# 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.

## 1 Introduction

We present **LeoParDS**, an automatic tool for passive learning of broadcast protocols (BPs) 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 [27] or disjunctive guards [20].

Given a set of labeled example traces, **LeoParDS** can be used to identify a BP that agrees with these examples, and that optionally is of minimal size with this property. Moreover, given a BP $\mathcal{B}$, **LeoParDS** can generate a *characteristic sample* for $\mathcal{B}$, i.e., a set of labeled example traces from which a learner can correctly infer a BP $\mathcal{B}'$ equivalent to $\mathcal{B}$.

What makes **LeoParDS** unique is that it supports learning for a parametric number of processes, i.e., if $\mathcal{B}'$ is learned from a characteristic sample of $\mathcal{B}$, then the language of $\mathcal{B}'^n$ ($n$ processes running $\mathcal{B}'$ concurrently) is guaranteed to be equivalent to the language of $\mathcal{B}^n$, for any $n$. [3] The characteristic sample is guaranteed to be finite if $\mathcal{B}$ is *fine*, i.e., it does not have hidden states 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 $n > k$. In this case, **LeoParDS** will automatically detect the cutoff.

**LeoParDS** relies on the theoretical ideas developed in [24]. There, we have devised a learning algorithm that can infer a correct BP from a sample that is

[3] 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*.

consistent with a fine BP, and proved that it will derive a minimal equivalent BP if the sample is sufficiently complete (subsumes a *characteristic sample*). On the negative side we showed that characteristic samples may be of exponential size, and that under standard cryptographic assumptions, fine BPs are not polynomially predictable.

To start bridging the gap between these theoretical results and their application in practice, we have implemented the techniques presented in [24]. 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 [2,43,35,32,47,17,34] have been extended to various other computational models including non-deterministic and alternating automata [19,13,5], symbolic and lattice automata [7,23,26], $\omega$-automata [37,22,6,3,4,11,38,12], register automata [31,16], multiplicity, weighted and probabilistic automata and grammars [9,45,8,25], and more.

The concurrent models for which a learning algorithm has been developed include communicating automata [14], workflow Petri nets [21], and negotiation protocols [39]. 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 further refined and improved [41,28,30]. Constraint-based algorithms are also used for learning temporal logic-formulas, see e.g. [40,44].

In terms of tools, the open source libraries LibAlf [15] and LearnLib [33] implemented many of the algorithms for learning DFAs, Mealy and Moore machines, as well as for some of the more powerful computational models such as visibly-pushdown automata. For learning $\omega$-regular languages there is the library ROLL [36]. 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. In addition to that, it can solve four related tasks. The five main tasks that the tool can solve are:

1. BPGen: Given bounds on the numbers of states and actions, return a random BP $\mathcal{B}$ within these bounds.

2. CSGen: Given a fine BP $\mathcal{B}$, return a characteristic sample $\mathcal{S}_\mathcal{B}$ for it.
3. RSGen: Given a BP $\mathcal{B}$, subject to certain parameters, return a random sample $\mathcal{S}$ of words with labels corresponding to $\mathcal{B}$.
4. BPInf (and BPInfMin): Given a sample $\mathcal{S}$, return a (minimal) BP that is consistent with $\mathcal{S}$.
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.

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 Generation of Random BPs

The module BPGen randomly generates BPs with no hidden states, i.e., states that do not have an outgoing broadcast sending transition. Given parameters $\underline{\mathrm{M}}_s, \overline{\mathrm{M}}_s$ that bound the number of states and $\underline{\mathrm{M}}_a, \overline{\mathrm{M}}_a$ that bound the number of actions, a number $n_s$ between $\underline{\mathrm{M}}_s$ and $\overline{\mathrm{M}}_s$ is chosen randomly for the number of states.[4] Then a number $n_a$ between $\underline{\mathrm{M}}_a$ and $\overline{\mathrm{M}}_a$ of actions is chosen and the actions are distributed randomly between the states. For any state that has not been associated with an action so far, additional actions are added to these states to ensure that the BP has no hidden states.

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

As an example, if BPGen is called with $\underline{\mathrm{M}}_s = 2, \overline{\mathrm{M}}_s = 3, \underline{\mathrm{M}}_a = 2, \overline{\mathrm{M}}_a = 3$, the output could be the BP $\mathcal{B}$ shown in Fig. 1. $\mathcal{B}$ has two states, $s_0$ and $s_1$, where $s_0$ is the initial state, and it has two actions $a$ and $b$. As in any BP, for each of the actions, there is exactly one sending transition — these are labelled $a!!$ and $b!!$, respectively —, and for each action and each state, there is exactly one receiving transition (or response) — these are labelled $a??$ and $b??$, respectively.

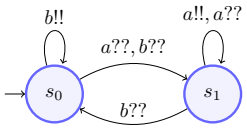


Fig. 1: BP $\mathcal{B}$ with 2 states

## 3 Characteristic Sample Generation

The module CSGen generates a characteristic sample from a given BP without hidden states, as described in [24, Sec.5]. We assume a given bound, $\overline{\mathrm{M}}_p$, on the number of processes to ensure termination also in case that the given BP does not have a cutoff.

[4] We require $\underline{\mathrm{M}}_s > 1$ and and $\underline{\mathrm{M}}_a \geq 1$.

If the input BP $\mathcal{B}$ has cutoff $c \leq \overline{\mathrm{M}}_p$, procedure CSGen generates a *characteristic sample* $\mathcal{S}_\mathcal{B}$ for $\mathcal{B}$, i.e., a sample that is sufficient (per the results in [24]) to uniquely identify the language $\mathcal{L}(\mathcal{B})$ of $\mathcal{B}$ and generate a minimal $\mathcal{B}'$ that has the same language. Otherwise, i.e., if either $c > \overline{\mathrm{M}}_p$ or $\mathcal{B}$ does not have a cutoff, it will generate the trimming of the characteristic sample that consists only of triples where the number of processes is bounded by $\overline{\mathrm{M}}_p$.

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. The language $\mathcal{L}(\mathcal{B}^n)$ is the set of feasible words with $n$ processes, and $\mathcal{L}(\mathcal{B})$ is the union over all $\mathcal{L}(\mathcal{B}^n)$. For the BP $\mathcal{B}$ from Fig. 1, the triple $(bb, 1, \mathrm{T})$ states that the sequence of actions $bb$ is feasible in $\mathcal{B}^1$ (as a single process can take the sending transition labeled with $b$!! from the initial state an arbitrary number of times), the triple $(ba, 1, \mathrm{F})$ states that the sequence of actions $ba$ is infeasible in $\mathcal{B}^1$ (as a single process can never move from $s_0$ to $s_1$), and the triple $(ba, 2, \mathrm{T})$ states $ba$ is feasible in $\mathcal{B}^2$ (if one process executes $b$!!, the other will move along the receiving transition labeled $b$?? to $s_1$, and can then take the sending transition on $a$!! in the next step).

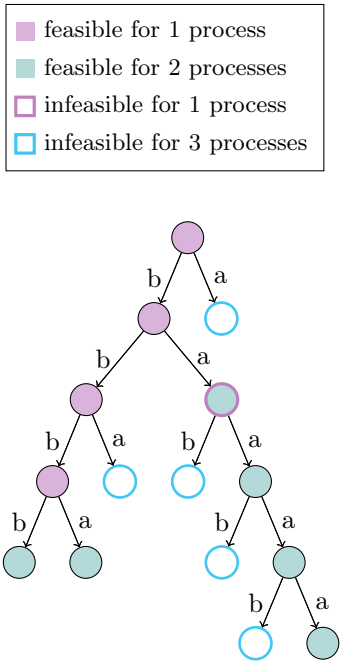


Fig. 2: Tree representing the characteristic sample of $\mathcal{B}$

A tree representing the characteristic sample $\mathcal{S}_\mathcal{B}$ for $\mathcal{B}$ is sketched in Fig. 2. Note that for any BP, feasibility of word $w$ for $n$ processes, i.e., $(w, n, \mathrm{T})$ implies that $(w, m, \mathrm{T})$ for any $m \geq n$. Similarly, infeasibility of a word $w$ for $n$ processes, i.e., $(w, n, \mathrm{F})$, 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 the figure, nodes in violet mean that $(w, 1, \mathrm{T}) \in \mathcal{S}_\mathcal{B}$, stating that $w$, the sequence of actions from the root to the given node, is feasible in $\mathcal{B}^1$. Nodes in teal represent a triple $(w, 2, \mathrm{T}) \in \mathcal{S}_\mathcal{B}$, corresponding to words that are feasible in $\mathcal{B}^2$, but *not* making any statement about feasibility of $w$ in $\mathcal{B}^1$ (e.g., $bbbb$ is feasible both in $\mathcal{B}^2$ and in $\mathcal{B}^1$, but $\mathcal{S}_\mathcal{B}$ only contains $(bbbb, 2, \mathrm{T})$). If a node has a violet border, we additionally have $(w, 1, \mathrm{F}) \in \mathcal{S}_\mathcal{B}$, stating that $w$ is infeasible in $\mathcal{B}^1$. Similarly, for nodes with a blue border we have $(w, 3, \mathrm{F}) \in \mathcal{S}_\mathcal{B}$, stating that $w$ is infeasible in $\mathcal{B}^3$.

## 4 Random Sample Generation

The module RSGen receives a parameter $F_w$ that describes the number of words required to be in the sample, a parameter $\overline{\mathrm{M}}_\ell$ that bounds the length of words in the sample, a parameter $\overline{\mathrm{M}}_p$ that bounds the number of processes, and an optional parameter $F_r$ to determine the ratio of positive words in the sample.

If the optional parameter is not given, RSGen repeatedly calls procedure RWGen, which draws uniformly at random a number $\ell$ in 1 to $\overline{\mathrm{M}}_\ell$ and then constructs a word $w$ of length $\ell$ by randomly drawing an action for each position of the word. It then draws uniformly at random a number $p$ in 1 to $\overline{\mathrm{M}}_p$ and checks whether $w$ is feasible in $\mathcal{B}$ with $p$ processes, and adds the triple with the respective answer $(w, p, \mathrm{T})$ or $(w, p, \mathrm{F})$ 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 $\overline{\mathrm{M}}_p$, and a length $\ell$ in 1 to $\overline{\mathrm{M}}_\ell$, and then simulates a random run of those $p$ processes for $\ell$ steps as follows. It holds a state vector 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).

To see the effect of positive vs. negative words in the sample, in the graphs showing the results of our experiments (in Sect. 7), we often distinguish between positive and negative words in the sample.

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

## 5 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 [24]. As usual in passive learning algorithms, it guarantees to return a *minimal* representation *only if* the sample subsumes a characteristic sample. 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 subsume a characteristic sample. The high-level architecture of BPInf and BPInfMin are given in Fig. 3 and Fig. 4, respectively.

The procedure GenConstr generates from a given sample $\mathcal{S}$ five sets of constraints $C_1, \ldots, C_5$, as described in [24, Sec.4]. These constraints are passed to

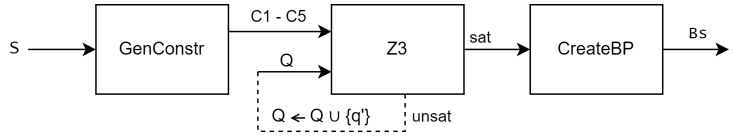


Fig. 3: The high-level architecture of BPInf

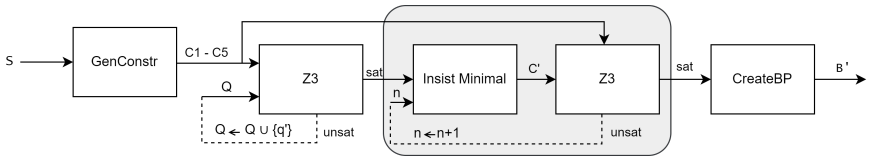


Fig. 4: The high-level architecture of BPInfMin

the SMT solver Z3. [5] 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 [24, Thm 4.1]. Moreover, if the sample $\mathcal{S}$ is a characteristic sample of some BP $\mathcal{B}$ then $\mathcal{B}'$ will be equivalent to $\mathcal{B}$ and minimal among all BPs that are equivalent to $\mathcal{B}$ [24, 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 $a \in A$. The function $f^{\mathsf{st}}$ associates each action with its origin state, the function $f^{!!}$ associates each action with its target state, and the function $f_a^{??}$ 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 $Q = \{f^{\mathsf{st}}(a) \colon a \in A\}$. 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 $f^{\mathsf{st}}(b)$. If the sample subsumes a characteristic sample then the constraints $C_1, \ldots, C_5$ are satisfiable and the BP $\mathcal{B}'$ that is constructed by CreateBP will be correct and minimal.

If the sample does not subsume a characteristic sample 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 subsume a characteristic sample. 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.

[5] 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.

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 as well.[6]

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.

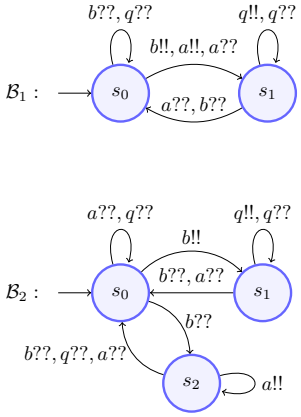


Fig. 5

## 6 Checking approximate equivalence of two BPs

Since checking equivalence of BPs is probably infeasible[7], we implemented an approximate equivalence check. The module AEQ receives two BPs $\mathcal{B}_1$ and $\mathcal{B}_2$, a bound $\overline{\text{M}}_c$ for the cutoff, a bound $\overline{\text{M}}_\ell$ on the length of words and a bound $\overline{\text{M}}_w$ on the number of words, and a bound $\overline{\text{M}}_t$ on the running time.

The first approach tries to generate a characteristic sample for both $\mathcal{B}_1$ and $\mathcal{B}_2$ incrementally, first for 1 process, then for 2 processes, etc., until reaching the

[6] Alternatively we can search for the exact $n$ using a binary search, but we haven't yet implemented this option.

[7] Checking reachability of local states has Ackermannian complexity [46], and to the best of our knowledge no algorithm with a better complexity for checking equivalence is known.

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 characteristic sample of $\mathcal{B}_1$ it checks whether $w$ is feasible with $n$ processes in $\mathcal{B}_2$ iff $b = \mathrm{T}$, and similarly it checks whether $w$ is infeasible for each triple where $b = \mathrm{F}$. 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 characteristic sample 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 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 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".

## 7 Experiments

We ran all experiments on a cluster with Intel Xeon E5-2620 v4 CPUs. We allocated one CPU core and 30GB of RAM.

For our experiments we randomly generated 2789 BPs with a number of states in $[2, 20]$, number of actions in $[1, 8]$ (recall that additional actions may be added to ensure that there are no hidden states). 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.

| **#states** | **%BPs** |
|---|---|
| [ 2, 5) | 29.22% |
| [ 5, 10) | 36.75% |
| [10, 15) | 21.30% |
| [15, 20] | 12.73% |

(a)

| **#words** | **%BPs** |
|---|---|
| [ 0, 25) | 22.19% |
| [25, 50) | 42.88% |
| [50, 75) | 25.74% |
| [75, 100] | 9.18% |

(b)

| **pos-ratio** | **%BPs** |
|---|---|
| [0.0, 0.10) | 26.46% |
| [0.10, 0.25) | 47.11% |
| [0.25, 0.50) | 21.55% |
| [0.50, 1.0] | 4.88% |

(c)

| **SMT(min)** | **#BPs** |
|---|---|
| [ 0, 5) | 1992 |
| [ 5, 30) | 643 |
| [30, 60] | 154 |
| **timeout** | 291 |

(d)

Table 1: Statistical information

Table 1a provides 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

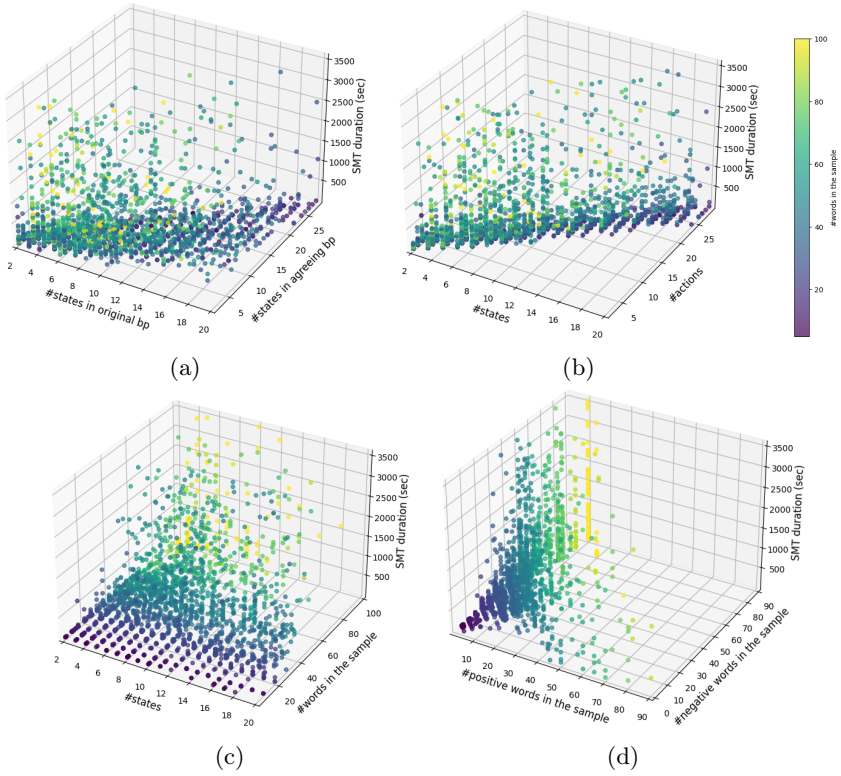


Fig. 6: SMT duration according to several parameters

850 generated random samples we used RWGen to randomly generate words. As discussed in Sec. 4, words generated by RWGen are more likely to be negative, and the probability increases with the length of words. For this reason 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 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 on 64.67% of the examples it terminated in less that 5 minutes, and it timed out on about 9.45%. Note that Tables 1a to 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 all figures (6a-6d) 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 shows the number of states in the randomly generated BP, and the $y$-axis shows the number of states 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. We can also see that the SMT time is affected mostly by the number of words in the sample.

In Fig.6b the $x$-axis shows the number of states in the randomly generated BP, and the $y$-axis shows the number of actions in the randomly generated BP. 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.

In Fig.6c 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 the SMT time is mostly affected by the number of words.

In Fig.6d the $x$-axis shows the number of positive words in the sample and the $y$-axis shows the number of 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.

## 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 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 [18], rendezvous systems [27,1], or Petri nets/VASS [42,29]. In parallel, we plan to look into applications of **LeoParDS** on realistic benchmarks, which may require some of the mentioned extensions to be handled well.

## References

1. Benjamin Aminof, Tomer Kotek, Sasha Rubin, Francesco Spegni, and Helmut Veith. Parameterized model checking of rendezvous systems. In *CONCUR*, volume 8704 of *Lecture Notes in Computer Science*, pages 109–124. Springer, 2014.
2. Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

3. Dana Angluin, Timos Antonopoulos, and Dana Fisman. Query learning of derived $\omega\omega$\omega-tree languages in polynomial time. *Log. Methods Comput. Sci.*, 15(3), 2019.
4. Dana Angluin, Timos Antonopoulos, and Dana Fisman. Strongly unambiguous büchi automata are polynomially predictable with membership queries. In *28th EACSL Annual Conference on Computer Science Logic, CSL 2020, January 13-16, 2020, Barcelona, Spain*, pages 8:1–8:17, 2020.
5. Dana Angluin, Sarah Eisenstat, and Dana Fisman. Learning regular languages via alternating automata. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 3308–3314, 2015.
6. Dana Angluin and Dana Fisman. Learning regular omega languages. *Theor. Comput. Sci.*, 650:57–72, 2016.
7. George Argyros and Loris D'Antoni. The learnability of symbolic automata. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, pages 427–445, 2018.
8. Borja Balle and Mehryar Mohri. Learning weighted automata. In *Algebraic Informatics - 6th International Conference, CAI 2015, Stuttgart, Germany, September 1-4, 2015. Proceedings*, pages 1–21, 2015.
9. Amos Beimel, Francesco Bergadano, Nader H. Bshouty, Eyal Kushilevitz, and Stefano 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. León Bohn and Christof Löding. Constructing deterministic $\omega$-automata from examples by an extension of the RPNI algorithm. In *46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021, August 23-27, 2021, Tallinn, Estonia*, pages 20:1–20:18, 2021.
12. León Bohn and Christof Löding. Passive learning of deterministic büchi automata by combinations of dfas. In *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, pages 114:1–114:20, 2022.
13. Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. Angluin-style learning of NFA. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 1004–1009, 2009.
14. Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. Learning communicating automata from mscs. *IEEE Trans. Software Eng.*, 36(3):390–408, 2010.
15. Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R. Piegdon. libalf: The automata learning framework. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 360–364, 2010.
16. Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Active learning for extended finite state machines. *Formal Asp. Comput.*, 28(2):233–263, 2016.
17. Antonio Castellanos, Enrique Vidal, Miguel Angel Varó, and José Oncina. Language understanding and subsequential transducer learning. *Computer Speech & Language*, 12(3):193–228, 1998.

18. Giorgio Delzanno, Arnaud Sangnier, Riccardo Traverso, and Gianluigi 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.
19. François Denis, Aurélien Lemay, and Alain Terlutte. Residual finite state automata. In *STACS 2001, 18th Annual Symposium on Theoretical Aspects of Computer Science, Dresden, Germany, February 15-17, 2001, Proceedings*, pages 144–157, 2001.
20. E. Allen Emerson and Vineet 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.
21. Javier Esparza, Martin Leucker, and Maximilian Schlund. Learning workflow petri nets. *Fundam. Informaticae*, 113(3-4):205–228, 2011.
22. A. Farzan, Y-F. Chen, E.M. 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.
23. Dana Fisman, Hadar Frenkel, and Sandra Zilles. Inferring symbolic automata. *Log. Methods Comput. Sci.*, 19(2), 2023.
24. Dana Fisman, Noa Izsak, and Swen Jacobs. Learning broadcast protocols, 2023. Accepted to AAAI'24. `arXiv:2306.14284`.
25. Dana Fisman, Dolav Nitay, and Michal Ziv-Ukelson. Learning of structurally unambiguous probabilistic grammars. *Log. Methods Comput. Sci.*, 19(1), 2023.
26. Dana Fisman and Sagi Saadon. Learning and characterizing fully-ordered lattice automata. In *Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022, Virtual Event, October 25-28, 2022, Proceedings*, pages 266–282, 2022.
27. Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.
28. Olga Grinchtein, Martin Leucker, and Nir Piterman. Inferring network invariants automatically. In *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, pages 483–497, 2006.
29. Michel Hack. *Decidability questions for Petri Nets.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1976.
30. Marijn Heule and Sicco Verwer. Exact DFA identification using SAT solvers. In *Grammatical Inference: Theoretical Results and Applications, 10th International Colloquium, ICGI 2010, Valencia, Spain, September 13-16, 2010. Proceedings*, pages 66–79, 2010.
31. Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring canonical register automata. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, pages 251–266, 2012.
32. Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, pages 307–322, 2014.
33. Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source learnlib - A framework for active automata learning. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 487–495, 2015.

34. Pedro Garcia José Oncina. Identifying regular languages in polynomial time. In *Advances in structural and syntactic pattern recognition*, pages 99–108, 1992. doi:doi.org/10.1142/9789812797919_0007.
35. Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA, 1994.
36. Yong Li, Yu-Fang Chen, Lijun Zhang, and Depeng Liu. A novel learning algorithm for büchi automata based on family of dfas and classification trees. *Information and Computation*, 281:104678, 2021. URL: https://www.sciencedirect.com/science/article/pii/S0890540120301711, doi:https://doi.org/10.1016/j.ic.2020.104678.
37. O. Maler and A. Pnueli. On the learnability of infinitary regular sets. *Inf. Comput.*, 118(2):316–326, 1995.
38. Jakub Michaliszyn and Jan Otop. Learning infinite-word automata with loop-index queries. *Artif. Intell.*, 307:103710, 2022.
39. Anca Muscholl and Igor Walukiewicz. Active learning for sound negotiations. In *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, pages 21:1–21:12, 2022.
40. Daniel Neider and Ivan Gavran. Learning linear temporal properties. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–10, 2018.
41. Arlindo L. Oliveira and João P. Marques Silva. Efficient algorithms for the inference of minimum size dfas. *Machine Learning*, 44(1/2):93–119, 2001.
42. Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theor. Comput. Sci.*, 6:223–231, 1978.
43. Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, April 1993.
44. Rajarshi Roy, Dana Fisman, and Daniel 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. doi:10.24963/ijcai.2020/306.
45. Yasubumi Sakakibara. Learning context-free grammars using tabular representations. *Pattern Recognit.*, 38(9):1372–1383, 2005.
46. Sylvain Schmitz and Philippe Schnoebelen. The power of well-structured systems. In *CONCUR*, volume 8052 of *Lecture Notes in Computer Science*, pages 5–24. Springer, 2013.
47. Frits W. Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten 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, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, pages 223–243, 2022.