alphabets $b \upharpoonright_{\Sigma_i} = \epsilon$ and $\text{Obs}_{\Sigma_i}(\epsilon) = +$. In contrast, $\Omega_{\{a,b\}} = \{\{a, b\}, \{c\}, \{d\}\} \models \text{Obs}$ since the alphabet $\{a, b\}$ allows for distinguishing observations b and ab .

In our algorithm, this check on local and global observation functions is used to trigger an update of the current distribution exactly when necessary.

Based on the above proposition, we now define *counter-examples to a distribution*. By definition of local observations, if $\text{Obs}(\sigma) = +$, then $\text{Obs}_{\Sigma_i}(\sigma \upharpoonright_{\Sigma_i}) = +$. Hence, to obtain $\text{Obs}(\sigma) \neq \bigwedge_{\Sigma_i \in \Omega} \text{Obs}_{\Sigma_i}(\sigma \upharpoonright_{\Sigma_i})$, we must have a globally negative observation σ_N and a set of globally positive observations whose projections to the local components match the projections of σ_N , indicating a mismatch between global and local observations.

► **Definition 4.8** (Counter-example to a distribution). *A counter-example to $\Omega \models \text{Obs}$ is a pair $(\sigma_N, P) \in \text{Dom}(\text{Obs}) \times \text{Dom}(\text{Obs})^\Omega$ with*

- σ_N a negative observation $\text{Obs}(\sigma_N) = -$;
- P a function that maps each $\Sigma_i \in \Omega$ to a positive observation σ_{Σ_i} , i.e., $\text{Obs}(\sigma_{\Sigma_i}) = +$, such that $\sigma_N \upharpoonright_{\Sigma_i} = \sigma_{\Sigma_i} \upharpoonright_{\Sigma_i}$.

We call $\text{img}(P)$ the positive image of the counter-example. We write $\text{CED}(\Omega, \text{Obs})$ for the set of such counter-examples.

Although these counter-examples are not necessarily related to learning, we use the same terminology as in active learning. This is because the two concepts are directly linked in our case, as will be explained later.

► **Example 4.9.** Reusing the observation function Obs and the singleton distribution Ω_{singles} defined in Example 4.7, for every element of $\text{CED}(\text{Obs}, \Omega_{\text{singles}})$ we have $\sigma_N = b$. and $P(\{b\}) = ab$. For the remaining elements of Ω , there are more choices: $\{a\}$ can be mapped to either ϵ or c ; $\{c\}$ to either ϵ , a or ab ; and $\{d\}$ to either ϵ , a , ab or c .

Proposition 4.6, specialized to our definition of counter-examples, yields the following corollary, which will be used in the following to detect that a distribution is a model.

► **Corollary 4.10.** $\Omega \models \text{Obs} \iff \text{CED}(\Omega, \text{Obs}) = \emptyset$.

4.2 Resolving a Counter-example

Given a distribution Ω and a fixed observation function Obs , one key question is how to extend Ω to a new distribution Ω' modelling Obs . This is a difficult problem, as new counter-examples can arise when extending a distribution. In this subsection, we explain how to resolve a single counter-example as a first step.

When a counter-example (σ_N, P) to $\Omega \models \text{Obs}$ exists, it reveals a limitation in the distribution Ω : the projections of σ_N coincide with projections of elements in P , making them indistinguishable under the current components. To resolve such counter-examples, it is thus necessary and sufficient to augment Ω with new components that disrupt this matching. In the following, we will fix $(\sigma_N, P) \in \text{CED}(\Omega, \text{Obs})$ as a counter-example to $\Omega \models \text{Obs}$.

More precisely, for each pair of traces (σ_N, σ) with $\sigma \in P$, it suffices to identify a *discrepancy* between them. There are two types of discrepancies: *multiplicity discrepancies* and *order discrepancies*. A multiplicity discrepancy is a symbol occurring a different number of times in each trace. For this, given a trace σ , let $\Sigma^m(\sigma)$ denote the multiset of symbols occurring in σ . Note that $\Sigma^m(\sigma) = \Sigma^m(\sigma')$ if and only if σ is a permutation of σ' . The symmetric difference of multisets A and B is denoted $A \Delta B$.

► **Definition 4.11** (Multiplicity discrepancy). *Given a $\Sigma_i \in \Omega$, the set of multiplicity discrepancies for Σ_i is $\mathcal{D}_m^{\Sigma_i}(\sigma_N, P) = \Sigma^m(\sigma_N) \Delta \Sigma^m(P(\Sigma_i))$.*

We now define an *order discrepancy*, i.e., a pair of symbols whose relative positions differ between the traces. We do this by considering whether symbols that are *not* a multiplicity discrepancy, i.e., those appearing the same number of times in both traces, are permuted. We choose the permutation such that the relative order of identical symbols is maintained.

► **Definition 4.12** (Order discrepancy). *Given $\Sigma_i \in \Omega$, let $\theta = \Sigma \setminus (\Sigma^m(\sigma_N) \Delta \Sigma^m(P(\Sigma_i)))$ be the symbols on which σ_N and $P(\Sigma_i)$ agree and define $\sigma'_N = \sigma_N|_{\theta}$. Let π be the unique permutation such that $\sigma'_N = \pi(P(\Sigma_i)|_{\theta})$ and $\sigma'_N[j] = \sigma'_N[k] \implies \pi(j) < \pi(k)$, for all $j < k$. The set of order discrepancies for Σ_i is then:*

$$\mathcal{D}_o^{\Sigma_i}(\sigma_N, P) = \{\{\sigma'_N[j], \sigma'_N[k]\} \mid k < j \wedge \pi(k) > \pi(j)\}$$

Multiplicity and order discrepancies can be found in linear and quadratic time, respectively. Finally, we define the *discrepancies* for a counter-example as sets that contain at least a discrepancy of either type for each alphabet in Ω .

► **Definition 4.13** (Discrepancy set). *A set $\delta \subseteq \Sigma$ is a discrepancy for (σ_N, P) iff for all $\Sigma_i \in \Omega$, either $\mathcal{D}_m^{\Sigma_i}(\sigma_N, P) \cap \delta \neq \emptyset$ or there is $\delta^{\Sigma_i} \in \mathcal{D}_o^{\Sigma_i}(\sigma_N, P)$ such that $\delta^{\Sigma_i} \subseteq \delta$. We write $\mathcal{D}(\sigma_N, P)$ for the set of all discrepancies for the counter-example (σ_N, P) .*

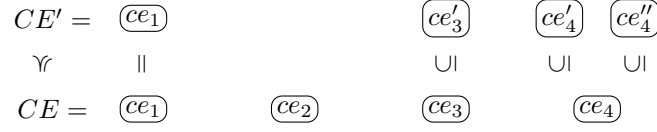
For a set of counter-examples $\{ce_1, \dots, ce_n\}$, we write $\mathcal{D}(\{ce_1, \dots, ce_n\}) = \{\{\delta_1, \dots, \delta_n\} \mid \forall i. \delta_i \in \mathcal{D}(ce_i)\}$, representing all possible selections of one discrepancy per counter-example.

► **Example 4.14** (Multiplicity discrepancy). Following Example 4.9 with the singleton distribution $\Omega_{singles} = \{\{a\}, \{b\}, \{c\}, \{d\}\}$, consider the counter-example $(\sigma_N, P) = (b, (\{a\} \mapsto \epsilon, \{b\} \mapsto ab, \{c\} \mapsto \epsilon, \{d\} \mapsto \epsilon))$. We find the following multiplicity discrepancies: $\mathcal{D}_m^{\Sigma_i}(\sigma_N, P) = \{b\}$, for $\Sigma_i \in \{\{a\}, \{c\}, \{d\}\}$, because b occurs once in σ_N vs. zero times in $P(\Sigma_i)$, and $\mathcal{D}_m^{\{b\}}(\sigma_N, P) = \{a\}$. Hence, $\mathcal{D}(\sigma_N, P)$ includes all subsets of $\{a, b, c, d\}$ containing $\{a, b\}$. For a different counter-example such as $(\sigma_N, P') = (b, (\{a\} \mapsto c, \{b\} \mapsto ab, \{c\} \mapsto ab, \{d\} \mapsto c))$, we obtain $\mathcal{D}_m^{\{a\}}(\sigma_N, P') = \mathcal{D}_m^{\{d\}}(\sigma_N, P') = \{b, c\}$ and $\mathcal{D}_m^{\{b\}}(\sigma_N, P') = \mathcal{D}_m^{\{c\}}(\sigma_N, P') = \{a\}$, so $\mathcal{D}(\sigma_N, P')$ includes any subset of $\{a, b, c, d\}$ that contains either $\{a, b\}$ or $\{a, c\}$.

► **Example 4.15** (Order discrepancy). Consider a singleton distribution $\Omega_{singles} = \{\{a\}, \{b\}, \{c\}\}$ with $\text{Obs}(abc) = +$ and $\text{Obs}(bac) = -$. This yields the counter-example $(\sigma_N, P) = (bac, (\{a\} \mapsto abc, \{b\} \mapsto abc, \{c\} \mapsto abc))$. For each $\Sigma_i \in \Omega_{singles}$, we find $\mathcal{D}_o^{\Sigma_i}(\sigma_N, P) = \{\{a, b\}\}$. Intuitively, these discrepancies reveals that singleton components allow for all permutations of a and b , but the observation function forbids some of them. Therefore, $\mathcal{D}(\sigma_N, P)$ includes all subsets of $\{a, b, c\}$ containing $\{a, b\}$.

We can now state that discrepancies are both sufficient and necessary additions to a distribution in order to eliminate their counter-examples.

► **Proposition 4.16.** *Suppose there exists $(\sigma_N, P) \in \text{CED}(\Omega, \text{Obs})$. For each discrepancy $\delta \in \mathcal{D}(\sigma_N, P)$, $(\sigma_N, P) \notin \text{CED}(\Omega \cup \{\delta\}, \text{Obs})$. Conversely, for any distribution Ω' of Σ where $\delta \not\subseteq \Sigma_i$, for all $\delta \in \mathcal{D}(\sigma_N, P)$ and all $\Sigma_i \in \Omega'$, (σ_N, P) is a counter-example to $\Omega' \models \text{Obs}$.*



■ **Figure 3** The different relations between the elements of two sets of counter-examples $CE \preceq CE'$.

4.3 Extending a Distribution to Model an Observation Function

Using the previous subsection as a basis, we leverage structural properties of distributions to restrict the possible counter-examples that can appear when updating the distribution. Finally, we devise an iterative process that is guaranteed to converge to a distribution modelling Obs.

Pre-ordering of Distributions

Distributions can be preordered by their “connecting power”, i.e., by the extent to which they connect symbols together as part of the same alphabets.

► **Definition 4.17** (Connectivity preorder). *Given two distributions Ω and Ω' of alphabet Σ , we say that Ω is less connecting than Ω' and write $\Omega \preceq \Omega'$ when $\forall \Sigma_i \in \Omega. \exists \Sigma_j \in \Omega'. \Sigma_i \subseteq \Sigma_j$ (equivalently, Ω' is said to be more connecting than Ω). The relation is strict, written $\Omega < \Omega'$, when $\Omega' \not\preceq \Omega$. The relation $\preceq$ forms a preorder with finite chains.*

We relate this notion to the sets of counter-examples for a fixed observation function to show that adding connections in a distribution makes the counter-example set progress along a preorder. For this, we first define a notion of inclusion for counter-examples.

► **Definition 4.18** (Counter-example inclusion). *Consider two distributions Ω and Ω' of Σ , (σ_N, P) a counter-example to $\Omega \models \text{Obs}$ and (σ'_N, P') a counter-example to $\Omega' \models \text{Obs}$. We write $(\sigma_N, P) \subseteq (\sigma'_N, P')$ whenever $\sigma_N = \sigma'_N$ and $\text{img}(P) \subseteq \text{img}(P')$. The strict inclusion $(\sigma_N, P) \subset (\sigma'_N, P')$ holds whenever $\sigma_N = \sigma'_N$ and $\text{img}(P) \subset \text{img}(P')$.*

In its simplest form, progress means eliminating counter-examples from the current set of counter-examples $CED(\Omega, \text{Obs})$. However, a counter-example ce might be replaced by new counter-examples ce' such that $ce \subseteq ce'$, which emerge when new connections are added to the distribution. Hence, progress means that some counter-examples are either eliminated or replaced by subsuming ones, as depicted in Figure 3.

► **Definition 4.19** (Counter-example set preordering). *Consider CE and CE' sets of counter-examples. We write $CE \preceq CE'$ when $\forall ce' \in CE'. \exists ce \in CE. ce \subseteq ce'$. We write $CE < CE'$ when, furthermore, either $ce \subset ce'$ or $CE \preceq CE' \setminus \{ce'\}$, for $ce \in CE$ and $ce' \in CE'$.*

► **Example 4.20.** We give a short example of the preorder on set of counter-examples inspired by our running example, that will later appear in Example 5.5:

$$\{(b, (\{a, b\} \mapsto cb, \{c\} \mapsto \epsilon, \{d\} \mapsto \epsilon))\} \preceq \{(b, (\{a, b\} \mapsto cb, \{b, c\} \mapsto ab, \{d\} \mapsto \epsilon))\}.$$

Notice that these two singleton sets' only difference is $\{c\} \mapsto \epsilon$ replaced by $\{b, c\} \mapsto ab$ such that the image of the first is strictly included in the image of the second.

Using the above definitions, we can prove that increasing the connecting power of a distribution ensures that the set of counter-example progresses.

► **Proposition 4.21.** *Consider two distributions Ω and Ω' of Σ . We have $\Omega \preceq \Omega' \Rightarrow CED(\Omega, \text{Obs}) \preceq CED(\Omega', \text{Obs})$.*

As an immediate consequence, whenever Ω has no counter-examples, any distribution Ω' that is more connecting than Ω will also have none, i.e., both distributions will model the same observations.

► **Corollary 4.22.** *Let Ω be a distribution of Σ such that $\Omega \models \text{Obs}$. For any distribution Ω' of Σ such that $\Omega \preceq \Omega'$, we have $\Omega' \models \text{Obs}$.*

Fixing the distribution

Using Propositions 4.16 and 4.21, from an initial distribution $\Omega \not\models \text{Obs}$ we can create a more connecting one that entails a strict progression in counter-examples.

► **Corollary 4.23.** *Suppose that $CED(\Omega, \text{Obs}) \neq \emptyset$. For $(\sigma_N, P) \in CED(\Omega, \text{Obs})$, we pick a discrepancy $\delta_{(\sigma_N, P)} \in \mathcal{D}(\sigma_N, P)$. For any non-empty subset S of $CED(\Omega, \text{Obs})$, let $\Omega' = \Omega \cup \{\delta_{ce} \mid ce \in S\}$. Then $\Omega \prec \Omega'$ and $CED(\Omega, \text{Obs}) \prec CED(\Omega', \text{Obs})$.*

► **Remark 4.24.** Corollary 4.23 gives us the freedom to select any discrepancy δ_{ce} , for each counter-example ce . We can select discrepancies that result in a least connecting distribution, which yields a locally optimal greedy strategy for progress. The intuition behind this choice is that it leads to: (1) more components that are individually smaller and easier to learn; and (2) fewer synchronizing actions between components, thus reducing the complexity of coordination among learners (see Section 5.1 for details).

By iteratively applying this Corollary, we can eliminate counter-examples until reaching a distribution that models the observations. This leads to the following convergence result:

► **Theorem 4.25.** *Suppose $\Omega \not\models \text{Obs}$. The above process converges to a distribution $\Omega' \models \text{Obs}$ such that $\Omega \prec \Omega'$ after finitely many of steps. For choosing $S = CED(\Omega, \text{Obs})$ at each step, the number of steps is bounded by $|\mathcal{P}(\Sigma)|$.*

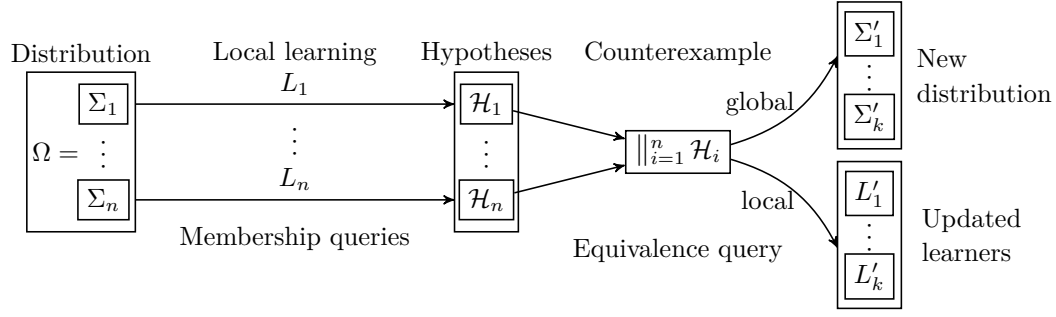
Canonical distributions: inducing a partial order

Distributions have few constraints: they only need to span the entire alphabet, which leaves room for redundancies. We propose to remove redundancies without affecting the distribution's connecting power, by removing alphabets completely contained within another.

► **Definition 4.26** (Canonical distribution). *Consider a distribution $\Omega = \{\Sigma_1, \dots, \Sigma_n\}$ and $\text{Sub} = \{\Sigma_i \in \Omega \mid \exists \Sigma_j \in \Omega. \Sigma_i \subset \Sigma_j\}$. The associated canonical distribution is $\llbracket \Omega \rrbracket_{\preceq} = \Omega \setminus \text{Sub}$.*

As one would expect $\llbracket \cdot \rrbracket_{\preceq}$ collapses equivalence classes of the preorder $\preceq$ (i.e., $\Omega \preceq \Omega'$ and $\Omega' \preceq \Omega$) to create a strict partial order. Canonical distributions allow minimizing the number of alphabets in the distribution while retaining the same connecting power. This means that counter-examples can be easily translated between a distribution and its canonical form, and hence the following proposition.

► **Proposition 4.27.** $CED(\llbracket \Omega \rrbracket_{\preceq}, \text{Obs}) = \emptyset \Leftrightarrow CED(\Omega, \text{Obs}) = \emptyset$



■ **Figure 4** Overview of the algorithm.

5 Compositional Learning Algorithm

In this section, we present our algorithm to compositionally learn an unknown system $SUL = M_1 \parallel M_2 \parallel \dots \parallel M_n$ consisting of the parallel composition of n LTSs, given only a Teacher for the whole SUL and knowledge of the global alphabet Σ_{SUL} .

A bird's eye view of the algorithm is provided in Figure 4. The key idea is to learn each component via a separate learner. Each learner L_i poses membership queries independently, which are suitably translated to queries for the global Teacher, until it produces a hypothesis $\mathcal{H}_i$. Hypotheses returned by local learners are combined to create a global equivalence query. Counter-examples obtained through equivalence queries $(\sigma, b) \in \Sigma_{SUL}^* \times \{-, +\}$ are classified as either *global* or *local*. They are global when the updated observations $\text{Obs}' = \text{Obs} \cup \{(\sigma, b)\}$ and the current distribution Ω of Σ_{SUL} are such that $\Omega \not\models \text{Obs}'$, and local otherwise. Global counter-examples are used to refine the distribution Ω , possibly creating components/learners. Local counter-examples are used to update the state of local learners.

We briefly recall the local learning procedure, which was introduced in previous work, before moving on to presenting the details of our main algorithm in Section 5.2.

5.1 Local learners

For each alphabet Σ_i in the distribution Ω , we spawn a learner L_i that is tasked with formulating a hypothesis for the corresponding component. A key feature of this learner is the ability to translate *local* membership queries into *global* ones for the entire SUL, and to interpret the results of global queries at the local level [31]. The main difficulty is that this translation is not always feasible: when a membership query contains synchronizing actions, cooperation with the other learners is required. This should be done in a way that is consistent with the current distribution, *i.e.*, if $\Omega \models \text{Obs}$, then for any global membership query result $(\sigma, b) \in \Sigma_{SUL}^* \times \{-, +\}$, we require $\Omega \models \text{Obs} \cup \{(\sigma, b)\}$. Furthermore, due to the nature of local observation functions, it is possible that at first $\text{Obs}_{\Sigma_i}(\sigma) = -$, and later this becomes $\text{Obs}_{\Sigma_i}(\sigma) = +$ as the set of global observations is extended through counter-examples; the learners must account for this. In summary, local learners should:

- translate local queries to global ones, preserving $\Omega \models \text{Obs}$. If a local query cannot be translated, the answer is “unknown”;
- be able to handle “unknown” entries in the learning data structure (*e.g.*, an observation table), ensuring progress even in the presence of incomplete information; and
- be able to correct negative counter-examples on the basis of a later positive counter-example.

Our previous work [31] shows one way to implement such a learner as an extension of the L^* algorithm [4], in which LTSs are represented as prefix-closed DFAs. However, we remark that the algorithms proposed in the present work are independent of the implementation of these local learners, as long as they satisfy the requirements above. Thus, our L^* -based implementation can be swapped out for other active learner algorithms, such as TTT [21].

5.2 Main algorithm

The main algorithm is presented in Algorithm 1. Initially, **Obs** is empty, and the distribution Ω contains singletons of the alphabet of the SUL. The algorithm iteratively performs the following steps.

Each learner is run in parallel until producing a hypothesis. Observations **Obs** are suitably updated to record the interactions with the Teacher. Next, the local hypotheses are composed in parallel to form $\mathcal{H}$, which is submitted to the Teacher as an equivalence query $Teacher(\mathcal{H})$. If the query returns no counter-example, the algorithm returns $\mathcal{H}$ and terminates. Otherwise, the returned counter-example (σ_{ce}, b) is added to **Obs**.

Crucially, when (σ_{ce}, b) is a global counter-example, it corresponds exactly to counter-examples to the distribution (Definition 4.8), as shown in the following lemma.

- **Lemma 5.1.** *Given a global counter-example (σ, b) , let $\text{Obs}' = \text{Obs} \cup \{(\sigma, b)\}$:*
- *if $b = -$, then there is $P \in \text{Dom}(\text{Obs})^\Omega$ such that $(\sigma, P) \in \text{CED}(\Omega, \text{Obs}')$.*
 - *else, there is $\sigma_N \in \text{Dom}(\text{Obs})$, $S \subseteq \Omega$ and $P \in \text{Dom}(\text{Obs})^{\Omega \setminus S}$ such that $(\sigma_N, P \cup (\Sigma_i \mapsto \sigma)_{\Sigma_i \in S}) \in \text{CED}(\Omega, \text{Obs}')$*

Furthermore, all of the elements of $\text{CED}(\Omega, \text{Obs}')$ have the above structure.

Therefore, based on this lemma, the distribution is augmented with discrepancies for a chosen subset S of distribution counter-examples, following Corollary 4.23. This process eventually converges to provide Ω such that $\Omega \models \text{Obs}$ (Theorem 4.25). The new distribution is then optimized by making it canonical (Definition 4.26) and, if desired, increasing its connectivity. The optimization step does not affect counter-example-freeness (by Proposition 4.27 and Corollary 4.22) and may be used to reduce synchronizations, which improves performance (see Section 6). New learners are then started over the updated alphabets.¹

If, instead, the counter-example is local, its projections are forwarded to the local learners, and the next iteration starts.

► **Remark 5.2.** We leave the selection of counter-example set S as an implementation choice. While $S = \text{CED}(\Omega, \text{Obs})$ maximizes counter-example elimination, finding all counter-examples may be expensive. In our implementation, we process just one counter-example at a time, which in practice often yields a valid distribution after a single update step.

Our main theorem for this section states that the algorithm terminates and returns a correct model of the SUL.

► **Theorem 5.3.** *Let $\text{SUL} = M_1 \parallel M_2 \parallel \dots \parallel M_n$ consist of n parallel LTSs. Algorithm 1 terminates and returns $\mathcal{H}_1, \mathcal{H}_2, \dots, \mathcal{H}_k$ such that $\mathcal{H}_1 \parallel \mathcal{H}_2 \parallel \dots \parallel \mathcal{H}_k \equiv \text{SUL}$.*

► **Remark 5.4.** We make no claims regarding the number of components returned by the algorithm, nor whether $\mathcal{H}_i \equiv M_i$, for all i . This is because the final distribution may vary depending on counter-example and discrepancy choices. Moreover, different sets of

¹ In practice, learners leverage **Obs** to partially initialize their observation tables.

■ **Algorithm 1** Main learning algorithm.

Input: The alphabet of the SUL Σ_{SUL} and the teacher *Teacher*.
Init: $\text{Obs} = \emptyset$, $\Omega = \{\{a\} \mid a \in \Sigma_{SUL}\}$, $\mathbb{L} = \{\text{new learner } L_i \text{ on } \Sigma_i \mid \Sigma_i \in \Omega\}$

```

1 while True do
2   foreach  $L_i \in \mathbb{L}$  in parallel do
3     | Locally learn using Teacher until an hypothesis  $\mathcal{H}_i$  is returned
4     |  $\text{Obs} \leftarrow \text{Obs} \cup \{(\sigma, b) \mid (\sigma, b) \in \text{global membership queries of } L_i\}$ 
5    $\mathcal{H} = \parallel_{i=1}^{|\Omega|} \mathcal{H}_i$ 
6    $\text{CEX} = \text{Teacher}(\mathcal{H})$ 
7   if  $\text{CEX}$  is empty then
8     | Return  $\mathcal{H}_1, \dots, \mathcal{H}_{|\Omega|}$ 
9   else if  $\text{CEX} = (\sigma_{\text{ce}}, b)$  then
10    |  $\text{Obs} \leftarrow \text{Obs} \cup \{(\sigma_{\text{ce}}, b)\}$ 
11    | if  $(\sigma_{\text{ce}}, b)$  is global then
12      | while  $\text{CED}(\Omega, \text{Obs}) \neq \emptyset$  do
13        | Pick  $S \subseteq \text{CED}(\Omega, \text{Obs})$  non-empty
14        |  $\Omega \leftarrow \Omega \cup \Omega'$  for some  $\Omega' \in \mathcal{D}(S)$ 
15      |  $\text{Optimize}(\Omega)$ 
16      |  $\mathbb{L} \leftarrow \{\text{new learner } L_i \text{ on } \Sigma_i \mid \Sigma_i \in \Omega\}$ 
17    | else
18      | foreach  $L_i \in \mathbb{L}$  learning on  $\Sigma_i$  do
19        | Forward  $\sigma_{\text{ce}}|_{\Sigma_i}$  to  $L_i$ 

```

component LTSs can result in the same parallel composition (see [31, Remark 2]). While the composition $\mathcal{H}_1 \parallel \mathcal{H}_2 \parallel \dots \parallel \mathcal{H}_k$ may not be canonical (i.e., minimal), each $\mathcal{H}_i$ is guaranteed to be a canonical model of the local observation function Obs_{Σ_i} , as each local learner is a (slightly modified) instance of L^* , for which we have minimality guarantees.

► **Example 5.5** (Example run). We give an example run where the target SUL is the model of Figure 1. For the sake of simplicity, we focus on the global counter-examples and the subsequent distribution updates, considering only one distribution counter-example per step. Moreover, we consistently select a smallest discrepancy for each counter-example as our greedy strategy to minimize the connectivity of the resulting distribution.

We start from $\Omega_{\text{single}} = \{\{a\}, \{b\}, \{c\}, \{d\}\}$. The local alphabets initially contain only one symbol, so local learners will make membership queries about traces containing exclusively that symbol. This leads to the components depicted below.

$$T_{\{a\}} = \rightarrow (s_0) \xrightarrow{a} (s_1) \quad T_{\{b\}} = \rightarrow (s_0) \quad T_{\{c\}} = \rightarrow (s_0) \xrightarrow{c} (s_0) \quad T_{\{d\}} = \rightarrow (s_0)$$

The first global counter-example is $(ab, +)$, yielding several counter-examples to $\Omega_{\text{single}} \models \text{Obs}$, of which we consider $(b, (\{a\} \mapsto \epsilon, \{b\} \mapsto ab, \{c\} \mapsto \epsilon, \{d\} \mapsto \epsilon))$. The smallest discrepancy for this counter-example is $\{a, b\}$. We use it to update the distribution and obtain $\Omega_{ab} = \{\{a, b\}, \{c\}, \{d\}\}$ ², which models the current observations. The new component over $\{a, b\}$ is then learned locally, producing (the $\{c\}$ and $\{d\}$ components are unchanged):

² We made the distribution canonical (Definition 4.26) and removed the $\{a\}$ and $\{b\}$ components.

$$T_{\{a,b\}} = \rightarrow(s_0) \xrightarrow{a} (s_1) \xrightarrow{b} (s_2) \quad T_{\{c\}} = \rightarrow(s_0) \xrightarrow{c} (s_0) \quad T_{\{d\}} = \rightarrow(s_0)$$

The next global counter-example $(cb, +)$, leads to distribution counter-example $(b, (\{a, b\} \mapsto cb, \{c\} \mapsto \epsilon, \{d\} \mapsto \epsilon))$. Its smallest discrepancy is $\{b, c\}$ and the new distribution is $\Omega_{ab,bc} = \{\{a, b\}, \{b, c\}, \{d\}\}$. Although the counter-example has been handled, $\Omega_{ab,bc}$ does not model the observations, as $CED(\Omega_{ab,bc}, \text{Obs})$ contains $(b, (\{a, b\} \mapsto cb, \{b, c\} \mapsto ab, \{d\} \mapsto \epsilon))$. Its smallest discrepancy $\{a, b, c\}$ gives $\Omega_{abc} = \{\{a, b, c\}, \{d\}\}$, modelling the observations.

To finish our example, the next global counter-example is $(abd, +)$. The corresponding distribution counter-example is $(d, (\{a, b, c\} \mapsto \epsilon, \{d\} \mapsto abd))$. There are two smallest discrepancies for this counter-example: $\{a, d\}$ and $\{b, d\}$. Selecting $\{b, d\}$ leads to $\{\{a, b, c\}, \{b, d\}\}$, which models the target language and exactly corresponds to the decomposition of Figure 1. Selecting $\{a, d\}$ creates unnecessary connections, resulting (after some omitted steps) in either $\{\{a, b, c\}, \{a, d\}, \{b, d\}\}$ or $\{\{a, b, c\}, \{a, b, d\}\}$ as a final distribution.

Our current implementation selects either discrepancy as both are locally optimal. Finding efficient ways to explore multiple discrepancy choices for globally optimal distributions remains an open challenge for future work.

6 Experiments

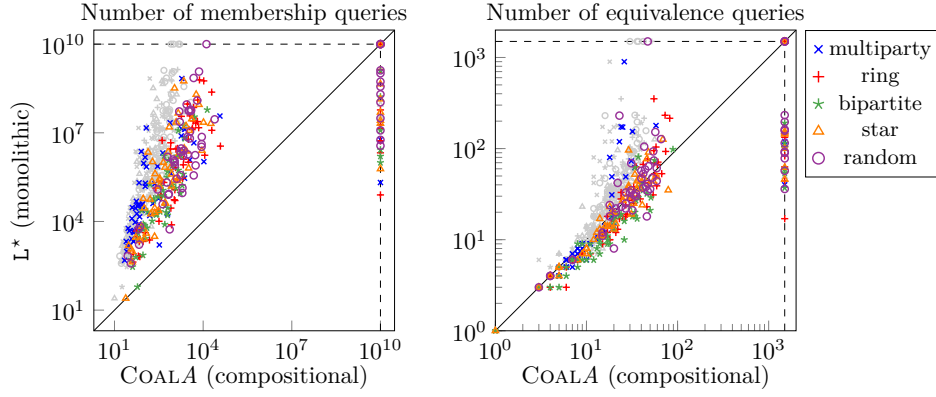
We evaluate the effectiveness of our approach in terms of savings in membership and equivalence queries. We did not expect to gain efficiency (absolute execution time) over the mature available tools, because our current implementation of local learners [31] interfaces with an external SAT solver for forming local hypotheses. To start with, we extended the tool COAL [31] into COALA (for COmpositional Automata Learner with Alphabet refinement), by adding the ability to refine the alphabets based on global counter-examples. The tool is based on LearnLib 0.18.0 [22]. As discussed in Section 5.2, the theory allows optimizing the distribution. Our implementation of this is based on greedily finding a (clique) edge cover in the hypergraph (Σ, Δ) , where Δ is the set of all discrepancies found thus far. Since the learners L_i perform better when their alphabet contains more local actions, our algorithm tries to optimise for this. We sometimes also merge components to convert synchronizations into local actions.

To validate our approach, we experiment with learning LTSs obtained from three sources:

1. 300 randomly generated LTSs from our previous work [31], varying in structure and size.
2. 328 LTSs obtained from Petri nets that are provided by the Hippo tool [40] website;³ these are often more sequential in nature than our other models.
3. Two scalable realistic models, namely *CloudOpsManagement* from the 2019 Model Checking Contest [3] and a *producers/consumers* model [41, Fig. 8].

Using a machine with four Intel Xeon 6136 processors and 3TB of RAM running Ubuntu 20.04, we apply each of three approaches: (i) our black-box compositional approach (COALA), (ii) compositional learning with given alphabets (COAL) [31], and (iii) monolithic learning with L^* (as implemented in LearnLib). COAL can be viewed as an idealized (best-case) baseline where the knowledge of the system decomposition is already available. Each run has a time-out of 30 minutes. We record the number of membership and equivalence queries posed to the Teacher, which we assume answers queries in constant time. This eliminates

³ <https://hippo.uz.zgora.pl/>



■ **Figure 5** Performance of L^* and compositional learning on 300 randomly generated models. Dashed lines indicate time-outs. Results obtained with COALA are in gray.

variations in runtime caused by the Teacher, and ensures that local negative counter-examples (see Section 5.1) are always eventually corrected. Resolving this limitation is orthogonal to the current work and left for future research. A complete replication package is at [18].

Random models

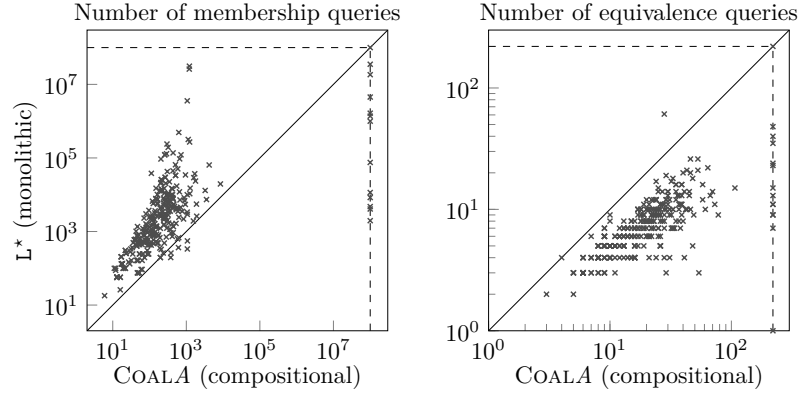
Figure 5 shows the results on our random models. The colors indicate various communication structures (see [31] for details). As a reference, the results obtained with COALA are given in gray. We observe that COALA requires significantly fewer membership queries than L^* (note the logarithmic scale) and is closer to the theoretical optimum of COALA; the results show 5-6 orders of magnitude of improvement in a large number of concurrent systems. The number of equivalence queries required by COALA is typically slightly higher, but results of larger instances suggest that COALA scales better than its monolithic counterpart. The data shows that also in the case of equivalence queries, it is not uncommon to gain an order of magnitude of saving by using our approach. We note that COALA timeouts occur not due to a high query count, but because local learners use SAT solving to construct minimal hypotheses from observation tables that contain unknowns. This process can be computationally expensive.

Petri Net models

The results of learning the Hippo models are given in Figure 6. Here we are not able to run COALA, since the component alphabets are not known. COALA does not perform as well as on random models, in particular it requires more equivalence queries than monolithic L^* . This is explained by the fact that these Petri nets contain mostly sequential behavior, *i.e.*, the language roughly has the shape $\mathcal{L}_1 \cdot a \cdot \mathcal{L}_2$ for some languages $\mathcal{L}_1 \subseteq \Sigma_1^*$ and $\mathcal{L}_2 \subseteq \Sigma_2^*$. Even though our learner is able to find the decomposition $\{\Sigma_1 \cup \{a\}, \Sigma_2 \cup \{a\}\}$, we do not gain much due to the absence of concurrent behavior. In the Hippo benchmark set, COALA typically finds between two and nine components.

Realistic models

Finally, Table 1 shows the results of learning two scalable models with relevant parameters indicated. No timeouts are reported in the table because all systems were successfully learned within the time limit. COALA scales well as the SUL size increases, requiring roughly a



■ **Figure 6** Performance of L^* and compositional learning on Hippo models. Dashed lines indicate time-outs or out-of-memory.

constant factor more queries than COAL for both CloudOps and producers/consumers. We remark that practically all runtime of COALA and COAL is spent in the local learners: they require expensive SAT queries to construct a local hypothesis. Improving the implementation of local learners would decrease these times significantly; we stress that this is orthogonal to the goal of the current work. Furthermore, in any practical scenario, the processing of queries by the Teacher forms the bottleneck and L^* would be much slower than COALA.

■ **Table 1** Performance of COALA and L^* for realistic composite systems. Reported runtimes are in seconds. The number of refinement iterations is listed under “it.” and the number of components found under “com”.

model	states	COALA					COAL				L^*		
		time	memQ	eqQ	it.	com	time	memQ	eqQ	com	time	memQ	eqQ
CloudOps $W=1, C=1, N=3$	690	225.21	7 686	106	67	9	1.27	880	24	5	3.29	2 740 128	88
CloudOps $W=1, C=1, N=4$	1 932	235.97	9 521	115	74	10	1.93	923	26	6	29.62	22 252 120	216
CloudOps $W=2, C=1, N=3$	3 858	232.87	25 968	98	63	8	357.51	9 812	29	5	12.69	12 574 560	99
CloudOps $W=2, C=1, N=4$	10 824	708.95	33 616	111	72	10	555.82	9 532	30	6	143.28	91 178 900	227
ProdCons $K=3, P=2, C=2$	1 664	3.84	685	26	23	5	1.40	301	13	5	3.05	2 141 165	43
ProdCons $K=5, P=1, C=1$	170	1.03	451	22	14	3	0.88	118	11	3	0.40	160 126	30
ProdCons $K=5, P=2, C=1$	662	1.59	379	24	17	4	0.93	158	13	4	2.79	2 523 625	91
ProdCons $K=5, P=2, C=2$	2 240	5.53	697	28	25	5	2.35	282	14	5	6.40	6 984 705	93
ProdCons $K=5, P=3, C=2$	8 750	17.82	1 305	31	30	6	6.33	604	15	6	72.87	60 186 235	187
ProdCons $K=5, P=3, C=3$	30 344	73.69	2 454	34	37	7	32.59	1 269	17	7	321.54	222 567 729	193
ProdCons $K=7, P=2, C=2$	2 816	5.26	715	29	25	5	2.20	307	15	5	16.92	15 792 997	135

7 Conclusion

We presented a novel active learning algorithm that automatically discovers component decompositions using only global observations. Unlike previous approaches, our technique handles a general synchronization scheme common in automata theory and process calculi, without any prior knowledge of component structure. We developed a theory of alphabet distributions and their relation to observations, formally characterizing counter-examples that indicate inconsistencies between distributions and observations, and providing systematic methods to resolve them. The algorithm spawns a local learner for each component, dynamically refining the distribution of the alphabet based on counter-examples. Our

COALA implementation dramatically reduces membership queries and achieves better query scalability than monolithic learning on highly concurrent systems. Future work will focus on:

- developing a theory of counter-example and discrepancy selection to characterize optimal distributions;
- extending our approach to apartness-based learning [37] and more expressive formalisms such as register automata [9] or timed automata [39];
- leveraging our compositional techniques to analyze AI-generated code [32]; and
- improving the efficiency of the local learners [24].

References

- 1 Fides Aarts, Joeri de Ruiter, and Erik Poll. Formal models of bank cards for free. In *ICST*, pages 461–468, 2013. doi:10.1109/ICSTW.2013.60.
- 2 Fides Aarts, Julien Schmaltz, and Frits W. Vaandrager. Inference and abstraction of the biometric passport. In *ISO LA*, volume 6415, pages 673–686, 2010. doi:10.1007/978-3-642-16558-0_54.
- 3 Elvio Amparore et al. Presentation of the 9th Edition of the Model Checking Contest. In *TACAS*, volume 11429, pages 50–68, 2019. doi:10.1007/978-3-030-17502-3_4.
- 4 Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987. doi:10.1016/0890-5401(87)90052-6.
- 5 Franco Barbanera, Mariangiola Dezani-Ciancaglini, Ivan Lanese, and Emilio Tuosto. Composition and decomposition of multiparty sessions. *Journal of Logical and Algebraic Methods in Programming*, 119:100620, 2021. doi:10.1016/J.JLAMP.2020.100620.
- 6 Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. Learning Communicating Automata from MSCs. *IEEE Transactions on Software Engineering*, 36(3):390–408, 2010. doi:10.1109/TSE.2009.89.
- 7 Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472, 2005. doi:10.1007/B137241.
- 8 Ilaria Castellani, Madhavan Mukund, and P. S. Thiagarajan. Synthesizing distributed transition systems from global specification. In *FSTTCS*, volume 1738, pages 219–231, 1999. doi:10.1007/3-540-46691-6_17.
- 9 Simon Dierl, Paul Fiterau-Brostean, Falk Howar, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. Scalable tree-based register automata learning. In *TACAS*, volume 14571, pages 87–108, 2024. doi:10.1007/978-3-031-57249-4_5.
- 10 Andrzej Ehrenfeucht and Grzegorz Rozenberg. Partial (set) 2-structures. part II: state spaces of concurrent systems. *Acta Informatica*, 27(4):343–368, 1990. doi:10.1007/BF00264612.
- 11 Tiago Ferreira, Harrison Brewton, Loris D’Antoni, and Alexandra Silva. Prognosis: closed-box analysis of network protocol implementations. In *SIGCOMM*, pages 762–774, 2021. doi:10.1145/3452296.3472938.
- 12 Paul Fiterau-Brostean, Ramon Janssen, and Frits W. Vaandrager. Combining Model Learning and Model Checking to Analyze TCP Implementations. In *CAV*, volume 9780, pages 454–471, 2016. doi:10.1007/978-3-319-41540-6_25.
- 13 Paul Fiterau-Brostean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits W. Vaandrager, and Patrick Verleg. Model learning and model checking of SSH implementations. In *SPIN*, pages 142–151, 2017. doi:10.1145/3092282.3092289.
- 14 Markus Frohme and Bernhard Steffen. Compositional learning of mutually recursive procedural systems. *International Journal on Software Tools for Technology Transfer*, 23(4):521–543, 2021. doi:10.1007/s10009-021-00634-y.

- 15 Dominik Fuchß, Haoyu Liu, Tobias Hey, Jan Keim, and Anne Koziolk. Enabling architecture traceability by llm-based architecture component name extraction. In *ICSA*, pages 1–12, 2025. doi:10.1109/ICSA65012.2025.00011.
- 16 E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978. doi:10.1016/S0019-9958(78)90562-4.
- 17 Roberto Guanciale and Emilio Tuosto. Realisability of pomsets via communicating automata. In *ICE*, volume 279, pages 37–51, 2018. doi:10.4204/EPTCS.279.6.
- 18 Léo Henry, Mohammad Mousavi, Thomas Neele, and Matteo Sammartino. Replication package for the paper “Compositional Active Learning of Synchronous Systems through Automated Alphabet Refinement”, April 2025. doi:10.5281/zenodo.15170685.
- 19 C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 2004.
- 20 Falk Howar and Bernhard Steffen. Active automata learning in practice - an annotated bibliography of the years 2011 to 2016. In *Machine Learning for Dynamic Software Analysis*, volume 11026, pages 123–148, 2018. doi:10.1007/978-3-319-96562-8_5.
- 21 Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *RV*, volume 8734, pages 307–322, 2014. doi:10.1007/978-3-319-11164-3_26.
- 22 Malte Isberner, Falk Howar, and Bernhard Steffen. The Open-Source LearnLib: A Framework for Active Automata Learning. In *CAV*, volume 9206, pages 487–495, 2015. doi:10.1007/978-3-319-21690-4_32.
- 23 Rainer Koschke. *Architecture Reconstruction*. Springer-Verlag, 2009. doi:10.1007/978-3-540-95888-8.
- 24 Loes Kruger, Sebastian Junges, and Jurriaan Rot. Small test suites for active automata learning. In *TACAS*, volume 14571, pages 109–129, 2024. doi:10.1007/978-3-031-57249-4_6.
- 25 Faezeh Labbaf, Jan Friso Groote, Hossein Hojjat, and Mohammad Reza Mousavi. Compositional learning for interleaving parallel automata. In *FoSSaCS*, volume 13992, pages 413–435, 2023. doi:10.1007/978-3-031-30829-1_20.
- 26 Bas Luttik. Unique parallel decomposition in branching and weak bisimulation semantics. *Theoretical Computer Science*, 612:29–44, 2016. doi:10.1016/j.tcs.2015.10.013.
- 27 Aryan Bastany Mahboubeh Samadi and Hossein Hojjat. Compositional learning for synchronous parallel automata. In *FASE*, volume 15693, pages 101–121, 2025. doi:10.1007/978-3-031-90900-9_6.
- 28 Stefan Marksteiner, Marjan Sirjani, and Mikael Sjödín. Automated passport control: Mining and checking models of machine readable travel documents. In *ARES*, 2024. doi:10.1145/3664476.3670454.
- 29 Joshua Moerman. Learning product automata. In *ICGI*, volume 93, pages 54–66, 2018. URL: <http://proceedings.mlr.press/v93/moerman19a.html>.
- 30 Madhavan Mukund. From global specifications to distributed implementations. In *Synthesis and Control of Discrete Event Systems*, pages 19–35, 2002. doi:10.1007/978-1-4757-6656-1_2.
- 31 Thomas Neele and Matteo Sammartino. Compositional automata learning of synchronous systems. In *FASE*, volume 13991, pages 47–66, 2023. doi:10.1007/978-3-031-30826-0_3.
- 32 Marc North, Amir Atapour-Abarghouei, and Nelly Bencomo. Code gradients: Towards automated traceability of llm-generated code. In *RE*, pages 321–329, 2024. doi:10.1109/RE59067.2024.00038.
- 33 Mathijs Schuts, Jozef Hooman, and Frits W. Vaandrager. Refactoring of legacy software using model learning and equivalence checking: An industrial experience report. In *IFM*, volume 9681, pages 311–325, 2016. doi:10.1007/978-3-319-33693-0_20.
- 34 Maurice H. ter Beek, Rolf Hennicker, and José Proença. Team automata: Overview and roadmap. In *COORDINATION*, volume 14676, pages 161–198, 2024. doi:10.1007/978-3-031-62697-5_10.
- 35 Maurice H. ter Beek, Rolf Hennicker, and José Proença. Overview and roadmap of team automata. *CoRR*, abs/2501.13589, 2025. doi:10.48550/arXiv.2501.13589.

- 36 Frits W. Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, 2017. doi:10.1145/2967606.
- 37 Frits W. Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A new approach for active automata learning based on apartness. In *TACAS*, volume 13243, pages 223–243, 2022. doi:10.1007/978-3-030-99524-9_12.
- 38 Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. CacheQuery: Learning Replacement Policies from Hardware Caches. In *PLDI*, pages 519–532, 2020. doi:10.1145/3385412.3386008.
- 39 Masaki Waga. Active learning of deterministic timed automata with myhill-nerode style characterization. In *CAV*, pages 3–26, 2023. doi:10.1007/978-3-031-37706-8_1.
- 40 Remigiusz Wiśniewski, Grzegorz Bazydło, Marcin Wojnakowski, and Mateusz Popławski. Hippo-CPS: A Tool for Verification and Analysis of Petri Net-Based Cyber-Physical Systems. In *Petri Nets 2023*, volume 13929, pages 191–204, 2023. doi:10.1007/978-3-031-33620-1_10.
- 41 W.M. Zuberek. Petri net models of process synchronization mechanisms. In *SMC*, volume 1, pages 841–847, 1999. doi:10.1109/ICSMC.1999.814201.

A

 Proofs

A.1 Section 4

► **Proposition 4.6.** $\Omega \models \text{Obs}$ if and only if for all traces $\sigma \in \text{Dom}(\text{Obs})$ it holds that $\text{Obs}(\sigma) = \bigwedge_{\Sigma_i \in \Omega} \text{Obs}_{\Sigma_i}(\sigma_{\upharpoonright \Sigma_i})$.

Proof. We make the proof by double implication.

Suppose $\Omega \models \text{Obs}$. By definition this means that there is $\mathcal{L} \models \text{Obs}$ such that $\mathcal{L} \models \text{Obs}$. Since $\mathcal{L}$ is a product language, Lemma 4.4 yields that $\mathcal{L} = \{\sigma \mid \forall 1 \leq i \leq n. \sigma_{\upharpoonright \Sigma_i} \in \mathcal{L}_i\}$ where $\mathcal{L}_i = \{\sigma_{\upharpoonright \Sigma_i} \mid \sigma \in \mathcal{L}\}$ for all i .

Now for any $\sigma \in \text{Dom}(\text{Obs})$, we wish to show that

$$\text{Obs}(\sigma) = + \text{ iff } \bigwedge_{\Sigma_i \in \Omega} \text{Obs}_{\Sigma_i}(\sigma_{\upharpoonright \Sigma_i}) = +.$$

We separate the two implications.

If $\text{Obs}(\sigma) = +$, then $\text{Obs}_{\Sigma_i}(\sigma_{\upharpoonright \Sigma_i}) = +$ for all i by definition of local observation functions and we have our result.

If $\bigwedge_{\Sigma_i \in \Omega} \text{Obs}_{\Sigma_i}(\sigma_{\upharpoonright \Sigma_i}) = +$, it means that for any i , $\text{Obs}_{\Sigma_i}(\sigma_{\upharpoonright \Sigma_i}) = +$. By definition of local observations, for all i there is $\sigma'_i \in \text{Dom}(\text{Obs})$ such that $\text{Obs}(\sigma'_i) = +$ and $\sigma_{\upharpoonright \Sigma_i} = \sigma'_i|_{\Sigma_i}$. As $\mathcal{L} \models \text{Obs}$, $\sigma'_i \in \mathcal{L}$.

This implies that $\forall i. \exists \sigma'_i \in \mathcal{L} \wedge \sigma_{\upharpoonright \Sigma_i} = \sigma'_i|_{\Sigma_i}$. Hence for all i , $\sigma_{\upharpoonright \Sigma_i} \in \mathcal{L}_i$ by definition of $\mathcal{L}_i$. It follows by definition of $\mathcal{L}$ that $\sigma \in \mathcal{L}$. As $\mathcal{L} \models \text{Obs}$, we have $\text{Obs}(\sigma) = +$.

Conversely, suppose that $\forall \sigma \in \text{Dom}(\text{Obs}), \text{Obs}(\sigma) = \bigwedge_{\Sigma_i \in \Omega} \text{Obs}_{\Sigma_i}(\sigma_{\upharpoonright \Sigma_i})$.

Consider the language $\mathcal{L} = \{\sigma \in \text{Dom}(\text{Obs}) \mid \text{Obs}(\sigma) = +\}$ and, for all i , let $\mathcal{L}_i = \{\sigma \in \text{Dom}(\text{Obs}_{\Sigma_i}) \mid \text{Obs}_{\Sigma_i}(\sigma) = +\}$. Clearly, we have that $\mathcal{L} \models \text{Obs}$. Furthermore, by assumption, $\mathcal{L} = \{\sigma \mid \forall 1 \leq i \leq n, \sigma_{\upharpoonright \Sigma_i} \in \mathcal{L}_i\}$. Hence $\Omega \models \mathcal{L}$ by definition. $\Omega \models \text{Obs}$ follows. ◀

► **Corollary 4.10.** $\Omega \models \text{Obs} \iff \text{CED}(\Omega, \text{Obs}) = \emptyset$.

Proof. We make the proof by double implication.

When $\text{CED}(\Omega, \text{Obs}) \neq \emptyset$ there is a counter-example (σ_N, P) to $\Omega \models \text{Obs}$. In particular $\text{Obs}(\sigma_N) = -$ and for all $\Sigma_i \in \Omega$ $\text{Obs}_{\Sigma_i}(\sigma_N|_{\Sigma_i}) = \text{Obs}_{\Sigma_i}(P(\Sigma_i)|_{\Sigma_i}) = +$ by definition of a counter-example and local observation functions. Hence by Proposition 4.6, $\Omega \not\models \text{Obs}$.

Conversely, when $CED(\Omega, \text{Obs}) = \emptyset$ then for any $\sigma \in \text{Dom}(\text{Obs})$ such that $\text{Obs}(\sigma) = -$ there is some i such that for any $\sigma' \in \text{Dom}(\text{Obs})$ verifying $\text{Obs}_{\Sigma_i}(\sigma' \upharpoonright_{\Sigma_i}) = \text{Obs}_{\Sigma_i}(\sigma \upharpoonright_{\Sigma_i})$, $\text{Obs}(\sigma') = -$.

It follows that for any $\sigma \in \text{Dom}(\text{Obs})$ such that $\text{Obs}(\sigma) = -$, $\bigwedge_{1 \leq i \leq n} \text{Obs}_{\Sigma_i}(\sigma \upharpoonright_{\Sigma_i}) = -$. As by definition of local observations, for any $\sigma \in \text{Dom}(\text{Obs})$ such that $\text{Obs}(\sigma) = +$, $\bigwedge_{1 \leq i \leq n} \text{Obs}_{\Sigma_i}(\sigma \upharpoonright_{\Sigma_i}) = +$ we have $\Omega \models \text{Obs}$ by Proposition 4.6. $\blacktriangleleft$

► **Proposition 4.16.** *Suppose there exists $(\sigma_N, P) \in CED(\Omega, \text{Obs})$. For each discrepancy $\delta \in \mathcal{D}(\sigma_N, P)$, $(\sigma_N, P) \notin CED(\Omega \cup \{\delta\}, \text{Obs})$. Conversely, for any distribution Ω' of Σ where $\delta \not\subseteq \Sigma_i$, for all $\delta \in \mathcal{D}(\sigma_N, P)$ and all $\Sigma_i \in \Omega'$, (σ_N, P) is a counter-example to $\Omega' \models \text{Obs}$.*

Proof. Consider a discrepancy δ . We first prove that introducing δ into Ω is sufficient to remove the counter-example. As Obs is not changed by the change from Ω to Ω' we have to prove that there is no $\sigma_{\Sigma_i} \in \text{img}(P)$ such that $\sigma_N \upharpoonright_{\delta} = \sigma_{\Sigma_i} \upharpoonright_{\delta}$.

Fix $\sigma_{\Sigma_i} \in \text{img}(P)$.

- If $\mathcal{D}_m^{\Sigma_i}(\sigma_N, P) \cap \delta \neq \emptyset$ then there is $a \in \delta$ that occurs in different multiplicities in σ_N and σ_{Σ_i} . Hence, a also occurs in different multiplicities in $\sigma_N \upharpoonright_{\delta}$ and $\sigma_{\Sigma_i} \upharpoonright_{\delta}$. Hence, $\sigma_N \upharpoonright_{\delta} \neq \sigma_{\Sigma_i} \upharpoonright_{\delta}$.
- Otherwise, following the notations of Definition 4.12, $\sigma'_N = \pi(\sigma_{\Sigma_i} \upharpoonright_{\theta})$ and for all $j < k$ such that $\sigma'_N[j] = \sigma'_N[k]$ it holds that $\pi(j) < \pi(k)$. By preserving the order of equal symbols, we maintain that $\sigma[j] = a$ is the l -th occurrence of a in σ'_N iff $\sigma'[\pi(j)] = a$ is the l -th occurrence of a in σ_i .

Now $\{a, b\} = \{\sigma'_N[j], \sigma'_N[k]\} \subseteq \delta$ by definition of a discrepancy and $\{a, b\}$ is created from a so-called *inversion* in π : a pair (j, k) such that $j < k$ and $\pi(j) > \pi(k)$. By our assumption on π , we know that $a \neq b$ and furthermore that in σ' at least one more copy of b precedes this a (namely the b at $\sigma'[\pi(k)]$). As a result of this and the fact that $\{a, b\} \subseteq \delta$, we have $\sigma'_N \upharpoonright_{\delta} \neq \sigma_{\Sigma_i} \upharpoonright_{\theta \upharpoonright_{\delta}}$. It follows directly that $\sigma_N \upharpoonright_{\delta} \neq \sigma_{\Sigma_i} \upharpoonright_{\delta}$.

We now prove that it is necessary. Consider Ω' defined as above and fix $\Sigma_j \in \Omega'$. In the following, we show that there is $\sigma \in \text{img}(P)$ such that $\sigma_N \upharpoonright_{\Sigma_j} = \sigma \upharpoonright_{\Sigma_j}$.

For this, as there is no discrepancy $\delta \in \mathcal{D}(\sigma_N, P)$ subset to Σ_j we know that there is at least one $\Sigma_i \in \Omega$ such that no order nor multiplicity discrepancy for $\sigma \upharpoonright_{\Sigma_i}$ is a subset of Σ_j . Because of this, $\Sigma^m(\sigma_N) \Delta \Sigma^m(P(\Sigma_i)) \cap \Sigma_j = \emptyset$. Hence $P(\Sigma_i) \upharpoonright_{\Sigma_j}$ and $\sigma_N \upharpoonright_{\Sigma_j}$ are equal up to permutation. Consider any permutation π' such that $\sigma'_N = \pi'(\sigma_i)$ (following again the notations of Definition 4.12). Without loss of generality, we do not permute the position of equal symbols, which fixes a unique permutation $\pi' = \pi$. For any $\{\{\sigma'_N[j], \sigma'_N[k]\} \mid j < k \wedge \pi(j) > \pi(k)\}$, we know (as these are elements of $\mathcal{D}_o^{\Sigma_i}(\sigma_N, P)$) that we don't have $\{\sigma'_N[j], \sigma'_N[k]\} \subseteq \Sigma_j$. It follows that the restriction of π to the indices corresponding to symbols in Σ_j is non-decreasing and thus the identity. It follows that $\sigma_N \upharpoonright_{\Sigma_j} = P(\Sigma_i) \upharpoonright_{\Sigma_j}$. $\blacktriangleleft$

► **Proposition 4.21.** *Consider two distributions Ω and Ω' of Σ . We have $\Omega \preceq \Omega' \Rightarrow CED(\Omega, \text{Obs}) \preceq CED(\Omega', \text{Obs})$.*

Proof. We first prove that $CED(\Omega, \text{Obs}) \preceq CED(\Omega', \text{Obs})$. Suppose $\Omega \preceq \Omega'$. If $CED(\Omega', \text{Obs}) = \emptyset$ then we have our result. Else, take $(\sigma_N, P) \in CED(\Omega', \text{Obs})$.

We know that for each $\Sigma_i \in \Omega$, there is $\Sigma_j^i \in \Omega'$ such that $\Sigma_i \subseteq \Sigma_j^i$. Fix such a Σ_j^i . We have that $\sigma_N \upharpoonright_{\Sigma_i} = P(\Sigma_j^i) \upharpoonright_{\Sigma_i}$ as $\Sigma_i \subseteq \Sigma_j^i$ and $(\sigma_N, P) \in CED(\Omega', \text{Obs})$.

By doing this for all $\Sigma_i \in \Omega$, we get $(\sigma_N, (P(\Sigma_j^i))_{\Sigma_i \in \Omega}) \in CED(\Omega, \text{Obs})$ and by definition $(\sigma_N, (P(\Sigma_j^i))_{\Sigma_i \in \Omega}) \subseteq (\sigma_N, P)$. $\blacktriangleleft$

► **Corollary 4.23.** *Suppose that $CED(\Omega, \text{Obs}) \neq \emptyset$. For $(\sigma_N, P) \in CED(\Omega, \text{Obs})$, we pick a discrepancy $\delta_{(\sigma_N, P)} \in \mathcal{D}(\sigma_N, P)$. For any non-empty subset S of $CED(\Omega, \text{Obs})$, let $\Omega' = \Omega \cup \{\delta_{ce} \mid ce \in S\}$. Then $\Omega \prec \Omega'$ and $CED(\Omega, \text{Obs}) \prec CED(\Omega', \text{Obs})$.*

Proof. $\Omega \prec \Omega'$ follows directly from the definition of Ω' : all elements of Ω are preserved. Furthermore, for $ce \in S$, for all $\Sigma_i \in \Omega$ we have $\delta_{ce} \not\subseteq \Sigma_i \in \Omega$ as $\sigma_N \upharpoonright \delta \neq \sigma_{\Sigma_i} \upharpoonright \delta$ and $\sigma_N \upharpoonright \Sigma_i = \sigma_{\Sigma_i} \upharpoonright \Sigma_i$. Hence $\Omega' \not\prec \Omega$ and $\Omega \prec \Omega'$. From Proposition 4.21 we get that $CED(\Omega, \text{Obs}) \prec CED(\Omega', \text{Obs})$. Furthermore, we know by Proposition 4.16 that for a fixed $ce \in S$, $CED(\Omega \cup D_{ce}, \text{Obs})$ does not contain ce , ensuring that $CED(\Omega, \text{Obs}) \prec CED(\Omega \cup \delta_{ce}, \text{Obs})$. As furthermore $CED(\Omega \cup \delta_{ce}, \text{Obs}) \prec CED(\Omega', \text{Obs})$ by construction, we have that $CED(\Omega, \text{Obs}) \prec CED(\Omega', \text{Obs})$. ◀

► **Theorem 4.25.** *Suppose $\Omega \not\models \text{Obs}$. The above process converges to a distribution $\Omega' \models \text{Obs}$ such that $\Omega \prec \Omega'$ after finitely many of steps. For choosing $S = CED(\Omega, \text{Obs})$ at each step, the number of steps is bounded by $|\mathcal{P}(\Sigma)|$.*

Proof. In case of convergence, $\Omega \prec \Omega'$ is ensured directly by each step of the process by Corollary 4.23. We now prove that convergence occurs in a finite amount of steps. Each step eliminates all counter-examples that are part of the non-empty set S chosen, as proven in Proposition 4.16 and Corollary 4.23. As $CED(\Omega, \text{Obs}) \prec CED(\Omega', \text{Obs})$ the only possible new counter-examples are ones that contain strictly counter-examples of $CED(\Omega, \text{Obs})$. It follows that iterating this process makes strict progress in the following sense: either counter-examples are eliminated or they are replaced by ones with a strictly larger positive image. Because the positive image size of counter-examples is bounded by the size of the distribution, itself bounded by $|\mathcal{P}(\Sigma)|$, we know that this converges in a finite number of steps.

If $S = CED(\Omega, \text{Obs})$ at each step, we furthermore have that the size of the smallest positive image of a counter-example increases by at least one at each step, which bounds the number of steps by $|\mathcal{P}(\Sigma)|$. ◀

► **Proposition 4.27.** $CED(\llbracket \Omega \rrbracket_{\prec}, \text{Obs}) = \emptyset \Leftrightarrow CED(\Omega, \text{Obs}) = \emptyset$

Proof. We make the proof for a fixed $\Sigma_i \in \text{Sub}$, the result following by induction. By definition $\Omega' \prec \Omega$, which entails that $CED(\Omega, \text{Obs}) \prec CED(\Omega', \text{Obs})$. From there it follows that $CED(\Omega, \text{Obs}) = \emptyset \Rightarrow CED(\Omega', \text{Obs}) = \emptyset$. To get the other one, we reason by contrapositive. Suppose $CED(\Omega, \text{Obs}) \neq \emptyset$ and consider (σ_N, P) in it. Then clearly, taking $P' = (P(\Sigma_j))_{j \neq i}$, $(\sigma_N, P') \in CED(\Omega', \text{Obs})$ and we have our result. ◀

A.2 Section 5

► **Lemma 5.1.** *Given a global counter-example (σ, b) , let $\text{Obs}' = \text{Obs} \cup \{(\sigma, b)\}$:*

- *if $b = -$, then there is $P \in \text{Dom}(\text{Obs})^\Omega$ such that $(\sigma, P) \in CED(\Omega, \text{Obs}')$.*
- *else, there is $\sigma_N \in \text{Dom}(\text{Obs})$, $S \subseteq \Omega$ and $P \in \text{Dom}(\text{Obs})^{\Omega \setminus S}$ such that $(\sigma_N, P \cup (\Sigma_i \mapsto \sigma)_{\Sigma_i \in S}) \in CED(\Omega, \text{Obs}')$*

Furthermore, all of the elements of $CED(\Omega, \text{Obs}')$ have the above structure.

Proof. We know that $\Omega \models \text{Obs}$ and $\Omega \not\models \text{Obs}'$ since the counter-example is global. From Corollary 4.10 we get that $CED(\Omega, \text{Obs}) = \emptyset$ and $CED(\Omega, \text{Obs}') \neq \emptyset$.

Consider $(\sigma_N, P) \in CED(\Omega, \text{Obs}')$. Since it is not a counter-example to $\Omega \models \text{Obs}$, σ must appear in it. If $b = -$ we must have that $\sigma_N = \sigma$ by definition of counter-examples to a distribution. Similarly, if $b = +$ then $\sigma \in \text{img}(P)$. Observing that $\text{Dom}(\text{Obs}') = \text{Dom}(\text{Obs}) \cup \{\sigma\}$, we obtain our result. ◀

► **Theorem 5.3.** *Let $SUL = M_1 \parallel M_2 \parallel \dots \parallel M_n$ consist of n parallel LTSs. Algorithm 1 terminates and returns $\mathcal{H}_1, \mathcal{H}_2, \dots, \mathcal{H}_k$ such that $\mathcal{H}_1 \parallel \mathcal{H}_2 \parallel \dots \parallel \mathcal{H}_k \equiv SUL$.*

Proof. The correctness of the returned hypothesis is guaranteed by the Teacher.

The algorithm's termination is established by showing that the “while True” loop cannot execute indefinitely. This is because the two types of counter-examples can only occur finitely many times:

- For local counter-examples, when the distribution Ω is fixed, all the learners eventually converge to a hypothesis after encountering finitely many local counter-examples, as shown in [31, Theorem 2].
- For global counter-examples, each global counter-example leads to an updated distribution Ω' such that $\Omega \prec \Omega'$ (Theorem 4.25). This update can only happen finitely many times, as the top element of $\prec$ is $\{\Sigma_{SUL}\}$. ◀