Learning Broadcast Protocols with LeoParDS

Noa Izsak[1][0009−0004−1333−2490], Dana Fisman[1][0000−0002−6015−4170], and Swen Jacobs[2][0000−0002−9051−4050]

[1] Ben Gurion University, Beer-Sheva, Israel
[2] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

Abstract. **LeoParDS** is a new tool for learning broadcast protocols (BPs) from a set of positive and negative example traces. It is the first tool that enables learning of a distributed computational model in a *parameterized* setting, i.e., with a parametric number of processes running the BP concurrently. We describe the tool along a running example, discuss some implementation details, and present experimental results on randomly generated BPs.

Keywords: learning computational models · broadcast protocols · parameterized verification · concurrent systems.

1 Introduction

We present **LeoParDS**[3], an automatic tool for passive learning of broadcast protocols from example traces. Broadcast protocols are one of the most powerful computational models for which some parameterized verification problems are still decidable, and are strictly more expressive than protocols using communication primitives such as pairwise rendezvous [33] or disjunctive guards [23].

A **Broadcast Protocol**, in short **BP**, is a tuple (S, s_0, L, R) consisting of a finite set of states S with an initial state $s_0 \in S$, a set of labels L and a transition relation $R \subseteq S \times L \times S$, where $L = \{a!!, a?? \mid a \in A\}$ for some set of actions A (see Fig.1). A transition labeled with $a!!$ is a broadcast *sending transition* (for action a), and a transition labeled with $a??$ is a broadcast *receiving transition* (for action a), also called a *response*. For each action $a \in A$, there is a unique state for the outgoing sending transition, and there must be exactly one outgoing response from every state. That is, when a process p is in a certain state s, and action a was broadcast by some other process q, the process p must respond by taking the unique $a??$ transition from s.

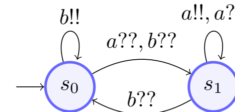


Fig. 1: BP $\mathcal{B}_{toy}$ with 2 states

Given a BP $\mathcal{B} = (S, s_0, L, R)$ we use $\mathcal{B}^n$ to denote the system composed of n indistinguishable (i.e., identical) processes that execute $\mathcal{B}$ in parallel. Let $[n]$ denote the set $\{0, 1, \ldots, n\}$. A *configuration* of $\mathcal{B}^n$ is a function $\mathbf{q} : S \to [n]$, assigning to each state $s \in S$ the number of processes that are currently in local state s. The *initial configuration* $\mathbf{q}_0$ is the configuration with $\mathbf{q}_0(s_0) = n$ and $\mathbf{q}_0(s) = 0$ for all $s \neq s_0$. In a *global transition*, all processes make a move: One process takes a sending transition (labeled $a!!$),

[3] **LeoParDS** can be accessed by GitHub or Zenodo (DOI 10.5281/zenodo.10968037)

modeling that it broadcasts the value a to all the other processes in the system. Simultaneously, all of the **other** processes take the receiving transition (labeled a??) from their current state. Note that in $\mathcal{B}^1$, namely when the system is composed of a single process, this process takes only sending transitions, and there are no responses since there are no other processes. For those less familiar with broadcast protocols, more definitions, examples, and intuitions are given in [29].

One of the important definitions is the *language* of a broadcast protocol. With a particular $n \in \mathbb{N}$ we use $\mathcal{L}(\mathcal{B}^n)$ to denote the set of words that are *feasible* in $\mathcal{B}^n$. For instance, consider the BP $\mathcal{B}_{toy}$ of Fig.1. Then bbb is feasible in $\mathcal{B}^1_{toy}$, and ba is feasible in $\mathcal{B}^2_{toy}$ but not in $\mathcal{B}^1_{toy}$ (since the single process remains in s_0 after taking b and a cannot be broadcast from s_0). A BP $\mathcal{B}$ defines an infinite family of systems $\{\mathcal{B}^n\}_{n\in\mathbb{N}}$, often referred to as a *parameterized system*. We use $\mathcal{L}(\mathcal{B})$ for the set of words feasible by at least one member $\mathcal{B}^n$ of this family, namely $\mathcal{L}(\mathcal{B}) = \bigcup_{n\in\mathbb{N}} \mathcal{L}(\mathcal{B}^n)$.

LeoParDS deal with *fine BP*s [29]. A BP is *fine* if it does not have hidden states[4] and there exists a cutoff k, such that the language of $\mathcal{B}^k$ is equal to the language of $\mathcal{B}^n$ for any $n > k$.

The learning problem we are interested in is inferring a broadcast protocol $\mathcal{B}'$ from a sample consisting of (execution) traces of members of a family $\{\mathcal{B}^n\}_{n\in\mathbb{N}}$ for some unknown BP $\mathcal{B}$ such that $\mathcal{L}(\mathcal{B}')$ is consistent with the sample. In addition, if the sample subsumes a *characteristic sample* for $\mathcal{B}$ then we require that $\mathcal{L}(\mathcal{B}') = \mathcal{L}(\mathcal{B})$. A *characteristic sample*, in short CS, is an important notion from the literature on passive learning of automata. Loosely speaking, a characteristic sample is a set of labeled words from the unknown target language U from which a learner can infer an automaton B whose language is *equivalent* to U.

LeoParDS can also be used to generate a CS for fine BPs, as well as automatically detecting the cutoff of a fine BP. Additional functionalities of **LeoParDS** are listed in the *overview* paragraph. What makes **LeoParDS** unique is that it supports learning for a parametric number of processes, namely without assuming a fixed known number of processes that run in parallel.[5]

LeoParDS relies on the theoretical ideas developed in [29]. There, a learning algorithm is devised that can infer a correct BP from a sample that is consistent with a fine BP, and it is proven that it will derive a minimal equivalent BP if the sample is sufficiently complete (i.e., subsumes a CS). Various other learning problems concerning fine BPs are answered in [29], unfortunately on the negative side. In particular, it is shown (i) that the consistency problem for fine BPs is NP-hard, (ii) that characteristic samples may be of size exponential w.r.t. the number of the states of the BP, and (iii) that under standard cryptographic assumptions, fine BPs are not polynomially predictable.

These negative results, and the fact that the inference algorithm also relies on a constraint solver (an SMT solver to be precise), make one wonder whether

[4] A hidden state is a state where there are no outgoing broadcast sending transition.

[5] As far as we know, it is the first tool that enables learning of a distributed computational model in a *parameterized* setting. Hence the name **LeoParDS**, which stands for *Learning of Parameterized Distributed Systems*.

inferring a BP can be made practical, and may deter one from trying to invest in an implementation. Since availability of such a tool can benefit the scientific community, we took on ourselves the mission of implementing the first such tool. This paper is here to show that the single positive result from [29] actually holds in it many opportunities and deserves to be further studied by both theoreticians and practitioners. We note that many research fields such as studies of multi-agent systems [22,27,56], verifying concurrent systems and protocols [24,18] and distributed decision-making processes and strategies among multiple agents or players [15,43] in game-theory can use the techniques implemented in this tool to uncover novel insights and approaches in distributed computing.

To start bridging the gap between the theoretical results and their application in practice, we have implemented the techniques presented in [29]. Furthermore, for cases where the theory does not give us a complete solution, we implemented approximate methods that enhance the applicability of these techniques (at the cost of strong correctness guarantees). The result is a tool that allows us to demonstrate that even the techniques with a high theoretical complexity scale surprisingly well in practice (on randomly generated BPs), and that can solve a number of tasks that could benefit anyone interested in learning-based techniques for BPs, and potentially other parameterized distributed systems.

Related work. Learning of computational models is broadly classified into active and passive learning algorithms. The algorithms for active and passive learning of DFAs [2,51,42,39,55,17,41] have been extended to various other computational models including non-deterministic and alternating automata [21,12,5], symbolic and lattice automata [7,28,31], ω-automata [45,26,6,3,4,11,46], register automata [38,16], multiplicity, weighted and probabilistic automata and grammars [9,53,8,30], and more. The concurrent models for which a learning algorithm has been developed include communicating automata [13], workflow Petri nets [25], and negotiation protocols [47]. The main difference between our work and these works is that we assume that an *arbitrary* number of processes can interact while these works assume a *fixed* number of processes.

Our learning algorithm belongs to the class of constraint-based learning algorithms. The first constraint-based algorithm for DFAs is due to Biermann and Feldman [10]. This algorithm was refined and improved [49,35,37]. Constraint-based algorithms are also used for learning temporal logic-formulas [48,52].

In terms of tools, the open source libraries LibAlf [14] and LearnLib [40] implement many of the algorithms for learning DFAs, Mealy and Moore machines, as well as for more powerful computational models such as visibly-pushdown automata. For ω-regular languages there are the tools ROLL [44] and ALMA [32]. We are not aware of tools for learning the concurrent models mentioned above.

Overview. The central task of **LeoParDS** is to infer a BP from a given sample. This is task BPInf (BPInfMin) listed under 3 below. In addition to that, it can solve four related tasks. The five main tasks that the tool can solve are:

1. **CSGen**: Given a fine BP $\mathcal{B}$, return a characteristic sample $\mathcal{S}_{\mathcal{B}}$ for it.
2. **RSGen**: Given a BP $\mathcal{B}$, subject to certain parameters, return a random sample $\mathcal{S}$ of words with labels corresponding to $\mathcal{B}$.

3. **BPInf** (and **BPInfMin**): Given a sample $\mathcal{S}$, return a (minimal) BP that is consistent with $\mathcal{S}$.
4. **BPGen**: Given bounds on the numbers of states and actions, return a random BP $\mathcal{B}$ within these bounds.
5. **AEQ**: Given two BPs $\mathcal{B}_1$ and $\mathcal{B}_2$, return whether they are equivalent (i.e. accept the same language) up to some approximation.

Even though each of these 5 tasks can stand by itself, the experiments that we describe later on use all of them. First we generate a BP (BPGen). Given a BP one can do one of two things, (1) learn an equivalent (and potentially minimal) fine BP. This can be done by generating a CS (CSGen), and inferring a BP from it (BPInf). Or (2) learn a consistent BP that is not necessarily equivalent to the original BP. This can be done by generating a random consistent sample (RSGen) (which is not necessarily a CS) and inferring a BP that is consistent with the sample. For option (1), in order to evaluate the tool (i.e., to show that we indeed learned an equivalent BP) we run AEQ and check that the language of the original and the generated BP are indeed equivalent. We discuss these tasks in detail in §2-§6. In §7 we discuss empirical results gathered on runs of **LeoParDS** on a large collection of randomly generated BPs.

2 Characteristic Sample Generation

Roughly speaking, a characteristic sample (CS) $\mathcal{S}_\mathcal{B}$ is a set of labeled words from which a learner can correctly infer the target model $\mathcal{B}$, i.e., given $\mathcal{S}_\mathcal{B}$ or any sample subsuming it, a learner can generate a BP $\mathcal{B}'$ such that $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B}')$. The generation of a CS for BPs [29, Sec.5] is more involved than that for DFAs [34]. This section explains how the module CSGen generates a CS for a given fine BP.

In the context of BPs, a *sample* is a set of triples (w, n, b), where $w \in A^*$, i.e., a word over the actions A of the given BP $\mathcal{B}$, $n \in \mathbb{N}$ is a number of processes, and b is a truth value stating whether this word is feasible with n processes that execute $\mathcal{B}$ in parallel. When b is T (resp. F) this example is termed a *positive* (resp. *negative*) example. For the BP $\mathcal{B}_{toy}$ from Fig. 1, the triple $(bb, 1, \text{T})$ states that the sequence of actions bb is feasible in $\mathcal{B}^1_{toy}$ (as a single process can take the sending transition labeled with $b!!$ from the initial state an arbitrary number of times). The triple $(ba, 1, \text{F})$ states that the sequence of actions ba is infeasible in $\mathcal{B}^1_{toy}$ (as a single process can only execute b^*, but is not able to move from s_0 to s_1). The triple $(ba, 2, \text{T})$ states that ba is feasible in $\mathcal{B}^2_{toy}$ (if one process executes $b!!$, the second will move along the receiving transition labeled $b??$ from its current state s_0 to s_1, which will enable the sending transition on $a!!$, thus making ba feasible).

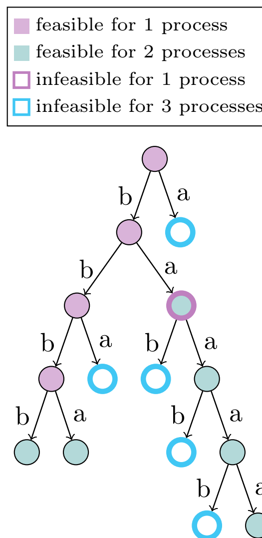


Fig. 2: Tree representing the CS of $\mathcal{B}_{toy}$

A tree representing the characteristic sample $\mathcal{S}_\mathcal{B}$ for $\mathcal{B}_{toy}$ is sketched in Fig. 2. Note that for any BP, feasibility of word w for n

processes, i.e., (w, n, T), implies that (w, m, T) for any $m \geq n$. Similarly, infeasibility of a word w for n processes, i.e., (w, n, F), implies that w is also infeasible for any $m \leq n$. We use these implications to avoid clutter in the figure by only representing feasible (resp., infeasible) examples for the minimal (resp., maximal) number of processes with which they appear in the sample. In Fig. 2, nodes in violet signify that $(w, 1, \text{T}) \in \mathcal{S}_{\mathcal{B}}$, stating that w, the sequence of actions from the root to the given node, is feasible in $\mathcal{B}^1_{toy}$. Nodes in teal represent a triple $(w, 2, \text{T}) \in \mathcal{S}_{\mathcal{B}}$, corresponding to words that are feasible in $\mathcal{B}^2_{toy}$, but *not* making any assumption about feasibility of w in $\mathcal{B}^1_{toy}$ (e.g., $bbbb$ is feasible both in $\mathcal{B}^2_{toy}$ and in $\mathcal{B}^1_{toy}$, but $\mathcal{S}_{\mathcal{B}}$ only contains $(bbbb, 2, \text{T})$). If a node has a violet border, we also have $(w, 1, \text{F}) \in \mathcal{S}_{\mathcal{B}}$, stating that w is infeasible in $\mathcal{B}^1_{toy}$. Similarly, nodes with a blue border such as $(w, 3, \text{F}) \in \mathcal{S}_{\mathcal{B}}$, signify that w is infeasible in $\mathcal{B}^3_{toy}$.

In order to ensure termination also in case the given BP has no cutoff, CSGen assumes a given bound, $\overline{\text{M}}_p$, on the number of processes. If the input BP $\mathcal{B}$ has cutoff $c \leq \overline{\text{M}}_p$, procedure CSGen generates a CS $\mathcal{S}_{\mathcal{B}}$ for $\mathcal{B}$. Otherwise, i.e., if either $c > \overline{\text{M}}_p$ or $\mathcal{B}$ does not have a cutoff, it will generate the trimming of the CS that consists only of triples where the number of processes is bounded by $\overline{\text{M}}_p$.

3 Random Sample Generation

In many scenarios the data set given to the learning algorithm may not necessarily subsumes a characteristic set. **LeoParDS** can also be given a user-defined sample. In addition, RSGen can be used to generate a random sample. The module RSGen receives a parameter F_w that describes the number of words required to be in the sample, a parameter $\overline{\text{M}}_\ell$ that bounds the length of words in the sample, a parameter $\overline{\text{M}}_p$ that bounds the number of processes, and an optional parameter F_r to determine the ratio of positive words in the sample (out of the total number of words in the sample). If the optional parameter is not given, RSGen repeatedly calls procedure RWGen, which draws uniformly at random a number ℓ in 1 to $\overline{\text{M}}_\ell$ and then constructs a word w of length ℓ by randomly drawing an action for each position of the word. It then draws uniformly at random a number p in 1 to $\overline{\text{M}}_p$ and checks whether w is feasible in $\mathcal{B}$ with p processes, and adds the triple with the respective answer (w, p, T) or (w, p, F) to the sample.

Since the probability that a randomly drawn word is in the language of a given BP is very small and decreases with increasing length of words, the optional parameter F_r can be used to obtain a sample with the desired ratio of positive examples. If this parameter is given, an alternative procedure RPWGen is used to generate positive words, which randomly draws a number of processes p in 1 to $\overline{\text{M}}_p$, and a length ℓ in 1 to $\overline{\text{M}}_\ell$, and then simulates a random run of those p processes for ℓ steps as follows. It holds a state vector, a configuration, that records the position of all processes. At the beginning, they are all at the initial state. At step $i \in [1..\ell]$ it checks what are the enabled actions A' according to current positions of the processes. It then randomly chooses an action a from A' and simulates the transition on this action (one process, the sender, takes the sending transition, and the rest of the processes follow the receiving transition). The negative words, i.e. a $(1 - F_r)$ of the words, will be randomly generated and checked to be negative examples, if so then they will be added to the sample.

As a small example, if we call RSGen with the BP $\mathcal{B}_{toy}$ from Fig. 1 and parameters $F_w = 5$, $\overline{\mathrm{M}}_\ell = 5$, $\overline{\mathrm{M}}_p = 3$, $F_r = 0.2$, the output could be the sample $\mathcal{S} = \{(aabab, 2, \mathrm{F}), (abbb, 2, \mathrm{F}), (baa, 3, \mathrm{T}), (bba, 2, \mathrm{F}), (ba, 1, \mathrm{F})\}$.

4 Inference of a BP from a Sample

The modules BPInf and BPInfMin are the central part of the tool **LeoParDS**. The module BPInf infers a BP from a given sample as described in [29]. As usual in passive learning algorithms, it guarantees to return a *minimal* representation *only if* the sample subsumes a CS. The module BPInfMin is a modification of BPInf that is guaranteed to always return the minimal BP consistent with the sample, i.e., even if the sample does not subsumes a CS. The high-level architecture of BPInf and BPInfMin are given in Fig. 3 and Fig. 4, respectively.

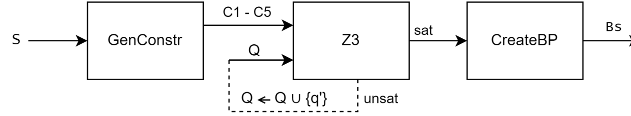


Fig. 3: The high-level architecture of BPInf

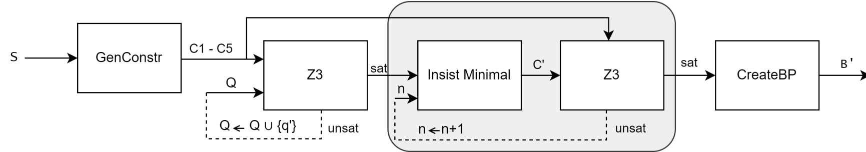


Fig. 4: The high-level architecture of BPInfMin

The procedure GenConstr generates from a given sample $\mathcal{S}$ five sets of constraints $C_1, \ldots, C_5$, as described in [29, Sec.4]. These constraints are passed to the SMT solver Z3 [19].[6] If the result is `sat`, i.e., the constraints are satisfiable, then a satisfying assignment is passed to CreateBP that constructs from it a BP $\mathcal{B}'$ that is consistent with $\mathcal{S}$ as per [29, Thm 4.1]. Moreover, if the sample $\mathcal{S}$ subsumes a CS of some BP $\mathcal{B}$ then $\mathcal{B}'$ will be *correct*, namely equivalent to $\mathcal{B}$ and minimal among all BPs that are equivalent to $\mathcal{B}$ [29, Thm 5.3].

The constraints $C_1, \ldots, C_5$ are defined with respect to a set of states Q, a set of actions A, and partial functions $f^{\mathsf{st}} : A \to Q$, $f^{!!} : A \to Q$, and $f^{??}_a : Q \to Q$ for every $a \in A$. Functions f^{st} and $f^{!!}$ associate with each action the origin and target state, respectively, and each function $f^{??}_a$, for each action $a \in A$, associates with each state q the target state for the receiving $a??$ transition from q. Initially, A consists of all actions in the sample and Q consists of one state per action, namely $Q = \{f^{\mathsf{st}}(a) \mid a \in A\}$. If from some states more than one action is enabled then fewer states will be required. For example, if $f^{\mathsf{st}}(a) = f^{\mathsf{st}}(b)$, then the set of states will have a single value representing both $f^{\mathsf{st}}(a)$ and $f^{\mathsf{st}}(b)$.

[6] Note that all variables in our SMT constraints are over finite domains with known size, implying that our constraints are decidable, and Z3 provides a decision procedure for such constraints.

If the sample subsumes a CS then the constraints $C_1, \ldots, C_5$ are satisfiable and the BP $\mathcal{B}'$ constructed by CreateBP is correct and minimal. If the sample does not subsume a CS then there are two options. The first option is that the constraints $C_1, \ldots, C_5$ are immediately satisfiable, although the sample does not subsumes a CS. The second is that the constraints are not satisfiable. In this case, we incrementally add states to Q. (See the loop in Fig.3 at the middle.) Each new state is associated with an action that does not appear in the sample. We add the states incrementally and try to satisfy the constraints $C_1, \ldots, C_5$ relative to the larger set of states Q until they become satisfiable, at which point CreateBP will return a consistent BP that agrees with the sample.

In both cases it could be that the satisfying assignment used more states in Q than necessary. Hence, if we want to insist that a minimal BP is returned, BPInfMin proceeds as follows. A new parameter n that bounds the number of states to n is introduced. (See the loop on the gray part of Fig.4.) BPInfMin tries to satisfy the constraints with the additional requirement that each state corresponds to a number in $[1..n]$. Since we gradually increment n it is guaranteed that we return a minimal BP that is consistent with the sample in this case too.[7]

Note that even in the second case, i.e., when the constraints are not satisfiable immediately and states were added incrementally until the constraints became satisfiable, unless we use BPInfMin, the returned BP might not be minimal. To see how this can happen consider the sample $\mathcal{S} = \{(aa, 1, \text{F}), (ba, 1, \text{F}), (bb, 1, \text{F}), (bab, 2, \text{T}), (baabb, 2, \text{T})\}$, which agrees with the BP $\mathcal{B}_1$ of Fig.5.

In order to have a satisfying assignment for this sample, we must have more states than $f^{\mathsf{st}}(a)$ and $f^{\mathsf{st}}(b)$ (which are the initial value of the set of Q). Note that from the sample, we know that $f^{\mathsf{st}}(b)$ is the initial state (since bab is feasible). However, $f^{!!}(b) \neq f^{\mathsf{st}}(a)$ (since $(ba, 1, \text{F}) \in \mathcal{S}$) and $f^{!!}(b) \neq f^{\mathsf{st}}(b)$ (since $(bb, 1, \text{F}) \in \mathcal{S}$). Therefore, a new state is required so that $f^{!!}(b)$ will be equal to it. Let $f^{\mathsf{st}}(q)$ be the state added in the loop in order to find a satisfying assignment. Now, with $Q = \{f^{\mathsf{st}}(a), f^{\mathsf{st}}(b), f^{\mathsf{st}}(q)\}$ the constraints are satisfiable. However, as we can see in Fig.5, both of the BPs $\mathcal{B}_1, \mathcal{B}_2$ agree with the sample $\mathcal{S}$. Yet, the SMT solver may return the one with three states rather than the one with two. The language of the two BPs is different (note that $a \in \mathcal{L}(\mathcal{B}_1^1)$ but a is infeasible in $\mathcal{L}(\mathcal{B}_2^1)$) yet both are consistent with the given sample.

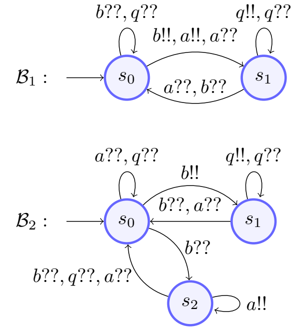


Fig. 5

5 Generation of Random BPs

LeoParDS can be used to generate a sample for a user-defined BP with no hidden states. To generate a random BP, the module BPGen randomly generates BPs with no hidden states. The module receives parameters $\underline{\text{M}}_s, \overline{\text{M}}_s$ that bound the number of states and $\underline{\text{M}}_a, \overline{\text{M}}_a$ that bound the number of actions. It first

[7] Alternatively we can search for the exact n using a binary search, but we have not yet implemented this option.

chooses at random a number $n_s > 1$ between $\underline{\mathrm{M}}_s$ and $\overline{\mathrm{M}}_s$; this will be the number of states. Each state s is initially associated with a unique action a that is the sending action from s. This assures us that the generated BP will have no hidden states. Then a random number n_a between $\underline{\mathrm{M}}_a$ and $\overline{\mathrm{M}}_a$ of additional actions is chosen; these actions are distributed randomly between the states. After this step there could be more than one sending transition from each state, as is the case in state s_0 in $\mathcal{B}_1$ in Fig. 5.

Finally, for every action a, its broadcast sending and receiving transitions are determined: the sending transition is an edge from the state s associated with a to a randomly chosen target state s', and is labelled by $a!!$. Furthermore, for every state s and every action $a \in A$, a receiving transition is determined by picking a random target state s', and this transition is labelled with $a??$.

As an example, if BPGen is called with $\underline{\mathrm{M}}_s = 2, \overline{\mathrm{M}}_s = 3, \underline{\mathrm{M}}_a = 0, \overline{\mathrm{M}}_a = 2$, the output could be the BP $\mathcal{B}_1$ or $\mathcal{B}_2$ shown in Fig. 5.

6 Checking approximate equivalence of two BPs

Since checking equivalence of BPs is probably infeasible[8], we implemented an approximate equivalence check. The module AEQ receives two BPs $\mathcal{B}_1$ and $\mathcal{B}_2$, a bound $\overline{\mathrm{M}}_c$ for the cutoff, a bound $\overline{\mathrm{M}}_\ell$ on the length of words, a bound $\overline{\mathrm{M}}_w$ on the number of words, and a bound $\overline{\mathrm{M}}_t$ on the running time.

The first approach tries to generate a CS for both $\mathcal{B}_1$ and $\mathcal{B}_2$ incrementally, first for 1 process, then for 2 processes, etc., until reaching the bound $\overline{\mathrm{M}}_c$. For each triple (w, n, b) where $w \in A^*$, $n \in \mathbb{N}$, $b \in \{\mathrm{T}, \mathrm{F}\}$ that is added to the CS of $\mathcal{B}_1$ it checks whether w is feasible (resp. infeasible) with n processes in $\mathcal{B}_2$ iff $b = \mathrm{T}$ (resp. $b = \mathrm{F}$), and similarly for each triple added to the CS of $\mathcal{B}_2$. If it is feasible in one but not in the other it returns “no”, otherwise it continues until reaching $\overline{\mathrm{M}}_c$ at which point it returns “yes”.

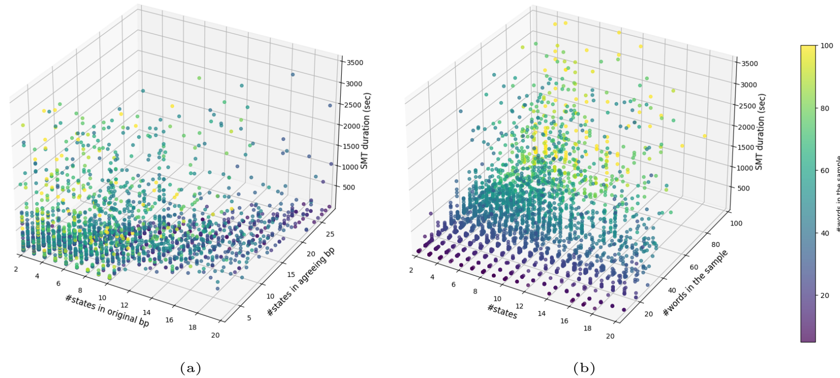
The second approach instead of exhaustively generating the CS of both BPs conducts a random walk on the BPs (and again checks for disagreement between the two BPs on some word and some number of processes). For a given number of processes n_c in $[1, \overline{\mathrm{M}}_c]$, starting with $n_c = 1$, it maintains a pair of configurations (state-vectors) $(\mathbf{v}_1, \mathbf{v}_2)$, one for each of the BPs. Initially $\mathbf{v}_i$ is the state-vector where all n_c processes are in the initial state of $\mathcal{B}_i$. It then defines A_i (for $i \in \{1, 2\}$) to be the set of actions enabled from $\mathbf{v}_i$ in $\mathcal{B}_i$. If $A_1 \neq A_2$ it returns “no”. Otherwise, it randomly chooses an action $a \in A_i$ and updates the current pair of configurations to be those obtained by the broadcast of action a. It continues this way until either $A_1 \neq A_2$ or the limit on the length of words is reached. If the limit on the length of a word is reached, it restarts the walk. If the limit on the number of words is reached, we increase n_c and repeat the process. If $n_c = \overline{\mathrm{M}}_c$ or the time bound is reached it returns “yes”.

[8] Checking reachability of local states has Ackermannian complexity [54]. To the best of our knowledge no better complexity algorithm for checking equivalence is known.

7 Experiments

Our experimental results may lack the fireworks associated with groundbreaking discoveries, but they carry immense value:
(1) **Practical Applicability: LeoParDS** consistently solved the generated instances. It navigated the intricacies of the inference problem, demonstrating its practical utility. The gap between theory and reality narrowed significantly.
(2) **Scalability:** As the problem complexity increased, **LeoParDS** maintained predictable performance. This scalability bodes well for future applications.
(3) **Broader Implications:** Our contribution extends beyond the inference problem for BPs. Researchers in diverse fields: verification, multi-agent learning, concurrent systems, strategies among multiple agents or players and others can leverage **LeoParDS**. By releasing it to the scientific community, we lay the foundation for future advancements.

We ran all experiments on a cluster with Intel Xeon E5-2620 v4 CPUs. We allocated one CPU core and 30GB of RAM with time limit of 1 hour. For our experiments we randomly generated 4149 BPs with a number of states in $[2, 20]$,



(a) (b)

Fig. 6: SMT duration according to several parameters

number of actions in $[0, 8]$. For each of these BPs we generated a random sample with a random number of words, F_w, in $[5, 100]$; with a bound of $\overline{\mathrm{M}}_\ell$=20 on the length of the words, and a bound of $\overline{\mathrm{M}}_p$=20 for the number of processes. The ratio of positive examples in the sample ranges between 0 and 1. Table 1a provides

#states	%BPs
[2, 5)	36.64%
[5, 10)	37.27%
[10, 15)	16.87%
[15, 20]	9.23%

(a)

#words	%BPs
[0, 25)	21.67%
[25, 50)	44.28%
[50, 75)	24.74%
[75, 100]	9.30%

(b)

pos-ratio	%BPs
[0.0, 0.10)	40.46%
[0.10, 0.25)	39.22%
[0.25, 0.50)	16.50%
[0.50, 1.0]	3.82%

(c)

SMT(min)	#BPs
[0, 5)	2914
[5, 30)	773
[30, 60]	161
timeout	301

(d)

Table 1: Statistical information

details on the number of states of the generated BPs. Note that the number of states of a minimal equivalent BP might be smaller. Table 1b provides details on the number of words in the generated samples. In the first 850 generated random samples we used RWGen to randomly generate words. As discussed in §3, words

generated by RWGen are more likely to be negative, and the probability increases with the length of words. Therefore, the ratio of positive words is generally low. To address this phenomenon, we implemented RPWGen which generates positive words. Table 1c provides details on the ratio of positive words in all the samples. We run BPInf on each of the generated samples. Table 1d provides details on the time (in minutes) it took the SMT-solver to find a satisfying assignment. The time to generate the BP from the assignment is negligible. We can see that 70.23% of the examples terminated in less than 5 minutes, and 7.25% timed out. Tables 1a-1c provide information only on the samples that did not time out.

Fig.6 shows the time needed for SMT solving (in seconds) relative to various parameters. In both figures 6a and 6b the z-axis is the SMT solving time, and the colors correspond to the number of words in the sample. In Fig. 6a the x-axis (resp. y-axis) shows the number of states in the randomly generated BP (resp. in the BP learned by BPInf). We can see that the number of states in the inferred BP is very close to the number of states in the generated BP. Recall that it could be smaller, since the randomly generated BP might not be minimal.

In Fig. 6b the x-axis shows the number of states in the randomly generated BP, and the y-axis shows the number of words in the sample. We can see that while the SMT time is affected mostly by the number of words in the sample, for BPs with a large number of states and actions the SMT time is larger even with fewer words, see Fig. 7a (App. A). When comparing the effect of positive vs negative words in the sample, we can see that the SMT time is mostly affected by the number of negative words. Note also that negative words are relatively longer, and may include more actions, both factors may affect the SMT time, see Fig. 7b (App. A).

8 Conclusions and Future Work

We have presented **LeoParDS**, the first automatic tool for passive learning of parameterized distributed systems, in particular in the form of broadcast protocols (BPs). In addition to its main task, the inference of a BP from a sample, **LeoParDS** supports the generation of random BPs, the generation of a characteristic or a random sample from a BP, as well as approximate equivalence checks between two BPs. All of these tasks come with a number of parameters that give the user control over the precision and the required resources. Based on the tasks that are already implemented, **LeoParDS** can be used for instance in the field of robotics and autonomous systems to learn control policies from observed sensor data, as well as a toolbox for future developments in the learning of parameterized distributed systems.

In future work, we plan to investigate both practice-oriented extensions of **LeoParDS**, such as the integration of other SMT solvers or heuristical support for non-fine BPs, and fundamental extensions to support other computational models, such as reconfigurable broadcast networks [20], rendezvous systems [33,1], or Petri nets/VASS [50,36]. In conclusion, **LeoParDS** is not just a solution; it's an invitation to explore uncharted territories.

References

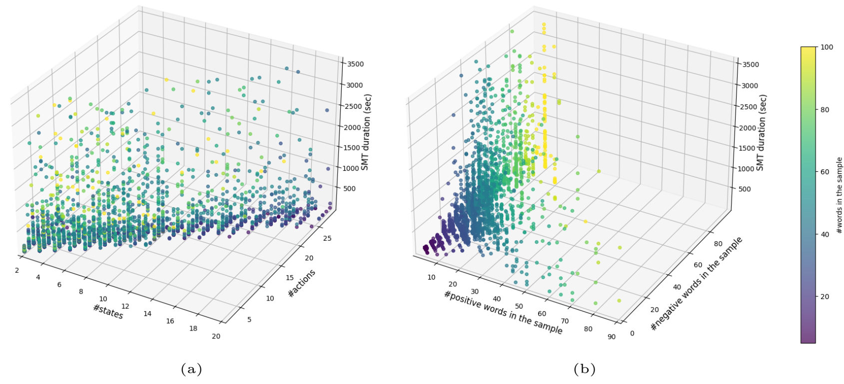
1. B. Aminof, T. Kotek, S. Rubin, F. Spegni, and H. Veith. Parameterized model checking of rendezvous systems. In *CONCUR*, volume 8704 of *Lecture Notes in Computer Science*, pages 109–124. Springer, 2014.
2. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
3. D. Angluin, T. Antonopoulos, and D. Fisman. Query learning of derived $\omega\omega$\omega-tree languages in polynomial time. *Log. Methods Comput. Sci.*, 15(3), 2019.
4. D. Angluin, T. Antonopoulos, and D. Fisman. Strongly unambiguous büchi automata are polynomially predictable with membership queries. In *28th EACSL Annual Conference on Computer Science Logic, CSL 2020*, pages 8:1–8:17, 2020.
5. D. Angluin, S. Eisenstat, and D. Fisman. Learning regular languages via alternating automata. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015*, pages 3308–3314, 2015.
6. D. Angluin and D. Fisman. Learning regular omega languages. *Theor. Comput. Sci.*, 650:57–72, 2016.
7. G. Argyros and L. D'Antoni. The learnability of symbolic automata. In *Computer Aided Verification - 30th International Conference, CAV 2018, Proceedings, Part I*, pages 427–445, 2018.
8. B. Balle and M. Mohri. Learning weighted automata. In *Algebraic Informatics - 6th International Conference, CAI 2015. Proceedings*, pages 1–21, 2015.
9. A. Beimel, F. Bergadano, N. H. Bshouty, E. Kushilevitz, and S. Varricchio. Learning functions represented as multiplicity automata. *J. ACM*, 47(3):506–530, 2000.
10. A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, June 1972.
11. L. Bohn and C. Löding. Constructing deterministic ω-automata from examples by an extension of the RPNI algorithm. In *46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021*, pages 20:1–20:18, 2021.
12. B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-style learning of NFA. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, 2009*, pages 1004–1009, 2009.
13. B. Bollig, J. Katoen, C. Kern, and M. Leucker. Learning communicating automata from mscs. *IEEE Trans. Software Eng.*, 36(3):390–408, 2010.
14. B. Bollig, J. Katoen, C. Kern, M. Leucker, D. Neider, and D. R. Piegdon. libalf: The automata learning framework. In *Computer Aided Verification, 22nd International Conference, CAV 2010. Proceedings*, pages 360–364, 2010.
15. C. Boutilier. Planning, learning and coordination in multiagent decision processes. In *TARK*, volume 96, pages 195–210. Citeseer, 1996.
16. S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Active learning for extended finite state machines. *Formal Asp. Comput.*, 28(2):233–263, 2016.
17. A. Castellanos, E. Vidal, M. A. Varó, and J. Oncina. Language understanding and subsequential transducer learning. *Computer Speech & Language*, 12(3):193–228, 1998.
18. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, apr 1986.

19. L. M. de Moura and N. S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
20. G. Delzanno, A. Sangnier, R. Traverso, and G. Zavattaro. On the complexity of parameterized reachability in reconfigurable broadcast networks. In *FSTTCS*, volume 18 of *LIPIcs*, pages 289–300. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.
21. F. Denis, A. Lemay, and A. Terlutte. Residual finite state automata. In *STACS 2001, 18th Annual Symposium on Theoretical Aspects of Computer Science,Proceedings*, pages 144–157, 2001.
22. A. Dorri, S. S. Kanhere, and R. Jurdak. Multi-agent systems: A survey. *IEEE Access*, 6:28573–28593, 2018.
23. E. A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *CADE*, volume 1831 of *Lecture Notes in Computer Science*, pages 236–254. Springer, 2000.
24. E. A. Emerson and K. S. Namjoshi. Automatic verification of parameterized synchronous systems (extended abstract). In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification, 8th International Conference, CAV '96, Proceedings*, volume 1102 of *Lecture Notes in Computer Science*, pages 87–98. Springer, 1996.
25. J. Esparza, M. Leucker, and M. Schlund. Learning workflow petri nets. *Fundam. Informaticae*, 113(3-4):205–228, 2011.
26. A. Farzan, Y.-F. Chen, E. Clarke, Y.-K. Tsay, and B.-Y. Wang. Extending automated compositional verification to the full class of omega-regular languages. In *TACAS*, pages 2–17, 2008.
27. J. Ferber and G. Weiss. *Multi-agent systems: an introduction to distributed artificial intelligence*, volume 1. Addison-wesley Reading, 1999.
28. D. Fisman, H. Frenkel, and S. Zilles. Inferring symbolic automata. *Log. Methods Comput. Sci.*, 19(2), 2023.
29. D. Fisman, N. Izsak, and S. Jacobs. Learning broadcast protocols. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(11):12016–12023, Mar. 2024.
30. D. Fisman, D. Nitay, and M. Ziv-Ukelson. Learning of structurally unambiguous probabilistic grammars. *Log. Methods Comput. Sci.*, 19(1), 2023.
31. D. Fisman and S. Saadon. Learning and characterizing fully-ordered lattice automata. In *Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022, Proceedings*, pages 266–282, 2022.
32. N. George. ALMA: automata learner using modulo 2 multiplicity automata. *CoRR*, abs/2301.04077, 2023.
33. S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.
34. E. M. Gold. Complexity of automaton identification from given data. *Inf. Control.*, 37(3):302–320, 1978.
35. O. Grinchtein, M. Leucker, and N. Piterman. Inferring network invariants automatically. In *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Proceedings*, pages 483–497, 2006.
36. M. Hack. *Decidability questions for Petri Nets.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1976.
37. M. Heule and S. Verwer. Exact DFA identification using SAT solvers. In *Grammatical Inference: Theoretical Results and Applications, 10th International Colloquium, ICGI 2010. Proceedings*, pages 66–79, 2010.

38. F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012. Proceedings*, pages 251–266, 2012.
39. M. Isberner, F. Howar, and B. Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *Runtime Verification - 5th International Conference, RV 2014. Proceedings*, pages 307–322, 2014.
40. M. Isberner, F. Howar, and B. Steffen. The open-source learnlib - A framework for active automata learning. In *Computer Aided Verification - 27th International Conference, CAV 2015, Proceedings, Part I*, pages 487–495, 2015.
41. P. G. José Oncina. Identifying regular languages in polynomial time. In *Advances in structural and syntactic pattern recognition*, pages 99–108, 1992.
42. M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA, 1994.
43. S. Kraus. *Automated Negotiation and Decision Making in Multiagent Environments*, pages 150–172. Springer Berlin Heidelberg, 2001.
44. Y. Li, Y.-F. Chen, L. Zhang, and D. Liu. A novel learning algorithm for büchi automata based on family of dfas and classification trees. *Information and Computation*, 281:104678, 2021.
45. O. Maler and A. Pnueli. On the learnability of infinitary regular sets. *Inf. Comput.*, 118(2):316–326, 1995.
46. J. Michaliszyn and J. Otop. Learning infinite-word automata with loop-index queries. *Artif. Intell.*, 307:103710, 2022.
47. A. Muscholl and I. Walukiewicz. Active learning for sound negotiations. In *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science,2022*, pages 21:1–21:12, 2022.
48. D. Neider and I. Gavran. Learning linear temporal properties. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018*, pages 1–10, 2018.
49. A. L. Oliveira and J. P. M. Silva. Efficient algorithms for the inference of minimum size dfas. *Machine Learning*, 44(1/2):93–119, 2001.
50. C. Rackoff. The covering and boundedness problems for vector addition systems. *Theor. Comput. Sci.*, 6:223–231, 1978.
51. R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, Apr. 1993.
52. R. Roy, D. Fisman, and D. Neider. Learning interpretable models in the property specification language. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 2213–2219, 2020.
53. Y. Sakakibara. Learning context-free grammars using tabular representations. *Pattern Recognit.*, 38(9):1372–1383, 2005.
54. S. Schmitz and P. Schnoebelen. The power of well-structured systems. In *CONCUR*, volume 8052 of *Lecture Notes in Computer Science*, pages 5–24. Springer, 2013.
55. F. W. Vaandrager, B. Garhewal, J. Rot, and T. Wißmann. A new approach for active automata learning based on apartness. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Proceedings, Part I*, pages 223–243, 2022.
56. G. Weiss. *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT press, 1999.

Appendix

A Additional Plots



(a) (b)

Fig. 7: More plots for SMT duration according to several parameters