Active learning for sound negotiations[*]

Anca Muscholl
LaBRI
Bordeaux University
France
anca@labri.fr

Igor Walukiewicz
LaBRI
CNRS, Bordeaux University
France
igw@labri.fr

ABSTRACT

We present two active learning algorithms for sound deterministic negotiations. Sound deterministic negotiations are models of distributed systems, a kind of Petri nets or Zielonka automata with additional structure. We show that this additional structure allows to minimize such negotiations. The two active learning algorithms differ in the type of membership queries they use. Both have similar complexity to Angluin's L^* algorithm, in particular, the number of queries is polynomial in the size of the negotiation, and not in the number of configurations.

CCS CONCEPTS

• **Theory of computation → Distributed computing models**.

KEYWORDS

Active learning, Distributed systems, Mazurkiewicz traces

ACM Reference Format:
Anca Muscholl and Igor Walukiewicz. 2022. Active learning for sound negotiations. In *37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (LICS '22), August 2–5, 2022, Haifa, Israel.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3531130.3533342

1 INTRODUCTION

The active learning paradigm proposed by Angluin [1] is a method used by a Learner to identify an unknown language. The paradigm assumes the existence of a Teacher who can answer membership and equivalence queries. Learner can ask if a word belongs to the language being learned, or if an automaton she constructed accepts that language. This setting allows for much more efficient algorithms than passive learning, where Learner receives just a set of positive and negative examples [8]. While passive learning has high theoretical complexity [18, 31], Angluin's L^*-algorithm can learn a regular language with polynomially many queries to Teacher. Active learning algorithms have been designed for many extensions of deterministic finite automata: automata on infinite words, on trees, weighted automata, nominal automata, bi-monoids for pomset languages [2, 3, 6, 13, 22, 24, 25, 37]. Following Angluin's original algorithm, several algorithmic improvements have been proposed [19, 21, 27], implemented in learning tools [5, 20], and used in case studies [9, 17, 26, 28, 30, 33].

Learning distributed systems is a particularly promising direction. First, because most systems are distributed anyway. Second, because distributed systems exhibit the state explosion phenomenon, namely, the state space of a distributed system is often exponential in the size of the description of the system. If we could learn a distributed system in time polynomial in the size of the description, we would be using state explosion to our advantage. Put differently, knowing something about the structure of the system would allow to speed up the learning process exponentially.

The learning results cited above all rely on the existence of canonical automata, even though sometimes these automata may not be minimal. This is a main obstacle for learning distributed systems. Consider the following example that can be reproduced in many kinds of systems. Suppose we have two processes, p_1 and p_2, both executing a shared action b. It means that on executing b the two processes update their state. The goal of the two processes is to test if the number of actions b is a multiple of 15. One solution is to make p_1 count modulo 3 and p_2 to count modulo 5. Each time when the two remainders are 0 they can declare that the number of b's they have seen is divisible by 15. The sum of the number of states of the two processes is $3 + 5 = 8$. Another possibility is that p_1 stores the two lower bits of count modulo 15, and p_2 stores the two higher bits. The sum of the number of states of the two processes is $4+4 = 8$. It is clear that there is no distributed system for this language with $2 + 5$ states or with $3 + 4$ states, as the number of global states would be $2 * 5 = 10$ and $3 * 4 = 12$, respectively. Thus we have two non-isomorphic minimal solutions. But it is not clear which of the two should be considered canonical. It is hard to imagine a learning procedure that would somehow chose one solution over the other. In this paper we avoid this major obstacle. The distributed automata we learn, *sound deterministic negotiations*, cannot implement any of the two solutions. The minimal solution for negotiations has 15 nodes and resembles the minimal deterministic automaton for the language.

Negotiations are a distributed model proposed by Esparza and Desel in [14], tightly related to workflow nets [35] and free-choice Petri nets. In one sentence, this model is a graph-based representation of processes synchronizing over shared actions. Figure 1 shows a negotiation corresponding to the workflow of an editorial board, with 4 processes *NA* (new application), *TS* (technical support), *EC* (editorial board chair), *EM* (editorial board member). Actions are written in blue, for instance svote (set-up vote) is a shared action of processes *EC* and *TS*. At node $n3$ processes *TS*, *EC* have the choice between actions svote and tech. Taking jointly svote leads process *TS* to $n6$ and *EC* to $n5$. The semantics of a negotiation is a set of *executions*, namely sequences of actions

[*]Work partially funded by ANR project FREDDA (ANR-17-CE40-0013).

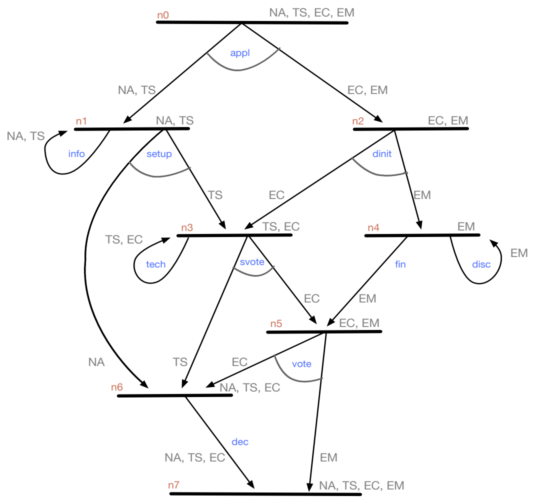

Figure 1: A sound, deterministic negotiation

that are executable from an initial to a final state. In our example, (appl)(setup)(dinit)(fin)(svote)(vote)(dec) is an execution. Executions are Mazurkiewicz traces [23] because there is a natural independence relation between actions: if the domains of two actions are disjoint, the actions are independent, and otherwise not.

Negotiations that are deterministic and sound, as the one in Figure 1, turn out to have a close relationship with finite automata. Soundness is a variant of deadlock-freedom, and determinism means that every state has at most one outgoing transition on a given label. Our first result is a canonical representation for sound deterministic negotiations by finite automata, that also provides a minimization result.

Based on this canonical representation, one could just use the standard Angluin algorithm L^* for DFA to learn sound, deterministic negotiations in polynomial time. This results in a rather unrealistic setting where Teacher is supposed to have access to the graph representation of a negotiation. When learning the negotiation from Figure 1, this setting would e.g. require Teacher to answer with a *local path* in the graph, like for example the leftmost path $(\mathsf{appl}_{TS})(\mathsf{setup}_{NA})(\mathsf{dec}_{EC})$ from $n0$ to $n7$. However, if the negotiation under learning is black-box, then equivalence queries need to be approximated by conformance testing [33]. In this case local paths are not accessible to Teacher: he can only apply executions to the system under learning. Therefore we assume in this paper that when the two negotiations are not equivalent Teacher replies with a counter-example in form of an execution that belongs to one negotiation but not to the other.

As Teacher replies with executions to equivalence queries, the main challenge is to extract some information from a counter-example execution allowing to extend the negotiation under learning. In our first algorithm Learner can ask membership queries about local paths. Membership queries about local paths are arguably difficult to justify, yet the algorithm is relatively simple and serves as a basis for the second algorithm.

Our second learning algorithm uses only executions, both for membership and for equivalence queries. With a counter-example at hand, Learner needs to be able to find a place to modify the negotiation she constructed so far. For this the negotiation needs to have enough structure to allow to build executions for membership queries. Even though this induces an important conceptual complication, the complexity of our second algorithm is comparable to that of the standard L^* algorithm for DFA. Moreover, equivalence queries in this algorithm can be done in Ptime, if the negotiation to learn is given explicitly to Teacher.

Related work. The active learning paradigm was initially designed for regular languages [1]. It is still the basis of all other learning algorithms. From the optimizations proposed in the literature [5, 19, 21, 27] we adopt two in this work. We use discriminator trees instead of rows, as this allows to gain a linear factor on the number of membership queries. We also use binary search to find a place where a modification should be made. This gives a reduction from m to $\log(m)$ membership queries to process a counterexample of size m. As it is also common by now, we add only those suffixes from a counter-example that are needed to create new states or transitions. These and some other optimizations are implemented in the TTT-algorithm [19].

There are many extensions of the active learning setting to richer models: ω-regular languages, weighted languages, nominal languages, tree languages, series-parallel pomsets [2, 3, 6, 13, 22, 25]. All of them rely on the existence of a canonical automaton for a given language. The algorithm for learning non-deterministic automata is not an exception as it learns residual finite state automata. Categorical frameworks have been recently proposed to cover the majority of these examples and provide new ones [7, 32, 38].

To our knowledge the first active learning algorithm for concurrent models is [4], where message-passing automata are learned from MSC scenarios. However, this algorithm requires a number of queries that is exponential in the number of processes and the channel bounds. Recently, an active learning algorithm for series-parallel pomsets was proposed [37]. This algorithm learns bimonoids recognizing series-parallel pomsets, which may be exponentially larger than a pomset automaton accepting the language. It relies on a representation of series-parallel pomsets as trees, and learns a tree automaton accepting the set of representations. Note that languages of deterministic sound negotiations and of series-parallel pomsets are incomparable. For example, the pomsets corresponding to executions of the negotiation from Figure 1 are not series-parallel.

Negotiations have been proposed by Esparza and Desel [10, 14]. It is a model inspired by workflow nets [34, 35] but using processes like in Mazurkiewicz trace theory and Zielonka automata [11, 23, 39]. Workflow nets have been studied extensively, in particular variants of black-box learning [36], but we are not aware of any result about active learning of such nets.

Structure of the paper. In the next section we give an overview and the context of the paper. In Section 3 we define sound, deterministic negotiations. Section 4 presents the result on minimization. Section 5 recalls briefly Angluin's L^* algorithm. Sections 6, and 7 describe the two learning algorithms that are the main result of the paper. A full version can be found at https://arxiv.org/abs/2110.02783.

2 OVERVIEW

Before going into the technical content of our work we give a high-level overview of the key concepts and results.

A negotiation is like a finite automaton with many tokens. The behavior of a finite automaton can be described in terms of one token moving between states, that we prefer to call nodes, in the graph of the automaton. At first, the token is in the initial node. It can then take any transition outgoing from this node and move further. If the transition is labelled by b, we say that the automaton takes action b. With this view, words accepted by the automaton are sequences of actions leading the token from the initial node to a final one.

What happens if we put two tokens in the initial node? When we look at the sequences of actions that are taken we will get a shuffle of words in the language of the automaton. This is concurrency without any synchronization.

Negotiations are like finite automata with several tokens and a very simple synchronization mechanism. The number of tokens is fixed and each of them is called a process, say from a finite set *Proc*. The processes move from one node to another according to the synchronization mechanism described in the following. Every node has its (non-empty) domain $dnode : N \to 2^{Proc}$ and a set of outgoing actions. The node's domain says which processes can reach it: process p can reach only nodes n with $p \in dnode(n)$. The synchronization requirement is that *all* processes in $dnode(n)$ leave node n jointly, after choosing a common outgoing action. Taking the same action at node n means that processes from $dnode(n)$ "negotiate" which action they take jointly. As in the case of finite automata, an *execution* in a negotiation is determined by a sequence of actions labelling the transitions taken, except that now one action corresponds to a move of potentially several processes. The non-deterministic variant of this simple mechanism can simulate 1-safe Petri Nets or Zielonka automata, albeit with many deadlocks.

Recall the negotiation in Figure 1, with the four processes $Proc = \{NA, TS, EC, EM\}$. Nodes are represented by horizontal bars. The initial node is on the top, and the final one at the bottom. The domain of every node, $dnode(n)$, is indicated just above the node to the right. Actions are written in blue, with $Act = \{\mathsf{appl}, \mathsf{setup}, \dots, \mathsf{dec}\}$. From every node there are several outgoing transitions on the same action, one transition per process in the domain of the node. For example, from the initial node there is an action appl with four transitions, one for each process. We denote by appl_{TS} the transition labelled appl of process TS. Transition appl_{TS} leads TS from $n0$ to $n1$. Node $n1$ has two outgoing transitions, info and setup. Both involve the two processes NA, TS. Every transition from node n involves all processes in the domain of n.

All processes start in the initial node $n0$. After action appl processes NA, TS reach node $n1$, from where they can take action setup leading TS to node $n3$, and NA to node $n6$. In parallel processes EC, EM reach node $n2$ from where they can take action dinit, which makes EC rejoin TS in node $n3$. They can continue like this forming an execution from $n0$ to $n7$: (appl)(setup)(dinit)(fin)(svote)(vote)(dec). Observe that the order of setup and dinit is not relevant because they appear concurrently. We say that the two actions are independent because they have disjoint domains. On the other hand dinit and svote cannot be permuted because EC is in the domain of the two actions. Actions are therefore partially ordered in an execution. We write $L(\mathcal{N}) \subseteq Act^*$ for the set of all (complete) executions of negotiation $\mathcal{N}$.

More formally, actions in a negotiation are typed forming a *distributed alphabet*. Every action is assigned a set of processes participating in that action: $dom : Act \to 2^{Proc}$. Going back to our example from Figure 1: $dom(\mathsf{appl})$ is the set of all four processes, while $dom(\mathsf{setup}) = \{NA, TS\}$ and $dom(\mathsf{dinit}) = \{EC, EM\}$. For every node n and action a outgoing from n we have $dom(a) = dnode(n)$. This way executions of negotiations can be viewed as Mazurkiewicz traces [23]. As the domains of setup, dinit are disjoint the two actions are independent, so their order can be permuted: for all $u, v \in Act^*$, $u(\mathsf{setup})(\mathsf{dinit})v \in L(\mathcal{N})$ iff $u(\mathsf{dinit})(\mathsf{setup})v \in L(\mathcal{N})$.

Two negotiations $\mathcal{N}_1$, $\mathcal{N}_2$ over the same distributed alphabet are *equivalent* if $L(\mathcal{N}_1) = L(\mathcal{N}_2)$. Since we will consider negotiations without deadlocks, and our systems are deterministic, this is equivalent to the two negotiations being strongly bisimilar. The goal of active learning is to allow Learner to find a negotiation equivalent to the one known by Teacher, assuming Learner can ask membership and equivalence queries to Teacher.

Sound, deterministic negotiations. Negotiations can simulate Petri nets or Zielonka automata. The three models suffer from the main obstacle described in the introduction. For deterministic negotiations this changes when we impose soundness. A negotiation is *sound*, if every execution starting from the initial node can be extended to an execution that reaches a final node. (Without loss of generality we will assume that there is only one final node in a negotiation.) So soundness is a variant of deadlock freedom. A negotiation is *deterministic* if for every process p and action b every node has at most one outgoing edge labeled b and leading to a node with p in its domain. The negotiation from Figure 1 is sound and deterministic.

Sound deterministic negotiations have many interesting properties. While soundness looks like a semantic property, it can be decided in NLOGSPACE for deterministic negotiations [15]. Actually, soundness is characterized by forbidden patterns in the negotiation graph. Some quantitative properties of sound deterministic negotiations can be computed in PTIME, see [16]. But not everything is easy. Deciding if a given negotiation has some execution that belongs to a given regular language is PSPACE-complete.

Our results. Our first contribution is the observation that sound deterministic negotiations can be minimized. This presents prospects for Angluin-style learning, as there is a canonical object to learn. It also provides a simple polynomial-time equivalence algorithm for such negotiations.

To explain the minimization result, we need one more notion. A *local path* in a negotiation is a labelled path in the negotiation graph, for example $(\mathsf{appl}_{TS})(\mathsf{setup}_{NA})(\mathsf{dec}_{EC})$ in the negotiation from Figure 1. Since the negotiation is deterministic, the source node, the action, and the process uniquely determine the transition. We write appl_{TS} for the transition on appl of process TS. In general local paths are sequences over the alphabet $A_{dom} = \{a_p : a \in Act, p \in dom(a)\}$. We write $Paths(\mathcal{N})$ for the set of all local paths of $\mathcal{N}$ leading from the initial to the final node.

Negotiations can be minimized by simply minimizing the finite automaton for local paths, Proposition 4.4.1. This proposition suggests using Angluin-style learning for finite automata to learn sound negotiations. But this supposes that Learner asks questions about local paths, and Teacher replies with local paths as counter-examples. As already mentioned, we find it hard to justify this setting. Instead, we consider the scenario where Teacher replies with a complete execution (and not a local path).

Our first learning algorithm, Theorem 6.4, still allows Learner to ask membership queries about local paths. Admittedly, this may be not very realistic either, but the algorithm is instructive, using some concepts that are central for our second algorithm. The main challenge is how to extract from a counter-example given by Teacher some information allowing to modify a negotiation being learned. The crucial property is that when Learner runs a counter-example given by Teacher in a negotiation being learned then she can find an inconsistency in her information before the counter-example reaches a deadlock (Lemma 6.2).

In our second, main learning algorithm Learner can ask membership queries about executions, and not about local paths, Theorem 7.7. The challenge now is how to construct membership queries about executions, and how to extract useful information from the answers. In the first algorithm membership queries about local paths allowed to obtain information about the graph of the negotiation. It is not evident how to use executions to accomplish the same task. Even more so because the negotiations constructed by Learner are not necessarily sound at every stage of the learning process. Nevertheless we show that Learner is able to recover soundness just with membership queries. We use Mazurkiewicz traces of a special form to designate states of the negotiation to be learned, as well as for tests. Moreover, transitions cannot be just labelled by an action, but require trace supports. All these objects are controlled by invariants guaranteeing that Learner can always make progress. While conceptually more complex, the second algorithm has a similar estimate on the number of queries as the L^* algorithm.

3 BASIC DEFINITIONS

A *(deterministic) negotiation* describes the concurrent behavior of a set of processes. At every moment each process is in some node. A node has a domain, namely the set of processes required to execute one of its actions. If at some moment all the processes from the domain of the node are in that node, then they choose a common action (outcome) to perform. In deterministic negotiations, as the ones we consider here, the outcome determines uniquely a new node for every process.

We fix a finite set of processes *Proc*. A *distributed alphabet* is a set of actions *Act* together with a function $dom : Act \to 2^{Proc}$ telling what is the (non-empty) set of processes participating in each action. More generally, for a sequence of actions $w \in Act^*$ we write $dom(w)$ for the set of processes participating in w, so $dom(w) = \cup_{|w|_a > 0} dom(a)$.

Definition 3.1. A *negotiation diagram* over a distributed alphabet (Act, dom) is a tuple $\mathcal{N} = \langle Proc, N, dnode, Act, dom, \delta, n_{init}, n_{\text{fin}} \rangle$, where

- $Proc = \{p, q, \dots\}$ is a finite set of *processes*;
- $N = \{m, n, \dots\}$ is a finite set of *nodes*, each node n has a non-empty domain $dnode(n) \subseteq Proc$;
- n_{init} is the initial node, n_{fin} the final one, and $dnode(n_{init}) = dnode(n_{\text{fin}}) = Proc$;
- $\delta : N \times Act \times Proc \dot{\to} N$ is a partial function defining the transitions.

We also require that domains of nodes and actions match:

- if $n' = \delta(n, a, p)$ is defined then $dnode(n) = dom(a)$, $p \in dom(a) \cap dnode(n')$, and $\delta(n, a, q)$ is defined for all $q \in dom(a)$.

The *size* of $\mathcal{N}$ is $|N| + |\delta|$.

A *configuration* is a function $C : Proc \to N$ indicating for each process in which node it is. A node n is *enabled* in a configuration C if all processes from the domain of n are at node n, namely, $C(p) = n$ for all $p \in dnode(n)$. Note that any two simultaneously enabled nodes n, n' have disjoint domains, $dnode(n) \cap dnode(n') = \emptyset$. We say that a is an outgoing action from n if $\delta(n, a, p)$ is defined, denoted $a \in out(n)$. If n is enabled in C and $a \in out(n)$ then a transition to a new configuration $C \xrightarrow{a} C'$ is possible, where $C'(p) = \delta(n, a, p)$ for all $p \in dom(a)$, and $C'(p) = C(p)$ for $p \notin dom(a)$. As usual, we write $C \longrightarrow C'$ when there is some a with $C \xrightarrow{a} C'$, and $\xrightarrow{*}$ is the reflexive-transitive closure of $\longrightarrow$.

The *initial configuration* C_{init} is the one with $C_{init}(p) = n_{init}$ for all p. The *final configuration* C_{fin} is such that $C_{\text{fin}}(p) = n_{\text{fin}}$ for all p.

An *execution* is a sequence of transitions between configurations starting in the initial configuration

$$C_{init} = C_1 \xrightarrow{a_1} C_2 \xrightarrow{a_2} \dots \xrightarrow{a_i} C_{i+1} .$$

Observe that an execution is determined by a sequence of actions. A *successful execution* is one ending in C_{fin}. The language $L(\mathcal{N})$ of a negotiation is the set of successful executions, $L(\mathcal{N}) = \{w \in Act^* : C_{init} \xrightarrow{w} C_{\text{fin}}\}$.

The *graph* of $\mathcal{N}$ has the set of nodes N as vertices and edges $n \xrightarrow{(a,p)} n'$ if $n' = \delta(n, a, p)$. A *local path* is a path in this graph, and $Paths(\mathcal{N})$ denotes the set of local paths of negotiation $\mathcal{N}$, leading from the initial node n_{init} to the final node n_{fin}. W.l.o.g. we assume that each node belongs to some local path from n_{init} to n_{fin}. In a deterministic negotiation there is at most one outgoing action for every pair action/process (b, p). We prefer to write it as b_p. For example, $(\text{appl})_{NA}(\text{setup})_{NA}(\text{dec})_{EC}$ is a local path in the negotiation from Figure 1. The alphabet of local paths is then $A_{dom} = \{a_p : a \in Act, p \in dom(a)\}$. Clearly, $Paths(\mathcal{N})$ is a regular language over alphabet A_{dom}. For a sequence $w \in Act^*$ and a process p we write $w|_p$ for the *projection* of w on the set of actions having p in their domain. Note that these projections are, in particular, local paths. We often consider projections $w|_p = a_1 \dots a_k$ as words over alphabet A_{dom}, namely $(a_1)_p \dots (a_k)_p \in A^*_{dom}$. Coming back to Figure 1, the projection on NA of the complete execution (appl)(setup)(dinit)(fin)(svote)(vote)(dec) is the local path $(\text{appl})_{NA}(\text{setup})_{NA}(\text{dec})_{NA}$.

A negotiation diagram is *sound* if every execution $C_{init} \xrightarrow{*} C$ can be extended to a successful one, so $C_{init} \xrightarrow{*} C \xrightarrow{*} C_{\text{fin}}$.

A sound negotiation cannot have a deadlock, i.e., a configuration that is not final but from where no process can move. An example of a deadlock configuration is when process p is at node n_p with domain containing $\{p, q\}$, and process q is at node $n_q \neq n_p$ also with the domain containing $\{p, q\}$. Another possibility for a negotiation to be unsound is to have an execution that loops without the possibility of exiting the loop.

Sound, deterministic negotiations enjoy a lot of structure, in particular they can be decomposed hierarchically using finite automata and partial orders [16]. A notable property we will use often is that for every node n there is a unique reachable configuration in which node n is the unique enabled node:

Theorem 3.2 (Configuration $I(n)$[16])**.** Let $\mathcal{N}$ be a sound and deterministic negotiation. For every node n there exists unique configuration $I(n)$ such that node n is the only node enabled in $I(n)$.

The uniqueness property from this theorem is very powerful, whenever we have an execution $C_{init} \xrightarrow{*} C$, and n is the only node enabled in C then we know that $C = I(n)$, so we know where all the processes are.

Mazurkiewicz traces. For a given distributed alphabet (Act, dom), an equivalence relation $\approx$ on Act^* is defined as the transitive closure of $uabv \approx ubav$, for $dom(a) \cap dom(b) = \emptyset$, $u, v \in Act^*$. A *Mazurkiewicz trace* is a $\approx$-equivalence class, and a trace language is a language closed under $\approx$. Note that languages of negotiations are trace languages. We identify a word over Act with its $\approx$-equivalence class, so the trace it represents. Alternatively, a trace can be seen as a labeled partial order of a special kind. Finally let us introduce some notation about prefixes and suffixes of traces. When $w \in Act^*$, we write $\min(w)$, for the set $\{a \in Act : w \approx aw' \text{ for some } w' \in Act^*\}$ of minimal actions of w. Given $u, w \in Act^*$ we say that the u is a *trace-prefix* of w if there is some $v \in Act^*$ such that $uv \approx w$. In this case we call v a trace-suffix of w, and we denote it by $u^{-1}w$.

4 MINIMIZING NEGOTIATIONS

We show now a close connection between sound deterministic negotiations and finite automata. An interesting consequence is that sound deterministic negotiations can be minimized, and that the minimal negotiation is unique.

Here we will work with local paths as defined in Section 3. Recall that these are sequences over alphabet $A_{dom} = \{a_p : a \in Act, p \in dom(a)\}$ labelling paths in the graph of a negotiation. In particular a projection $w|_p$ of an execution w is a local path. The following simple observation about projections will be useful.

Lemma 4.1. Let $\mathcal{N}$ be a deterministic negotiation, $C \xrightarrow{u} C'$ an execution in $\mathcal{N}$, and p a process. The projection $u|_p$ of u on p is a local path in $\mathcal{N}$ from $C(p)$ to $C'(p)$.

The automata we will consider in the paper are deterministic (DFA), but incomplete. A DFA $\mathcal{A} = \langle S, A, out, \delta, s^0, F\rangle$ has S as the (finite) set of states, $\delta : S \times A \to S$ as partial transition function, and $out : S \to 2^A$ as map from states to their set of outgoing actions. Thus, $a \in out(s)$ iff $\delta(s, a)$ is defined. While out seems redundant, it is very convenient when learning incomplete automata, as we do in this paper. The next definition states a useful property of automata accepting $Paths(\mathcal{N})$.

Definition 4.2 (Dom-complete automata)**.** A finite automaton $\mathcal{A}$ over the alphabet A_{dom} is *dom-complete* if for every state s of $\mathcal{A}$ and every $a_p, a_q, b_q \in A_{dom}$:

(1) $a_p \in out(s)$ iff $a_q \in out(s)$, and
(2) if $\{a_p, b_q\} \subseteq out(s)$ then $dom(a) = dom(b)$.

Moreover, we require that $a_p \in out(s_{init})$ for some a with $dom(a) = Proc$, where s_{init} is the initial state of $\mathcal{A}$.

Remark 4.3. Observe that every trimmed DFA $\mathcal{A}$ accepting the language $Paths(\mathcal{N})$ for $\mathcal{N}$ sound and deterministic, is dom-complete, if $\mathcal{N}$ has at least one transition. (An automaton is *trimmed* if every state is reachable from the initial state and co-reachable from some final state). To see this consider a state s of $\mathcal{A}$. As $\mathcal{A}$ is trimmed, there is some $\pi \in A^*_{dom}$ with $s_0 \xrightarrow{\pi} s$. Consider $\{a_p, b_q\} \subseteq out(s)$. Once again thanks to trimness, πa_p and πb_q are prefixes of some words in $Paths(\mathcal{N})$. Since $\mathcal{N}$ is deterministic, π induces a local path in $\mathcal{N}$, from n_{init} to some node n. Hence, $dom(a) = dom(b) = dnode(n)$ by the definition of negotiation. The first property follows by a similar argument.

Let us spell out how to construct a negotiation from a dom-complete automaton. The conditions on the automaton are precisely those that make the result be a negotiation.

Definition 4.4. Let $\mathcal{A} = \langle S, A_{dom}, out, \delta_{\mathcal{A}}, s^0, s_f\rangle$ be a dom-complete DFA such that $out(s_f) = \emptyset$ for the unique final state s_f. We associate with $\mathcal{A}$ the negotiation $\mathcal{N}_{\mathcal{A}} = \langle Proc, N, dnode, Act, dom, \delta, n_{init}, n_{\text{fin}}\rangle$ where

- $N = S$, $n_{init} = s^0$, and $n_{\text{fin}} = s_f$,
- $dnode(s) = dom(a)$ if $a_p \in out(s)$ for some $a \in Act$ and $p \in Proc$; moreover, $dnode(s_f) = Proc$,
- $\delta(s, a, p) = \delta_{\mathcal{A}}(s, a_p)$ for all s, a, p.

The main result of this section says that the *minimal* automaton of $Paths(\mathcal{N})$ determines a sound deterministic negotiation.

Proposition 4.4.1. Let $\mathcal{N}$ be a sound deterministic negotiation and $\mathcal{A}$ the minimal DFA accepting $Paths(\mathcal{N})$. Then $L(\mathcal{N}) = L(\mathcal{N}_{\mathcal{A}})$. Moreover $\mathcal{N}_{\mathcal{A}}$ is deterministic and sound.

Corollary 4.4.1. Let $\mathcal{N}$ be sound and deterministic, and let $w \in Act^*$ be such that $w|_p \in Paths(\mathcal{N})$ for all $p \in Proc$. Then $w \in L(\mathcal{N})$.

Recall that for a regular language L any automaton accepting L can be mapped homomorphically to the minimal automaton of L. For deterministic, sound negotiations we have the same phenomenon, where homomorphisms map nodes to nodes, so that transitions are mapped to transitions with the same label.

Corollary 4.4.2. Let $\mathcal{N}$ be sound, deterministic, and let $\mathcal{A}$ be the minimal DFA accepting $Paths(\mathcal{N})$. Then there is a homomorphism from $\mathcal{N}$ to $\mathcal{N}_{\mathcal{A}}$.

Corollary 4.4.3. Language equivalence of sound, deterministic negotiations can be checked in Ptime.

5 ANGLUIN LEARNING FOR FINITE AUTOMATA

We briefly present a variant of Angluin's L^* learning algorithm for finite automata. Our approach is particular because it works with automata that are not necessarily complete. This will be very useful when we extend the algorithm to learn negotiations. The automata coming from negotiations, dom-complete automata as in Definition 4.2, are in general not complete.

Angluin-style learning of finite automata relies on the Myhill-Nerode equivalence relation, which in turn accounts for the unicity of the minimal DFA of a regular language. A Learner wants to compute the minimal DFA $\mathcal{A}$ of an unknown regular language $L \subseteq A^*$. For this she interacts with a Teacher by asking membership queries $w \in^? L$ and equivalence queries $L(\widetilde{\mathcal{A}}) =^? L$, for some word w or automaton $\widetilde{\mathcal{A}}$. To the first type of query Teacher replies yes or no, to the second Teacher either says yes, or provides a word that is a counterexample to the equality of the two languages.

Angluin's algorithm maintains two finite sets of words, a set $Q \subseteq A^*$ of *state words* and a set $T \subseteq A^*$ of *test words*. The sets Q, T are used to construct a deterministic candidate automaton $\widetilde{\mathcal{A}}$ for L. The elements of Q are the states of $\widetilde{\mathcal{A}}$. The set Q is prefix-closed and $\varepsilon \in Q$ is the initial state of $\widetilde{\mathcal{A}}$.

The set of words $T \subseteq A^*$ determines an equivalence relation $\equiv_T$ on A^* approximating Myhill-Nerode's right congruence $\equiv^L$ of L: $u \equiv_T v \quad$ if $\quad$ for all $t \in T, \quad ut \in L$ iff $vt \in L$. Angluin's algorithm maintains two invariants, *Uniqueness* and *Closure*.

Uniqueness for all $u, v \in Q$, if $u \equiv_T v$ then $u = v$.

Observe that if $u \equiv^L v$ then $u \equiv_T v$. So $\equiv_T$ has no more equivalence classes than the Myhill-Nerode's congruence $\equiv^L$. Since Angluin's algorithm adds at least one state in every round, the consequence of *Uniqueness* is that the number of rounds is bounded by the index of $\equiv^L$, or equivalently by the size of the minimal automaton for L.

In the original Angluin's algorithm the candidate automata maintained by Learner are complete, every state has an outgoing transition on every letter. When learning negotiations, it is more natural to work with automata that are incomplete. Because of this we have a third parameter besides Q, T, which is a mapping $out : Q \to 2^A$, telling for each state what are its outgoing transitions defined so far. The original closure condition of Angluin's algorithm now becomes:

Closure for all $u \in Q, a \in out(u)$ there exists some $v \in Q$ with $ua \equiv_T v$.

For (Q, T, out) satisfying *Uniqueness* and *Closure* we can now construct an automaton: $\widetilde{\mathcal{A}} = \langle Q, A, \delta, q_{init}, F\rangle$ with state space Q and alphabet A. The initial state is ε, and the final states of $\widetilde{\mathcal{A}}$ are the states $u \in Q \cap L$. The partial transition function $\delta : Q \times A \dot{\to} Q$ is defined by:

$$\delta(u, a) = v \quad \text{ if } \quad ua \equiv_T v \text{ and } a \in out(u)\,.$$

Thanks to *Uniqueness* there can be at most one v as above. While *Closure* guarantees that $\delta(u, a)$ is defined iff $a \in out(u)$.

The learning algorithm works as follows. Initially, $Q = T = \{\varepsilon\}$ and $out(\varepsilon) = \emptyset$. Note that (Q, T, out) satisfies *Uniqueness* and *Closure*. The algorithm proceeds in rounds. A round starts with (Q, T, out) satisfying both invariants. Learner can construct a candidate automaton $\widetilde{\mathcal{A}}$. She then asks Teacher if $\widetilde{\mathcal{A}}$ and $\mathcal{A}$ are equivalent. If yes, the algorithm stops, otherwise Teacher provides a counter-example word $w \in A^*$. It may be a positive counter-example, $w \in L \setminus L(\widetilde{\mathcal{A}})$, or a negative one, $w \in L(\widetilde{\mathcal{A}}) \setminus L$. In both cases Learner extends (Q, T, out) while preserving the invariants. Then a new round can start. For sake of completeness, the details can be found in the full version.

6 LEARNING NEGOTIATIONS WITH LOCAL QUERIES

We present our first algorithm for learning sound deterministic negotiations. This algorithm serves as intermediate step to the main learning algorithm of Section 7 that uses only executions as queries.

Recall that an execution is a sequence over *Act*; where *Act* is an alphabet of actions equipped with a domain function $dom : Act \to 2^{Proc}$. Local paths are sequences over the alphabet $A_{dom} = \{a_p : a \in Act, p \in dom(a)\}$. They correspond to paths in the graph of the negotiation.

We assume that Teacher knows a sound deterministic negotiation $\mathcal{N}$ over the distributed alphabet $(Act, dom : Act \to 2^{Proc})$. Learner wants to determine the minimal negotiation $\widetilde{\mathcal{N}}$ with $L(\widetilde{\mathcal{N}}) = L$. By Corollary 4.4.2 this minimal negotiation is $\mathcal{N}_{\mathcal{A}}$, with $\mathcal{A}$ the minimal automaton for the regular language $Paths(\mathcal{N})$. Our algorithm uses two types of queries:

- membership queries $\pi \in^? Paths(\mathcal{N})$, to which Teacher replies yes or no;
- equivalence queries: $L(\widetilde{\mathcal{N}}) =^? L(\mathcal{N})$ to which Teacher either replies yes, or gives an execution $w \in Act^*$ in the symmetric difference of $L(\widetilde{\mathcal{N}})$ and $L(\mathcal{N})$.

The structure of the algorithm will be very similar to the one for DFA from Section 5. Let us explain two new issues we need to deal with. Learner will keep a tuple (Q, T, out), with $Q, T \subseteq A^*_{dom}$ and $out : Q \to 2^{A_{dom}}$, satisfying invariants *Uniqueness* and *Closure*. This tuple defines an automaton $\widetilde{\mathcal{A}}$ as in Section 5. Learner constructs from $\widetilde{\mathcal{A}}$ a negotiation $\widetilde{\mathcal{N}}$ as in Definition 4.4. She proposes $\widetilde{\mathcal{N}}$ to Teacher, and if Teacher answers with a counter-example execution she uses it to extend (Q, T, out) and construct a new $\widetilde{\mathcal{N}}$. Compared to learning finite automata, we have two new issues. We need to impose additional invariants to obtain a dom-complete automaton $\widetilde{\mathcal{A}}$ (Definition 4.2) as this is required to construct $\widetilde{\mathcal{N}}$. More importantly, we need to find a way how to exploit a counter-example that is an execution and not a local path (Lemmas 6.2 and 6.3).

We will write L_P as shorthand for $Paths(\mathcal{N})$. Since final nodes of negotiations do not have outgoing actions, L_P is prefix-free. Said differently, all words in L_P are $\equiv_T$ equivalent as soon as $\varepsilon \in T$. In particular, there will be a unique final state (with no outgoing transitions) in the automaton $\widetilde{\mathcal{A}}$ constructed from (Q, T, out). We write $[u]_T$ for the $\equiv_T$-class of $u \in A^*_{dom}$. The learning algorithm will preserve the following invariants for a triple (Q, T, out):

Uniqueness For all $u, v \in Q$, $u \equiv_T v$ implies $u = v$.

Closure For every $u \in Q, a \in out(u)$ there exists $v \in Q$ with $ua \equiv_T v$.

Pref For every $u \in Q$ there is some $t \in T$ with $ut \in L_P$.

Domain For every $u \in Q$ and every $a_p, a_q \in A_{dom}$: $a_p \in out(s)$ iff $a_q \in out(s)$.

The first two invariants are the same as in Section 5. The third one is important to determine the domain of a node: if u can be extended to a complete path, we know one outgoing action from u, and this determines the domain of u. The last invariant is the first condition of dom-completeness (Definition 4.2). Note that *Closure* and *Pref* entail the other condition of dom-completeness: if $\{a_p, b_q\} \subseteq out(u)$ for $u \in Q$ then ua_p and ub_q are local paths because of *Closure* and

Pref; so u leads in $\mathcal{N}$ to some node n with outgoing actions a, b, hence $dom(a) = dom(b) = dnode(n)$.

Lemma 6.1. If (Q, T, out) satisfies all four invariants *Uniqueness*, *Closure*, *Pref*, *Domain* then the associated automaton $\widetilde{\mathcal{A}}$ is dom-complete, so a deterministic negotiation $\widetilde{\mathcal{N}} = \mathcal{N}_{\widetilde{\mathcal{A}}}$ can be defined, see Definition 4.4.

Henceforth we use $\widetilde{\mathcal{N}}$ to denote the negotiation $\mathcal{N}_{\widetilde{\mathcal{A}}}$ and $\widetilde{L}$ to denote the language of $\widetilde{\mathcal{N}}$.

The next two lemmas lay the ground to handle counter-examples provided by Teacher. Suppose (Q, T, out) satisfies all four invariants. Teacher replies with w in the symmetric difference of L and $\widetilde{L}$. As w is an execution, and not a local path, it can be seen as a (Mazurkiewicz) trace. We will use operations on traces introduced on page 5.

The main point of the next lemma is not stated there explicitly. An execution in a negotiation $\widetilde{\mathcal{N}}$ may reach a deadlock. The lemma says that we do not have to deal with this situation because we can look backwards either for a place where we need to add a node (*Node-mismatch*) or a transition (*Absent-trans*).

Lemma 6.2. Consider a positive counter-example $w \in L \setminus \widetilde{L}$. Let v be the maximal trace-prefix of w executable in $\widetilde{\mathcal{N}}$. So we have $\widetilde{C}_{init} \xrightarrow{v} \widetilde{C}$ in $\widetilde{\mathcal{N}}$, and no action in $\min(v^{-1}w)$ can be executed from $\widetilde{C}$. With at most $|Proc|$ membership queries Learner can determine one of the following situations:

Absent-trans: An action $b \in \min(v^{-1}w)$, a node $u \in A^*_{dom}$ of $\widetilde{\mathcal{N}}$, and a sequence $r \in Act^*$ starting with b such that for every $p \in dom(b)$:

$$u\, r|_p \in L_P \quad \text{and} \quad b_p \notin out(u)\,.$$

Node-mismatch: A process p, and a local path $\pi \in A^*_{dom}$ such that

$$v|_p\, \pi \in L_P \quad \not\Leftrightarrow \quad u\, \pi \in L_P \quad \text{with } u = \widetilde{C}(p)\,.$$

The case of negative counter-examples is much simpler, and we get the *Node-mismatch* case as in Lemma 6.2 for $\pi = \varepsilon$:

Lemma 6.3. Consider a negative counter-example $w \in \widetilde{L} \setminus L$, and let $\widetilde{C}_{init} \xrightarrow{w} \widetilde{C}$. With at most $|Proc|$ membership queries Learner can find a process p such that

$$w|_p \in L_P \quad \not\Leftrightarrow \quad u \in L_P \quad \text{for } u = \widetilde{C}(p)\,.$$

Processing counter-examples. We describe now how to deal with the two cases *Absent-trans* and *Node-mismatch* of Lemmas 6.2 and 6.3. Before we start we observe that *Pref* and *Closure* entail the following variant of *Pref*, that will be useful below:

Pref' for every $u' \in Q$ and every $a_p \in out(u')$ there is some $t \in T$ such that $u' a_p t \in L_p$.

Indeed, using *Closure* we get some $v \in Q$ with $u' a_p \equiv_T v$, and because of *Pref*, there is some t with $vt \in L_P$. So $u' a_p t \in L_P$.

Absent-trans *case.* We have some node $u \in Q$ with $ur|_p \in L_P$, for $r \in Act^*$ starting with b, and $b_p \notin out(u)$ for all $p \in dom(b)$.

For every $p \in dom(b)$, we add b_p to $out(u)$ and $(b^{-1}r)|_p$ to T. For each b_p, one at a time, we check if there is some $v \in Q$ with $ub_p \equiv_T v$. If not, we add ub_p to Q with $out(ub_p) = \emptyset$. This step preserves *Uniqueness* and *Domain*. Also *Pref* holds if ub_p is added to Q, because of $(b^{-1}r)|_p \in T$.

Finally, since T changed, *Closure* must be restored. *Closure* holds for newly added ub_p, since we set $out(ub_p) = \emptyset$. The other $u' \in Q$ are those that were there already at the beginning of the round. If $u' \neq u$ then $out(u')$ is unchanged, so *Pref'* continues to hold. For $u' = u$ we have established *Pref'* by adding $(b^{-1}r)|_p$ to T. In both cases, if for some $a_p \in out(u')$ there is no $v \in Q$ with $u' a_p \equiv_T v$ then we add $u' a_p$ to Q, and set $out(ua_p) = \emptyset$. Thanks to *Pref'*, invariant *Pref* holds after this extension. The other invariants are clearly preserved.

Node-mismatch *case.* We have a process p, a node $u \in Q$, a sequence $v \in Act^*$, and a local path π such that $v|_p\, \pi \in L_P \not\Leftrightarrow u\pi \in L_P$. Moreover $\widetilde{C}_{init} \xrightarrow{v} \widetilde{C}$ and $\widetilde{C}(p) = u$.

Let $v|_p = a_1 \dots a_k$ and $\varepsilon \xrightarrow{a_1} u_1 \dots \xrightarrow{a_k} u_k$ the run of $\widetilde{\mathcal{A}}$ on $v|_p$ (this run exists since $\widetilde{C}_{init} \xrightarrow{v} \widetilde{C}$). We have $u = u_k$ and $a_1 \dots a_k\, \pi \in L_P \not\Leftrightarrow u_k\, \pi \in L_P$. So there is some $i \in \{1, \dots, k-1\}$ such that

$$u_i\, a_{i+1} \dots a_k \pi \in L_P \not\Leftrightarrow u_{i+1}\, a_{i+2} \dots a_k \pi \in L_P$$

Such an i can be determined by binary search using $O(\log(k))$ membership queries. We add $a_{i+2} \dots a_k \pi$ to T, $u_i a_{i+1}$ to Q, and set $out(u_i a_{i+1}) = \emptyset$. The invariants *Uniqueness* and *Domain* are clearly preserved. For *Pref* note that a_{i+1} already belonged to $out(u_i)$, so thanks to *Pref'*, invariant *Pref* holds for $u_i a_{i+1}$ as well. For *Closure* we proceed as in case *Absent-trans*, by enlarging Q, if necessary.

Learning algorithm. We sum up the developments in this section in the learning algorithm shown below. The initialization step of our algorithm consists in asking Teacher an equivalence query for the empty negotiation $\mathcal{N}_\emptyset$; this is a negotiation consisting of two nodes n_{init}, n_{fin} and empty transition mapping δ. Teacher either says yes or returns a positive example $w \in L$. Note that the first action of w must involve all the processes because the domain of the initial node is the set of all processes. So $w = bw'$ for some $b \in Act$ with $dom(b) = Proc$. We initialize (Q, T, out) by setting $Q = \{\varepsilon\}$, $T = \{w|_p : p \in Proc\}$, and $out(\varepsilon) = \emptyset$. All invariants are clearly satisfied. Observe that we have here the *Absent-trans* case of Lemma 6.2 with $b \in Act$ as above, $u = v = \varepsilon$ and $r = w$.

Procedure $OUT(w, Q, T, out)$ adds missing transitions as described in case *Absent-trans*. It extends *out*, and possibly T. After calling *OUT* the invariants *Uniqueness*, *Pref*, *Domain* are satisfied. Each *OUT* is followed by *CLOS* that restores the *Closure* invariant, as also described in case *Absent-trans*. It may happen that nothing is added by *CLOS* operation. Procedure *BinS* performs a binary search and extends Q, T, out as described in the *Node-mismatch* case. After its call we are sure that *Closure* does not hold, so *CLOS* adds at least one new node. Thus in every iteration the algorithm extends at least one of *out* or Q. For the complexity of Algorithm 1, see the full version.

Theorem 6.4. Algorithm 1 actively learns sound deterministic negotiations, using membership queries on local paths and equivalence queries returning executions. It can learn a negotiation of size s using $O(s(s + |Proc| + \log(m)))$ membership queries and s equivalence queries, with m the size of the longest counter-example.

```
$(res, w) \leftarrow EquivQuery(\mathcal{N}_\emptyset)$;
if $(res = true)$ then return $\mathcal{N}_\emptyset$;
$(Q, T, out) \leftarrow (\{\varepsilon\}, \{w|_p : p \in Proc\}, out(\varepsilon) = \emptyset)$
 $OUT(w, Q, T, out)$ ;        // add missing transitions
$CLOS(Q, T, out)$ ;                   // restore Closure
while $(res = false)$ do
    $\widetilde{\mathcal{N}} \leftarrow Negotiation(Q, T, out)$;          // build $\widetilde{\mathcal{N}}$
    $(res, w) \leftarrow EquivQuery(\widetilde{\mathcal{N}})$ ;        // ask Teacher
    if $(res = true)$ then return $\widetilde{\mathcal{N}}$;
    if (Absent-trans) then $OUT(w, Q, T, out)$;     // add
     missing transitions
    if (Node-mismatch) then $BinS(w, Q, T, out)$;    // add
     new state
    $CLOS(Q, T, out)$ ;                // restore Closure
end
```
Algorithm 1: Learning sound negotiations with membership queries about local paths.

It is possible to modify Algorithm 1 so that equivalence queries are asked only for $\widetilde{\mathcal{N}}$ sound. We do this in our second, main Algorithm 2. Here the presentation is clearer without this step.

7 LEARNING NEGOTIATIONS BY QUERYING EXECUTIONS

Our second learning algorithm asks membership queries about executions and not about local paths. The immediate consequence is that Q and T are built from executions and not from local paths. Executions are sequences of actions from Act, but since Act is a distributed alphabet we consider them as (Mazurkiewicz) traces. The trace structure of executions will be essential. The challenge is how to construct membership queries about executions, and how to extract useful information from the answers.

Throughout the section we fix the sound deterministic negotiation $\mathcal{N}$ we want to learn. We use the same notations as in Section 6, namely $L, \widetilde{\mathcal{N}}, \widetilde{L}$. The negotiation $\widetilde{\mathcal{N}}$ will always be deterministic, but not necessarily sound. Yet, we will show that Learner can extend it to a sound negotiation with just membership queries. So $\widetilde{\mathcal{N}}$ will be sound at every equivalence query. This greatly simplifies dealing with counter-examples.

The construction is spread over several subsections. First, we describe how we use Mazurkiewicz traces to identify nodes in a negotiation (Figure 2). Building on this we can identify transitions in negotiations. In Section 7.2 we describe our representation of nodes and transitions of a negotiation in a learning algorithm. We also state there the invariants of the construction. Section 7.3 describes two operations for extending $\widetilde{\mathcal{N}}$. They are used in Sections 7.4 and 7.5 where we show how to handle counter-examples. Section 7.5 explains how to restore soundness of $\widetilde{\mathcal{N}}$. Finally, we present a learning algorithm in Section 7.6.

7.1 Technical set-up

We describe how to use traces to talk about nodes and transitions in a negotiation. We start with a couple of definitions.

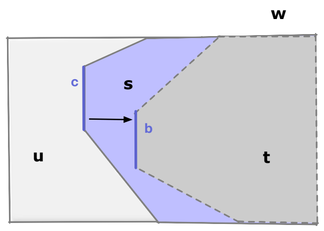

Figure 2: Partial order of execution ust. The blue part s is a (c, q)-step, of some process $q \in dom(b) \cap dom(c)$. No action of q, besides c, appears in s. Both t and st are co-prime. Actions c, b are outcomes of two nodes, and process q participates in both.

We use $u, v, w, s, r, t \in Act^*$ for sequences of actions and often consider them as partial orders, i.e., as Mazurkiewicz traces. Recall that we write $u \approx v$ when u, v represent the same Mazurkiewicz trace. For all other notations related to traces and configurations we refer to the end of Section 3. We will use extensively Theorem 3.2 stating the existence and uniqueness of the configuration $I(n)$ enabling precisely node n.

We start by defining two main kinds of traces used throughout the section (see Figure 2).

- $t \in Act^*$ is *co-prime* if t has a unique minimal element in the trace order. In other words, there is some $b \in Act$ such that every $v \in Act^*$ with $t \approx v$ starts with b. We write $b = \min(t)$ and $dmin(t)$ for the domain of $\min(t)$, namely, $dmin(t) = dom(b)$.
- $s \in Act^*$ is a *(b, p)-step* if $p \in dom(b)$, $s = bs'$ is co-prime and b is the only action involving p in s, namely, $p \notin dom(s')$.

The next two lemmas explain the link between co-prime traces and nodes of the negotiation. Lemma 7.2 roughly says that while process q goes from node m to node n by action b, the remaining processes execute u, after which n is the unique executable node. See also Figure 2 for an illustration.

Lemma 7.1. If $ut \in L$ and t is co-prime then:

- $C_{init} \xrightarrow{u} C$ is an execution of $\mathcal{N}$ with $C = I(n)$ for some node n, and $dnode(n) = dmin(t)$.
- If $ut' \in L$ for some t' then t' is also co-prime, and $dmin(t') = dmin(t)$.

Lemma 7.2. Let $C_{init} \xrightarrow{*} I(m) \xrightarrow{u} I(n) \xrightarrow{*} C_{\text{fin}}$ be an execution of $\mathcal{N}$. We have $m \xrightarrow{(b,p)} n$ if and only if u is a (b, p)-step.

The last lemma exhibits a structural property of sound deterministic negotiations in terms of co-prime traces and (b, p)-steps.

Lemma 7.3 (Crossing Lemma). If $ws_1t_1 \in L$ and $ws_2t_2 \in L$, where t_1, t_2 are co-prime, $p \in dmin(t_1) \cap dmin(t_2)$, and s_1, s_2 are (b, p)-steps, then

- $dmin(t_1) = dmin(t_2)$,
- $ws_1t_2 \in L$.

7.2 The learned negotiation

The negotiation learned by our algorithm is built from the following sets:

- $Q \subseteq Act^*$ is a set of traces, we often call them *nodes*. There should be a unique node in Q that is also in L.
- $T \subseteq Act^*$ is a set of co-prime traces, plus the empty trace ε.
- $S : Q \times Act \times Proc \to Act^*$ is a partial function giving *supports* for transitions: if defined, $S(u, b, p)$ is a (b, p)-step.

The use of co-prime traces for T is motivated by Lemma 7.1, as runs from configurations of the form $I(n)$ are co-prime traces. The support function is new. It is a generalization of the mapping *out* from Sections 5 and 6. As described by Lemma 7.2, when a process p executes an action b reaching a new node n, other processes need also to progress until n becomes the only executable node; such a progress is a trace, and the support $S(u, b, p)$ is one such trace.

Our construction will preserve the following invariants:

Uniqueness For every $u, v \in Q$, $u \equiv_T v$ implies $u = v$.

Pref For every $u \in Q$ there is $t \in T$ such that $ut \in L$.

Domain If the support $S(u, b, p)$ is defined then $S(u, b, q)$ is defined for all $q \in dom(b)$.

Pref' If the support $S(u, b, p)$ is defined then there exists some $t \in T$ with $u\, S(u, b, p)\, t \in L$. Moreover, if $t \neq \varepsilon$ then $p \in dmin(t)$.

Closure If the support $S(u, b, p)$ is defined then there is some $v \in Q$ with $u\, S(u, a, p) \equiv_T v$.

Uniqueness and *Closure* are the basic invariants, as in Sections 5 and 6. *Domain* and *Pref* are the counterparts of the invariants in Section 6. Note that *Pref'* is not a direct consequence of *Pref* and *Closure* because it puts an additional condition on $dmin(t)$. The next lemma shows how to restore the *Closure* invariant once the other four hold.

Lemma 7.4. If a triple (Q, T, S) satisfies all invariants *Uniqueness*, *Pref*, *Domain*, *Pref'*, *Closure*, and (Q, T', S') with $T \subseteq T'$ and $S \subseteq S'$ satisfies all invariants but *Closure*, then Learner can extend Q and restore all five invariants using $O(|S|(|T' \setminus T|) + (|S' \setminus S|)|T'|)$ membership queries.

From (Q, T, S) satisfying all invariants we can construct the negotiation $\widetilde{\mathcal{N}}$ such that:

- Q is the set of nodes of $\widetilde{\mathcal{N}}$,
- $dnode(u) = dmin(t)$ if $ut \in L$ for some co-prime $t \in T$, and $dnode(u) = Proc$ if $u \in L$,
- $u \xrightarrow{(b,p)} v$ if $S(u, b, p)$ defined and $u\, S(u, b, p) \equiv_T v$,
- $n_{init} = \varepsilon$, and n_{fin} is the unique node in $Q \cap L$.

Notice the use of supports in defining transitions. We cannot simply use actions to define transitions as T contains only co-prime traces.

Lemma 7.5. For every (Q, T, S) satisfying the invariants, the negotiation $\widetilde{\mathcal{N}}$ is deterministic and satisfies the following conditions:

- The domain $dnode(u)$ is well-defined for every node $u \in Q$.
- If $S(u, b, p)$ is defined then $dnode(u) = dom(b)$.
- If $u \xrightarrow{(b,p)} v$ then $p \in dnode(u) \cap dnode(v)$.

Note that $\widetilde{\mathcal{N}}$ need not be sound. In particular, even if (Q, T, S) defines a sound negotiation, the triple (Q', T', S') obtained after an application of Lemma 7.4 may not be sound. We will see how to restore soundness in Section 7.5. Before this we describe two operations that extend T' and S'.

7.3 Two operations to extend $\widetilde{\mathcal{N}}$

In response to an equivalence query Teacher may give a counter-example that Learner then analyses in order to extend $\widetilde{\mathcal{N}}$. This is described in Sections 7.4, 7.5 that follow. Here we present two operations used in these sections to actually extend $\widetilde{\mathcal{N}}$.

Absent-trans(u, r). Suppose that we have $u \in Q$, r co-prime with $ur \in L$, but $\min(r) \notin out(u)$. Since $ur \in L$ we know that $dmin(r) = dnode(u)$ by Lemma 7.1. Let $a = \min(r)$. For every process $p \in dom(a)$, consider the decomposition $r = ar'r_p$, where $p \notin dom(r')$, and r_p is either the co-prime trace with $p \in dmin(r_p)$, or $r_p = \varepsilon$. We set $S(u, a, p) = ar'$. Since we do it for all $p \in dom(a)$, invariant *Domain* holds. We add r_p to T to satisfy invariant *Pref'*. This way we restore invariants *Uniqueness*, *Pref*, *Domain*, and *Pref'*. The *Closure* invariant can be restored by Lemma 7.4.

Target-mismatch$(u' \xrightarrow{(b,p)} u, r)$. Assume we have a transition $u' \xrightarrow{(b,p)} u$ of $\widetilde{\mathcal{N}}$ and a co-prime trace r such that $u'S(u', b, p)r \in L \not\Leftrightarrow ur \in L$. Note that $p \in dnode(u)$ because of $u' \xrightarrow{(b,p)} u$ and Lemma 7.5. Also, $p \in dmin(r)$ because either $ur \in L$ and $p \in dnode(u)$, or $u'S(u', b, p)r \in L$ and *Pref'*. We add r to T. Clearly all the invariants but *Closure* continue to hold. Since *Closure* does not hold, applying Lemma 7.4 will add at least one new node to Q. Afterwards all invariants are restored.

We end with a very useful lemma allowing to detect the *Target-mismatch* case.

Lemma 7.6. Let $\varepsilon \xrightarrow{a_1,p_1} u_1 \xrightarrow{a_2,p_2} \cdots \xrightarrow{a_k,p_k} u_k$ be a local path in $\widetilde{\mathcal{N}}$, and $s_i = S(u_{i-1}, a_i, p_i)$ be the support of the i-th transition. Let also r be a co-prime trace such that $p_k \in dmin(r)$ and $u_k r \in L \not\Leftrightarrow s_1 \dots s_k r \in L$. There exists some index i such that

$$u_{i-1}\, s_i \dots s_k r \in L \not\Leftrightarrow u_i\, s_{i+1} \dots s_k r \in L$$

Moreover $u_{i-1} \xrightarrow{a_i,p_i} u_i$ together with $s_{i+1} \dots s_k r$ is an instance of the *Target-mismatch* case. Such an index i can be found with $O(\log(k))$ membership queries.

Proof. By assumption, $u_k r \in L \not\Leftrightarrow s_1 \dots s_k r \in L$ for r co-prime with $p_k \in dmin(r)$. Setting $u_0 = \varepsilon$ we see that we cannot have $u_{i-1}\, s_i \dots s_k r \in L \Leftrightarrow u_i\, s_{i+1} \dots s_k r \in L$ for all $i = 1, \dots, k$. Finding such an i is done with binary search.

In order to have get a *Target-mismatch* case we need to verify that $s_{i+1} \dots s_k r$ is co-prime. Recall that each s_i is co-prime with minimal element a_i. Since a_i and a_{i+1} have a process in common, a_{i+1} is after a_i in $s_i s_{i+1}$, hence all elements of s_{i+1} are after a_i in $s_i s_{i+1}$. Repeating this argument we obtain that $s_i \dots s_k$ is co-prime. Finally, $s_i \dots s_k r$ is co-prime because r is co-prime and $p_k \in dmin(s_k) \cap dmin(r)$. □

Corollary 7.6.1. Let $\varepsilon \xrightarrow{a_1,p_1} u_1 \xrightarrow{a_2,p_2} \cdots \xrightarrow{a_k,p_k} u_k$ be a local path in $\widetilde{\mathcal{N}}$, and $s_i = S(u_i, a_{i+1}, p_{i+1})$ be the support of the i-th transition. If $u_k \not\equiv_T s_1 \dots s_k$ then with $O(\log(k))$ queries one can find $u_i \xrightarrow{a_i,p_i}$

u_{i+1} and $s_{i+1} \dots s_k r$ forming an instance of the *Target-mismatch* case.

7.4 Handling a negative counter-example

Suppose Teacher replies to an equivalence query with a negative counter-example to the equivalence between $\mathcal{N}$ and $\widetilde{\mathcal{N}}$:

$$w \in \widetilde{L} \setminus L\,.$$

We show how to find a *Target-mismatch* case with $O(\log(|w|))$ membership queries.

Let v_1 be the longest prefix of w executable in $\mathcal{N}$. Let us suppose first that $v_1 = w$, so $C_{init} \xrightarrow{w} C \neq C_{\text{fin}}$ in $\mathcal{N}$. Since $\mathcal{N}$ is sound there must exist some action a executable in C. Chose some $p \in dom(a)$ and consider the projection $w|_p = a_1 \dots a_k$. In $\widetilde{\mathcal{N}}$ we have a local path $\varepsilon \xrightarrow{a_1,p} u_1 \dots \xrightarrow{a_k,p} u_k = u$ and $u \in L$ by assumption ($w \in \widetilde{L}$). Let $s_i = S(u_{i-1}, a_i, p)$ be the support of the i-th transition. If $u \equiv_T s_1 \dots s_k$ then $s_1 \dots s_k \in L$. But this is impossible, as $(s_1 \dots s_k)|_p = w|_p$, so $C_{init} \xrightarrow{s_1\dots s_k} C'$ with $C'(p) = C(p)$. So $u \not\equiv_T s_1 \dots s_k$ and we obtain by Corollary 7.6.1 an instance of the *Target-mismatch* case, after adding one trace to T.

Assume now that $w = v_1 b v_2$, and chose some $p \in dom(b)$. Consider the projection $v_1|_p = a_1 \dots a_k$ and the local path $\varepsilon \xrightarrow{a_1,p} u_1 \dots \xrightarrow{a_k,p} u_k \xrightarrow{b,p} u'$ in $\widetilde{\mathcal{N}}$. Let also $s_i = S(u_i, a_{i+1}, p)$, and set $u := u_k$. By the invariants of $\widetilde{\mathcal{N}}$ there are some $t, t' \in T$ with $ut \in L$, $u't' \in L$. Also, we have $uS(u,b,p) \equiv_T u'$, so $uS(u,b,p)t' \in L$. Suppose that $u \equiv_T s_1 \dots s_k$ holds. Then $s_1 \dots s_k$ is executable in $\mathcal{N}$ because of $s_1 \dots s_n t \in L$. Consider now $t'' := S(u,b,p)t'$ and observe that $s_1 \dots s_k t'' \notin L$: if $C_{init} \xrightarrow{v_1} C$ and $C_{init} \xrightarrow{s_1\dots s_k} C'$ then $C(p) = C'(p)$, so action b is impossible in $\mathcal{N}$ after executing $s_1 \dots s_k$. Therefore we have $ut'' \in L \not\Leftrightarrow s_1 \dots s_k t'' \in L$. We can conclude by applying Lemma 7.6 to the local path $\varepsilon \xrightarrow{a_1,p} u_1 \dots \xrightarrow{a_k,p} u_k$ and t'', obtaining an instance of the *Target-mismatch* case.

7.5 Handling a positive counter-example

Consider now the case where Teacher provides a positive counter-example:

$$w \in L \setminus \widetilde{L}$$

Compared to negative counter-example case, here we need to assume that $\widetilde{\mathcal{N}}$ is sound, in order to be able to use the Crossing Lemma 7.3. We can show that Learner can determine an instance either of *Absent-trans* or of the *Target-mismatch* situation with $O(\log(|w|)$ membership queries. The details can be found in the full version.

Making $\widetilde{\mathcal{N}}$ sound

Making $\widetilde{\mathcal{N}}$ sound is important for two reasons. The first one is that we use the soundness of $\widetilde{\mathcal{N}}$ when handling positive counter-examples. The second reason is that if Learner asks equivalence queries only when $\widetilde{\mathcal{N}}$ is sound, then Teacher can answer them in PTIME, according to Cor. 4.4.3.

After handling counter-examples $\widetilde{\mathcal{N}}$ is extended as described for the cases *Absent-trans* and *Target-mismatch* in Section 7.3. These do guarantee that the result satisfies the invariants, but do not guarantee that the result is sound. In the proposition below we show how $\widetilde{\mathcal{N}}$ can be made sound by Learner using only membership queries.

We assume that $\widetilde{\mathcal{N}}$ satisfies all the invariants of Section 7.2.

A local path in $\widetilde{\mathcal{N}}$, $\pi = (a_1,p_1)\dots(a_k,p_k)$ determines nodes through which it passes $\varepsilon \xrightarrow{a_1,p_1} u_1 \xrightarrow{a_2,p_2} \cdots \xrightarrow{a_k,p_k} u_k$ in $\widetilde{\mathcal{N}}$. We write $S(\pi)$ for the trace $S(u_0,a_1,p_1)\dots S(u_{k-1},a_k,p_k)$ concatenating the supports of the transitions of π. As we have observed in Lemma 7.6 this trace is co-prime. We say that π as above is a *p-path* if $p_i = p$ for $i = 1, \dots, k$.

Proposition 7.6.1. Learner can check in PTIME if $\widetilde{\mathcal{N}}$ is sound. If the answer is no, then Learner can find either an instance of *Absent-trans* or of *Target-mismatch*, with $O(s|T| + \log(m))$ membership queries.

PROOF. We assume throughout the proof that $\mathcal{N}$ is minimal. Checking whether a deterministic negotiation is sound is NLOGSPACE-complete [15]. A negotiation is not sound if and only its graph contains one of the following patterns:

F: A local path from n_{init} to some node n, action $a \in Act$, two nodes n_1, n_2 and two processes p_1, p_2 such that
- $\{p_1,p_2\} \subseteq dom(n) \cap dom(n_1) \cap dom(n_2)$;
- for $i = 1,2$ there exists a p_i-path π_i from node $\delta(n,a,p_i)$ to node n_i; and
- π_1 and π_2 are disjoint.

C: A local path which is a cycle and has no node n on it with $dom(n)$ containing all processes occurring in the cycle; moreover this cycle is reachable.

B: A node that is reachable from n_{init} by a p-path, but has not p-path to n_{fin}.

Assume first that Learner finds some pattern of type F (fork) in $\widetilde{\mathcal{N}}$. This means that she finds some words $u, u_1, u_2 \in Q$ with $u_1 \neq u_2$, $\{p_1,p_2\} \subseteq dnode(u) \cap dnode(u_1) \cap dnode(u_2)$, and local paths $\pi, \pi_1, \pi_2 \in A^*_{dom}$ with $\varepsilon \xrightarrow{\pi} u \xrightarrow{\pi_i} u_i$, and $\pi_i \in a_{p_i} A^*_{dom}$, for $i = 1,2$. Moreover, every support in $S(\pi_1)$ is a (b,p_1)-step for some b, and every support in $S(\pi_2)$ is a (c,p_2)-step, for some c.

Consider the local paths $\varepsilon \xrightarrow{\pi} u \xrightarrow{\pi_i} u_i$. For every prefix π'_i of π_i Learner verifies if $u'_i \equiv_T S(\pi\pi'_i)$. If it is not the case then using Cor 7.6.1 she can find an instance of the *Target-mismatch* case with $O(\log(s))$ membership queries, where s is the size of $\mathcal{N}$ (s bounds the lengths of the paths $\pi\pi_1, \pi\pi_2$). The overall number of membership queries here is $O(s|T| + \log(s))$, accounting for all prefixes.

We show that the remaining case is impossible. Towards contradiction suppose $u'_i \equiv_T S(\pi\pi'_i)$ for all prefixes π'_i of π and $i = 1,2$. By invariant *Pref*, both $S(\pi\pi_1)$ and $S(\pi\pi_2)$ are executable in $\mathcal{N}$. Since every support $S(u,b,p)$ is a (b,p)-step the trace $S(\pi)$ induces the local path π in $\mathcal{N}$ from n_{init} to some node n with outcome a and both p_1,p_2 in its domain (because $S(\pi)a$ is executable in $\mathcal{N}$). Similarly, $S(\pi\pi_1)$ induces the local p_1-path π_1 in $\mathcal{N}$, from n to some node n_1 with both p_1,p_2 in its domain (because of $\{p_1,p_2\} \subseteq dnode(u_1)$ and the *Pref* invariant applied to $u_1 \in Q$). Same applies to $S(\pi\pi_2)$: it induces the local p_2-path π_2 in $\mathcal{N}$, from n to some node n_2 with both p_1,p_2 in its domain. The two paths π_1, π_2 are disjoint because the corresponding nodes in $\widetilde{\mathcal{N}}$ are $\equiv_T$-inequivalent and $\mathcal{N}$ is minimal.

Since $\mathcal{N}$ is sound this implies $n_1 = n_2$, therefore $S(\pi\pi_1) \equiv^L S(\pi\pi_2)$, so in particular $S(\pi\pi_1) \equiv_T S(\pi\pi_2)$. We obtain a contradiction to $u_1 \not\equiv_T u_2$, using our assumption $u_1 \equiv_T S(\pi\pi_1)$ and $u_2 \equiv_T S(\pi\pi_2)$.

The two remaining cases, for (C) and (B) patterns, can be found in the full version. □

7.6 Learning algorithm

We assemble all the components presented until now into a learning algorithm. We assume there is an external call $\mathit{EquivQuery}(\widetilde{\mathcal{N}})$ giving Teacher's answer to the equivalence query $L(\widetilde{\mathcal{N}}) \overset{?}{=} L(\mathcal{N})$. The answer can be either *true* or a pair of a form (pos, w), (neg, w). In the latter case w is a counter-example to the equivalence and the first component indicates if this counter-example is positive or negative. Counter-examples are handled by procedure $\mathit{BinS}(ans, Q, T, S)$. It does a binary search on a counter-example and returns an instance of *Absent-trans* or *Target-mismatch*, as described in Sections 7.4 and 7.5. The result of $\mathit{BinS}(ans, Q, T, S)$ is either a tuple (abs, u, r) for which $\mathit{Absent\text{-}trans}(u, r)$ holds, or a tuple (mt, u_1, b, p, u_2, r) for which $\mathit{Target\text{-}mismatch}(u_1, b, p, u_2, r)$ holds. The procedures *OUT* and *TRG* extend (Q, T, S) as described in Section 7.3. Then procedure *CLOS* restores invariant *Closure* as described in Lemma 7.4. Finally, $\mathit{IsSound}(\widetilde{\mathcal{N}})$ checks if $\widetilde{\mathcal{N}}$ is sound; if not, it either returns an instance of *Absent-trans* ($res = (abs, u, r)$) or of *Target-mismatch* ($res = (mt, u_1, b, p, u_2, r)$), as described in Sections 7.5.

```
Init: $ans \leftarrow \mathit{EquivQuery}(\mathcal{N}_\emptyset)$;
if $(ans = true)$ then return $\mathcal{N}_\emptyset$;
$(Q, T, out) \leftarrow (\{\varepsilon\}, \{w\}, S = \text{empty function})$ ;        //
$\mathit{OUT}(\varepsilon, ans.w, Q, T, S)$ ;                                   //
$\mathit{CLOS}(Q, T, S)$ ;                             // restore Closure
while $(ans \neq true)$ do
    $\widetilde{\mathcal{N}} \leftarrow \mathit{Negotiation}(Q, T, S)$ ;               // build $\widetilde{\mathcal{N}}$
    $ans \leftarrow \mathit{EquivQuery}(\widetilde{\mathcal{N}})$ ;                  // ask Teacher
    if $(ans = true)$ then return $\widetilde{\mathcal{N}}$;           // if OK, stop
    $res \leftarrow \mathit{BinS}(ans, Q, T, S)$ ;                      // process
    repeat
        if $res = (abs, u, r)$ then $\mathit{OUT}(u, r, Q, T, S)$;
        if $res = (mt, u_1, b, p, u_2, r)$ then
          $\mathit{TRG}(u_1, b, p, u_2, r, Q, T, S)$;
        $\mathit{CLOS}(Q, T, S)$ ;                     // restore Closure
        $res \leftarrow \mathit{IsSound}(\widetilde{\mathcal{N}})$
    until $res \neq true$;;                             // $\widetilde{\mathcal{N}}$ sound
end
```

Algorithm 2: Learning algorithm with membership queries about executions.

The set of T of test traces is extended by *OUT* and *TRG*, by one for each new transition and each new state, respectively. Thus, $|T| \leq |Q| + |S|$. Because in each iteration of the while-loop either Q or S is extended, the number of equivalence queries is at most $|Q| + |S|$. As in previous sections, to simplify the complexity bound we use just one parameter s for the size of the negotiation, namely the sum of the number of nodes and the number of transitions. By m we denote the maximal size of counter-examples.

For the membership queries we observe that:

- *CLOS* uses overall $|T||S| \in O(|S|^2)$, so $O(s^2)$ membership queries (see Lemma 7.4).
- Handling a counter-example w uses each $O(\log(|w|)$ membership queries, so overall $O(s\log(m))$.
- Making $\widetilde{\mathcal{N}}$ sound uses $O(s|T| + \log(m))$ membership queries. So the overall number here is $O(s(s^2 + \log(m)))$.

We summarize the developments of this section in the following theorem.

Theorem 7.7. Algorithm 2 actively learns sound deterministic negotiations, using membership queries on executions and equivalence queries returning executions. It can learn a negotiation of size s using $O(s(s^2 + \log(m)))$ membership queries and s equivalence queries, where m is the maximal length of counter-examples.

The complexity bound for this algorithm is roughly by a factor s bigger than that of Angluin's algorithm for finite automata. This increase is due to the part making $\widetilde{\mathcal{N}}$ sound. Observe though that each time algorithm makes $\widetilde{\mathcal{N}}$ sound, it adds at least one state or one transition, so the number of equivalence queries decreases.

8 CONCLUSIONS

We have proposed two algorithms for learning sound deterministic negotiations. Due to concurrency, negotiations can be exponentially smaller than equivalent finite automata. Yet the complexity of our algorithms, measured in the number of queries, is polynomial in the size of the negotiation, and even comparable to that of learning algorithms for finite automata.

An immediate further work is to implement the algorithms. In particular, we have not discussed how to implement equivalence queries in our active learning algorithms. If Teacher has a negotiation given explicitly then the equivalence query can be done in PTIME. In more complicated cases this task is closely related to conformance checking [12], a field developing methods to check if a system under test conforms to a given model. Examples of ingenious ways of implementing the equivalence test can be found in [29]. Extension of these methods to distributed systems, such as negotiations, is an interesting research direction.

REFERENCES

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

[2] Dana Angluin and Dana Fisman. 2016. Learning regular omega languages. *Theor. Comput. Sci.* 650, 57-72 (2016).

[3] Borja Balle and Mehryar Mohri. 2015. Learning Weighted Automata. In *Algebraic Informatics - International Conference (CAI'15) (LNCS, Vol. 9270)*. Springer, 1–21.

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

[5] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David Piegdon. 2010. libalf: The automata learning framework. In *Computer Aided Verification (CAV'10) (LNCS, Vol. 6174)*. Springer, 360–364.

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

[7] Thomas Colcombet, Daniela Petrisan, and Riccardo Stabile. 2021. Learning Automata and Transducers: A Categorical Approach. In *Computer Science Logic (CSL'21) (LIPIcs, 183)*. 15:1–15:17.

[8] Colin de la Higuera. 2010. *Grammatical Inference: Learning Automata and Grammars.* Cambridge University Press.

[9] Joeri de Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *24th USENIX Security Symposium.* USENIX Association, 193–206. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter

[10] Jörg Desel, Javier Esparza, and Philipp Hoffmann. 2019. Negotiation as concurrency primitive. *Acta Informatica* 65, 2 (2019), 93–159.

[11] Volker Diekert and Grzegorz Rozenberg (Eds.). 1995. *The Book of Traces.* World Scientific, Singapore.

[12] Rita Dorofeeva, Khaled El-Fakih, Stéphane Maag, Ana R. Cavalli, and Nina Yevtushenko. 2010. FSM-based conformance testing methods: A survey annotated with experimental evaluation. *Inf. Softw. Technol.* 52, 12 (2010), 1286–1297. https://doi.org/10.1016/j.infsof.2010.07.001

[13] Frank Drewes and Johanna Högberg. 2007. Query Learning of Regular Tree Languages: How to Avoid Dead States. *Theory of Comp. Sys.* 40, 2 (2007), 163–185.

[14] Javier Esparza and Jörg Desel. 2013. On Negotiation as Concurrency Primitive. In *CONCUR 2013 - Concurrency Theory - 24th International Conference, (CONCUR'13) (LNCS, Vol. 8052).* Springer, 440–454. https://doi.org/10.1007/978-3-642-40184-8_31

[15] Javier Esparza, Denis Kuperberg, Anca Muscholl, and Igor Walukiewicz. 2018. Soundness in negotiations. *Log. Methods Comput. Sci.* 14, 1 (2018). https://doi.org/10.23638/LMCS-14(1:4)2018

[16] Javier Esparza, Anca Muscholl, and Igor Walukiewicz. 2017. Static analysis of deterministic negotiations. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'17).* IEEE Computer Society, 1–12. https://doi.org/10.1109/LICS.2017.8005144

[17] Paul Fiterau-Brostean, Ramon Janssen, and Frits W. Vaandrager. 2016. Combining Model Learning and Model Checking to Analyze TCP Implementations. In *Computer Aided Verification - 28th International Conference (CAV'16) (LNCS, Vol. 9780).* Springer, 454–471. https://doi.org/10.1007/978-3-319-41540-6_25

[18] E. Mark Gold. 1978. Complexity of automaton identification from given data. *Information & Control* 37, 3 (1978), 302–320.

[19] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. The TTT algorithm: A redundancy-free approach to active automata learning. In *Runtime verification (RV'14) (LNCS, Vol. 8734).* Springer, 307–322.

[20] Malte Isberner, Falk Howar, and Bernhard Steffen. 2015. The open-source LearnLib – A framework for active automata learning.. In *Computer Aided Verification (CAV'15) (LNCS, Vol. 9206).* Springer, 487–495.

[21] Michael J. Kearns and Umesh V. Vazirani. 1994. *An Introduction to Computational Learning Theory.* MIT Press.

[22] Ines Marusic and James Worrell. 2015. Complexity of equivalence and learning for multiplicity tree automata. *J. Mach. Learn. Res.* 16 (2015), 2465–2500.

[23] Antoni Mazurkiewicz. 1977. *Concurrent Program Schemes and their Interpretations.* DAIMI Rep. PB 78. Aarhus University, Aarhus.

[24] Jakub Michaliszyn and Jan Otop. 2020. Learning Deterministic Automata on Infinite Words. In *European Conference on Artificial Intelligence (ECAI'20) (Frontiers in Artificial Intelligence and Applications, Vol. 325).* IOS Press, 2370–2377. https://doi.org/10.3233/FAIA200367

[25] Joshua Moerman, Matteo Sammartino, Alexandra Silva, Bartek Klin, and Michal Szynwelski. 2017. Learning nominal automata. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17).* ACM, 613–625.

[26] Daniel Neider, Rick Smetsers, Frits W. Vaandrager, and Harco Kuppens. 2018. Benchmarks for Automata Learning and Conformance Testing. In *Models, Mindsets, Meta: The What, the How, and the Why Not? - Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 11200),* Tiziana Margaria, Susanne Graf, and Kim G. Larsen (Eds.). Springer, 390–416. https://doi.org/10.1007/978-3-030-22348-9_23

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

[28] Wouter Smeenk, Joshua Moerman, Frits W. Vaandrager, and David N. Jansen. 2015. Applying Automata Learning to Embedded Control Software. In *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9407),* Michael J. Butler, Sylvain Conchon, and Fatiha Zaïdi (Eds.). Springer, 67–83. https://doi.org/10.1007/978-3-319-25423-4_5

[29] Wouter Smeenk, Joshua Moerman, Frits W. Vaandrager, and David N. Jansen. 2015. Applying Automata Learning to Embedded Control Software. In *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015 (LNCS, Vol. 9407).* Springer, 67–83. https://doi.org/10.1007/978-3-319-25423-4_5

[30] Martin Tappler, Bernhard K. Aichernig, and Roderick Bloem. 2019. Model-Based Testing IoT Communication via Active Automata Learning. *CoRR* abs/1904.07075 (2019). arXiv:1904.07075 http://arxiv.org/abs/1904.07075

[31] Boris A Trakhtenbrot and Ya. M. Barzdin. 1973. *Finite Automata: Behavior and Synthesis.* North-Holland.

[32] Henning Urbat and Lutz Schröder. 2020. Automata Learning: An Algebraic Approach. In *ACM/IEEE Symposium on Logic in Computer Science (LICS'20).* ACM, 900–914. https://doi.org/10.1145/3373718.3394775

[33] Frits W. Vaandrager. 2017. Model learning. *Commun. ACM* 60, 2 (2017), 86–95.

[34] Wil M. P. van der Aalst. 1998. The Application of Petri Nets to Workflow Management. *J. Circuits Syst. Comput.* 8, 1 (1998), 21–66. https://doi.org/10.1142/S0218126698000043

[35] Wil M. P. van der Aalst. 2016. *Process Mining – Data Science in Action* (second edition ed.). Springer-Verlag Berlin Heidelberg.

[36] Wil M. P. van der Aalst, Josep Carmona, Thomas Chatain, and Boudewijn F. van Dongen. 2019. A Tour in Process Mining: From Practice to Algorithmic Challenges. *Trans. Petri Nets Other Model. Concurr.* 14 (2019), 1–35. https://doi.org/10.1007/978-3-662-60651-3_1

[37] Gerco van Heerdt, Tobias Kappé, Jurriaan Rot, and Alexandra Silva. 2021. Learning Pomset Automata. In *Foundation of Software Science and Computation Structures (FoSSaCS'21) (LNCS, Vol. 12650).* Springer, 510–530.

[38] Gerco van Heerdt, Matteo Sammartino, and Alexandra Silva. 2017. CALF: Categorical Automata Learning Framework. In *Computer Science Logic (CSL'17) (LIPIcs, Vol. 82).* Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:24.

[39] Wieslaw Zielonka. 1987. Notes on finite asynchronous automata. *RAIRO–Theoretical Informatics and Applications* 21 (1987), 99–135.