# Active learning with inductive counter examples

**Juliette JACQUOT**
(supervisor: Daniel STAN)

Technical Report $n^o$202501-techrep-JACQUOT, January 2025
revision 1

Regular Model Checking is a framework used to verify whether an algorithm meets a given specification, known as its correctness. This framework reduces the initial problem to a language learning one, by representing each state with a word using a finite alphabet.

However, this translation to a regular language of inductive invariant set does not come without its issues. Indeed, not every combination of letters represents a reachable state or an unsafe state, so a traditional oracle may have to improvise an answer for its membership query. As a result, the language expected to be learned may not be regular which prevents the learning algorithms from terminating.

In this report, we propose the $L_{\text{ICE}}$ framework, which introduces an "unknown" membership to a language, as well as a potential inductive relation between two words with an "unknown" membership in the form of inductive counter examples.

Le Regular Model Checking est un framework utilisé pour vérifier si un algorithme répond à une spécification donnée, appelée terminaison. Ce framework réduit le problème initial en un problème d'apprentissage de langage, en représentant chaque état par un mot en utilisant un alphabet fini.

Cependant, cette transition en un langage avec un ensemble invariant inductif n'est pas sans ses problèmes. En effet, toutes les combinaisons de lettres ne représentent pas un état atteignable ou un état non sûr, donc un oracle traditionnel devrait parfois improviser une réponse à sa requête d'appartenance. Par conséquent, le langage en cours d'apprentissage ne serait pas forcément un langage régulier, ce qui empêche la terminaison des algorithmes d'apprentissage.

Dans ce rapport, nous proposons le framework $L_{\text{ICE}}$, qui introduit une réponse "inconnu" à un langage, ainsi qu'une potentielle relation inductive entre deux mots avec une appartenance "inconnue" sous la forme de contre exemples inductifs.

**Keywords**
Active learning, $L^*$, Learner, Oracle, Regular Model Checking, SAT solving

Laboratoire de Recherche de l'EPITA
14-16, rue Voltaire – FR-94276 Le Kremlin-Bicêtre CEDEX – France
Tél. +33 1 53 14 59 22 – Fax. +33 1 53 14 59 13
juliette.jacquot@epita.fr – http://www.lre.epita.fr/

## Copying this document

Copyright © 2023 LRE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just “Copying this document”, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is provided in the file COPYING.DOC.

# Contents

# Chapter 1

# Introduction

After the introduction of active learning by Dana Angluin in 1987 [4], several advances have been made on the $L^*$ algorithm, such as the $L^\#$ algorithm [23]. However, those algorithms require a perfect oracle, which is not easy to implement in practice.

Indeed, some applications of active learning can have an easy implementation of membership queries, but the implementation of their equivalence queries tend to not be straightforward. That is the case in applications of active learning in protocol modeling, where you can run a real life system to get the answer to a membership query, but equivalence queries tend to resort to random sampling.

Likewise, some other applications of active learning can have an easy implementation of equivalence queries, but not have a clearly defined answer to membership queries in all cases. That is the case for Regular Model Checking.

In order to make language learning a possibility for practical uses, several attempts to add an "unknown" membership to a language were made. Those attempts consisted of either modifying the behavior of passive learning algorithms [13, 15], or adding a SAT solving step to an active learning algorithm [7, 20].

Some potential applications of language learning with an unknown membership are separating two languages [7] and solving Regular Model Checking problems [21]. However, those applications tend to not take advantage of the context they are used in.

The case studied in this report is the application of active learning to Regular Model Checking, in order to automate its resolution. Regular Model Checking [5] is a framework used to verify whether a given algorithm meets a specification, which is known as the algorithm's correctness.

A Regular Model Checking problem consists of finding a set of states which includes all reachable states, while excluding any unsafe state. Those states can be represented as words, which reduces the problem into a language learning one. Therefore, such a problem could be solved with active learning.

However, not all states are either reachable or unsafe. Indeed, some words can represent a state which was not included in the initial Regular Model Checking problem, and therefore does not have a set behavior. This could cause an issue when solving a language learning problem, since the language expected to be learned may not be a regular language. That could cause the language learning algorithm to never terminate due to arbitrary choices.

The addition of an unknown answer to the permitted oracle answers can be used to prevent that issue, but it is possible to optimize the behavior of said unknown answer. Indeed, the solution to a Regular Model Checking problem needs to be stable through its transition function, and is therefore an inductive invariant. That information can be used to reduce the number of possibilities introduced by the unknown memberships.

So, while the addition of an unknown answer is the most general extension of the framework, it is too general in this context because it does not reflect all the information an oracle can provide to the learner.

In this report, we add the concept of unknown membership as well as inductive relationships between words of unknown membership to the $L^*$ algorithm. This is done by adding inductive counter examples to equivalence requests, as well as giving more information during membership requests.

# Chapter 2

# Preliminaries

## 2.1 Languages

First, let's fix a set of symbols $\Sigma$ as an alphabet. Any word $w \in \Sigma^*$ can be defined as either the empty word $\varepsilon$ or a combination of symbols of $\Sigma$. The concatenation of two words $u \in \Sigma^*, v \in \Sigma^*$ is written either as $uv$ or $u \cdot v$ for more clarity. Likewise, the concatenation of two languages $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Sigma^*$ is defined as such: $L_1 \cdot L_2 = \{u \cdot v, u \in L_1, v \in L_2\}$.

The set of prefixes (resp. suffixes) of a word $w \in \Sigma^*$ is defined as such:
$\text{prefixes}(w) = \{u \in \Sigma^* \mid \exists v \in \Sigma^*, w = u \cdot v\}$ (resp. $\text{suffixes}(w) = \{v \in \Sigma^* \mid \exists u \in \Sigma^*, w = u \cdot v\}$).

The length of a word $w \in \Sigma^*$ is defined as follows: $\text{len}(w) = n, n \in \mathbb{N}, w \in \Sigma^n$.

A language $L \subseteq \Sigma^*$ is a subset of words of $\Sigma^*$. When given two languages $L_1 \subseteq \Sigma^*, L_2 \subseteq \Sigma^*$, we denote the (asymmetric) difference as such: $L_1 \setminus L_2 = \{w \in L_1 \mid w \notin L_2\}$.

## 2.2 Automata

A deterministic finite automaton, also known as DFA, is a five-tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ where $Q$ is a finite set of states, $\Sigma$ is the alphabet, $\delta : Q \times \Sigma \to Q$ is the transitions between states, $q_0 \in Q$ is the starting state and $F \subseteq Q$ is the set of accepting states.

The transition $\delta$ can be extended to words with the following inductive definition:

$$\hat{\delta} : Q \times \Sigma^* \to Q$$

$$\forall q, w \in Q \times \Sigma^*, \hat{\delta}(q, w) = \begin{cases} q, & w = \varepsilon \\ \hat{\delta}(\delta(q, a), v), & w = av, a \in \Sigma, v \in \Sigma^* \end{cases}$$

Furthermore, a word $w \in \Sigma^*$ is accepted by the automaton $\mathcal{A}$ if and only if $\hat{\delta}(q_0, w) \in F$. We can therefore give the following definition to the language accepted by the automaton $\mathcal{A}$: $L(\mathcal{A}) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$.

Every regular language can be recognized by a finite automaton, so the terms "regular" and "recognized by a DFA" can be used interchangeably.

|  | $\varepsilon$ |
|---|---|
| $\varepsilon$ | $+$ |
| $A$ | $-$ |
| $B$ | $+$ |

Figure 2.1: An observation table

## 2.3 Transducer

A finite state transducer, also known as a FST, is a variation of an automaton that takes a word as an input, and gives another word as an output. A FST can be represented as a five-tuple $T = (Q, \Sigma_1 \times \Sigma_2, \delta, I, F)$ where $Q$ is a finite set of states, $\Sigma_1$ is the input alphabet, $\Sigma_2$ is the output alphabet, $\delta : Q \times \Sigma_1 \to Q \times \Sigma_2$ is the transition between states, $I \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of accepting states.

The transition $\delta$ can be extended to words with the following inductive definition:

$$\hat{\delta} : Q \times \Sigma_1^* \to Q \times \Sigma_2^*$$

$$\forall q, w \in Q \times \Sigma_1^*, \hat{\delta}(q, w) = \begin{cases} (q, \varepsilon) & w = \varepsilon \\ (q', a_2 \cdot w') & w = a_1 v, a_1 \in \Sigma_1, v \in \Sigma_1^*, (q'', a_2) = \delta(q, a_1), \\ & (q', w') = \hat{\delta}(q'', v) \end{cases}$$

Using that definition, a word $w_1 \in \Sigma_1^*$ gives an output $w_2 \in \Sigma_2^*$ is and only if $\exists q_0, q_e \in I \times F, \hat{\delta}(q_0, w_1) = (q_e, w_2)$.

## 2.4 The $L^*$ algorithm

### 2.4.1 Principle

The $L^*$ algorithm is used to learn a language as an automaton, by sending two kinds of queries to a teacher: membership and equivalence queries. The membership queries are used to determine the membership of a given word: $M : \Sigma^* \to \{+, -\}$, and the equivalence queries are used to check whether the current hypothesis automaton represents the right language.

The data learned by the $L^*$ algorithm is stored in an observation table $(S, E, T)$, where $S \in \Sigma^*$ is a set of "accessor" words, $E \in \Sigma^*$ is a set of "distinguisher" words, and
$T : (S \cup (S \cdot \Sigma)) \cdot E \to \{+, -\}$ a function such that:

$$T(w) = \begin{cases} +, & w \in L \\ -, & w \notin L \end{cases}$$

An observation table can be represented as seen in figure 2.1. Each accessor word and their successors are used for the rows, and each distinguisher word is used to define the columns. The results of the map $T$ are then placed in their respective cases.

|  | $\varepsilon$ | $A$ |
|---|---|---|
| $\varepsilon$ | $+$ | $-$ |
| $A$ | $-$ | $-$ |
| $AA$ | $-$ | $+$ |
| $B$ | $+$ | $-$ |
| $AB$ | $-$ | $-$ |
| $AAA$ | $+$ | $-$ |
| $AAB$ | $-$ | $+$ |

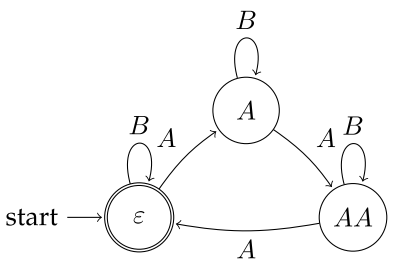

Figure 2.2: A closed and distinct observation table and its associated DFA

### 2.4.2 Filling the observation table

In order to generate an automaton from this table, it needs to meet two criteria: the table must be closed and consistent. If those criteria are not met by the observation table, the learner sends membership queries to its teacher before checking the criteria again.

A table $(S, E, T)$ is closed if $\forall w \in (S \cdot \Sigma) \setminus S,\ \exists s \in S,\ \forall e \in E,\ T(w \cdot e) = T(s \cdot e)$, meaning that each row of a successor of an accessor word must be equal to a row of an accessor word. For example, the table represented in figure 2.1 is not closed, because the row of $A$ is not equal to the row of $\varepsilon$, which is the only accessor word of the table.

A table $(S, E, T)$ is consistent if
$\forall s_1, s_2 \in S^2, (\forall e \in E, T(s_1 \cdot e) = T(s_2 \cdot e)) \Rightarrow (\forall a \in \Sigma, \forall e \in E, T(s_1 \cdot a \cdot e) = T(s_2 \cdot a \cdot e))$, meaning that if two accessors have the same rows, then the rows of their respective successors must be equal as well.
The criteria of consistence can be simplified into one of distinctness, which ensures that no accessor rows are equal: $\forall s_1, s_2 \in S^2, s_1 \neq s_2 \Rightarrow \exists e \in E,\ T(s_1 \cdot e) \neq T(s_2 \cdot e)$.

### 2.4.3 Building an automaton

In order to build an automaton representing a closed and distinct observation table $(S, E, T)$, we must determine every Myhill-Nerode equivalence class represented in said table.

The Myhill-Nerode equivalence class of a word $w \in \Sigma^*$ is noted $[w]$, and two words $w_1 \in S \cup (S \cdot \Sigma)$ and $w_2 \in S \cup (S \cdot \Sigma)$ are in the same equivalence class if $\forall e \in E, T(w_1 \cdot e) = T(w_2 \cdot e)$. That relation is noted $w_1 \equiv w_2$, and corresponds to two words having the same rows in the observation table.
Due to the table's distinctness, each accessor word can represent its own Myhill-Nerode equivalence class. The table's closedness then ensures that every other word falls into one of those equivalence classes.

The automaton representing the observation table can therefore defined as such:

$$(Q = \{[s],\ s \in S\},\quad \Sigma,\quad \delta([s], a) = [s'], s' \in S, s' \equiv s \cdot a,\quad q0 = [\varepsilon],\quad F = \{[s],\ s \in S \mid T(s) = +\})$$

A representation of such an automaton can be seen in figure 2.2.

## 2.5 Regular Model Checking

### 2.5.1 The general problem

When given a Regular Model Checking problem [5], the different states can be represented with an alphabet $\Sigma$, by associating any given state with a word $w \in \Sigma^*$. We can then define the set of starting states as a set of words $\mathrm{Init} \subseteq \Sigma^*$ and the set of unsafe states as a set of words $\mathrm{Bad} \subseteq \Sigma$.

The transitions from one state to another are represented by a function called $\mathrm{Post}$, and the inverse of that function is the function $\mathrm{Pre}$. Therefore, the set of reachable states (resp. unsafe states) is represented by $\mathrm{Post}^*(\mathrm{Init}) = \bigcup\limits_{n\in\mathbb{N}} \mathrm{Post}^n(\mathrm{Init})$ (resp. $\mathrm{Pre}^*(\mathrm{Bad}) = \bigcup\limits_{n\in\mathbb{N}} \mathrm{Pre}^n(\mathrm{Bad})$).

The Regular Model Checking problem consists of finding a language $L \subseteq \Sigma^*$, such that $L \subseteq \mathrm{Post}^*(\mathrm{Init})$, $L \cap \mathrm{Pre}^*(\mathrm{Bad}) = \emptyset$ and $\mathrm{Post}(L) \subseteq L$. This can be done by using a language learning algorithm, which can find an automaton representing a regular language meeting all those criteria.

### 2.5.2 Constraints used

When solving a Regular Model Checking problem, we must make some assumptions about the $\mathrm{Post}$ function.

- For any regular language $L \subseteq \Sigma^*$, $\mathrm{Post}(L)$ is computable as a DFA.
- Likewise, for any regular language $L \subseteq \Sigma^*$, $\mathrm{Pre}(L)$ is computable as a DFA.

In most applications of Regular Model Checking, these assumptions are often satisfied. Indeed, the transitions between states are often represented by transducers in those applications, and are therefore computable.

In order to reduce the scope of the project in this report, let's add the following constraint to the $\mathrm{Post}$ function of the Regular Model Checking problem:
The $\mathrm{Post}$ function must be length-preserving: $\forall w \in \Sigma^*, \mathrm{len}(w) = \mathrm{len}(\mathrm{Post}(w))$. This constraint is not restrictive in terms of the safety of an algorithm, because it can be bypassed by adding letters for padding.

Under these constraints, we have the following:

- For a fixed $n \in \mathbb{N}$, $\mathrm{Post}^n(L)$ is computable.
- For a fixed $n \in \mathbb{N}$, $\mathrm{Post}^*(L \cap \Sigma^n) = \mathrm{Post}^*(L) \cap \Sigma^n$ is computable because there is a finite amount of words in $\Sigma^n$.

However, $\mathrm{Post}^*(\mathrm{Init})$ is not computable. Not only is $\mathrm{Post}^*(\mathrm{Init})$ not necessarily a regular language, but the result of a $\mathrm{Post}$ function might not be regular or even computable for representations other than a DFA.

# Chapter 3

# The $L_{\mathrm{ICE}}$ Framework

## 3.1 Context

In order to solve a Regular Model Checking problem, we must find a language that is an inductive invariant, and which can prove the safety of the problem's system. A way to do so is to find a DFA which recognizes said invariant, however that process is not as straightforward as it might appear.

For example, an inductive invariant is not necessarily a regular language. The search for a regular language instead of any language is a restriction caused by the language learning algorithms used, which work by searching for an automaton and therefore a regular language.
However, this restriction is useful in the resolution of a Regular Model Checking problem. Indeed, we can only compute the $\mathrm{Post}$ of sets we can represent, which is the case of regular languages. In fact, a potential implementation of $\mathrm{Post}$ can take an automaton as an input and give another automaton as an output. Restricting the learning to a regular language is therefore helpful in the solving of the problem.

Another issue surrounding the inductive invariant is the fact that it is not unique. Indeed, the languages $\mathrm{Init}$ and $\mathrm{Bad}$ often do not contain all the words in $\Sigma^*$, which causes some words to have an undefined behavior. Due to these words, several inductive invariants can be a solution to the Regular Model Checking problem.
These multiple possibilities can make finding an oracle to use in the $L^*$ algorithm difficult, because said oracle would need to choose which invariant to learn arbitrarily. In most cases, the oracle chooses either the smallest or the biggest invariant possible, which are the languages $\mathrm{Post}^*(\mathrm{Init})$ and $\Sigma^* \setminus \mathrm{Pre}^*(\mathrm{Bad})$ respectively.
However, the oracle might make a choice of invariant which causes it to not be a regular language, which would prevent the $L^*$ algorithm from terminating.

The $L_{\mathrm{ICE}}$ framework is used to remove these arbitrary choices from the oracle, by adding the option of an unknown membership to a language. The words with such a membership can therefore change their behavior during the learning process, which prevents the language learning algorithm from being stuck trying to learn a non regular language.
This framework will use a set of criteria to choose a behavior for the words of unknown membership before giving a hypothesis automaton. These criteria include the inductive relationship between a word and its successor, and facilitate the learning of an inductive invariant

by reducing the amount of potential languages.

## 3.2 Learning criteria

In order to verify whether a language $L$ can prove the safety of a system $(\text{Init}, \text{Bad}, \text{Post})$, we check for the following criteria:

1. $\text{Init} \subseteq L$
2. $\text{Bad} \cap L = \emptyset$
3. $\text{Post}(L) \subseteq L$

According to theorems 1 and 2, if those three criteria are met, the language is a proof of the safety of the system because $\text{Post}^*(\text{Init}) \subseteq L$, $L \cap \text{Pre}^*(\text{Bad}) = \emptyset$ and $\text{Post}(L) \subseteq L$.

**Theorem 1.** *If a language is a subset of an inductive invariant, then the language of all its successors is a subset of that inductive invariant.*

*Proof.* Let $\text{Post} : 2^{\Sigma^*} \to 2^{\Sigma^*}$ a transition between words, $L \subseteq \Sigma^*, \text{Post}(L) \subseteq L$ an inductive invariant, and $\text{Init} \subseteq L$.
Let's show that $\text{Post}^*(\text{Init}) \subseteq L$ with an inductive reasoning.

$\forall n \in \mathbb{N}$, let $H_n$ be the following hypothesis: $\bigcup\limits_{i=0}^{n} \text{Post}^i(\text{Init}) \subseteq L$.

$\text{Post}^0(\text{Init}) = \text{Init}$ and $\text{Init} \subseteq L$, so the hypothesis $H_0$ is true.

Let $n \in \mathbb{N}$. Let's suppose that $H_n$ is true.

$$\begin{array}{rrll}
 & \bigcup\limits_{i=0}^{n} \text{Post}^i(\text{Init}) & \subseteq L & \\
\Rightarrow & \text{Post}\left(\bigcup\limits_{i=0}^{n} \text{Post}^i(\text{Init})\right) & \subseteq \text{Post}(L) & \\
\Rightarrow & \text{Post}\left(\bigcup\limits_{i=0}^{n} \text{Post}^i(\text{Init})\right) & \subseteq L & \text{because } \text{Post}(L) \subseteq L \\
\Rightarrow & \bigcup\limits_{i=0}^{n} \text{Post}^{i+1}(\text{Init}) & \subseteq L & \\
\Rightarrow & \bigcup\limits_{j=1}^{n+1} \text{Post}^j(\text{Init}) & \subseteq L & \text{with } j = i+1 \\
\Rightarrow & \bigcup\limits_{j=0}^{n+1} \text{Post}^j(\text{Init}) & \subseteq L & \text{because } \text{Post}^0(\text{Init}) = \text{Init} \subseteq L
\end{array}$$

Therefore, if $H_n$ is true, then $H_{n+1}$ is also true.

So, $\forall n \in \mathbb{N}$, $\bigcup\limits_{i=0}^{n} \text{Post}^i(\text{Init}) \subseteq L$.

Since $\text{Post}^*(\text{Init}) = \bigcup\limits_{i=0}^{+\infty} \text{Post}^i(\text{Init})$, we proved that $\text{Post}^*(\text{Init}) \subseteq L$. □

**Theorem 2.** *If a language is excluded from an inductive invariant, then the language of all its predecessors is excluded from that inductive invariant.*

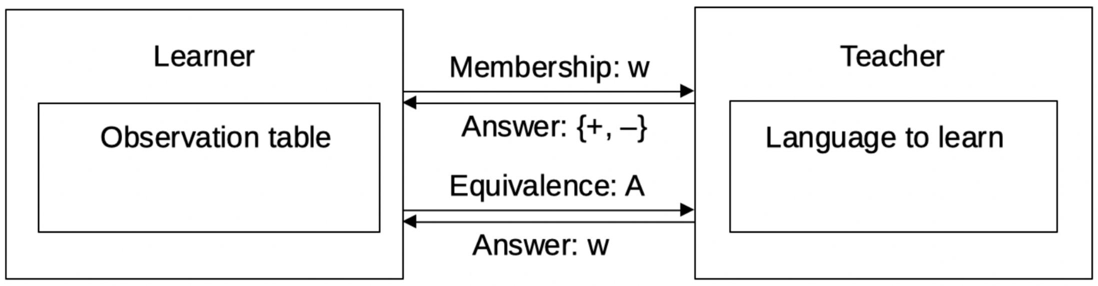

Figure 3.1: A representation of the $L^*$ algorithm

*Proof.* Let $\text{Post} : \Sigma^* \to \Sigma^*$ a transition between words, $L \subseteq \Sigma^*, \text{Post}(L) \subseteq L$ an inductive invariant, and $\text{Bad} \subseteq \Sigma^*, L \cap \text{Bad} = \emptyset$.
Let's define $\text{Pre} = \text{Post}^{-1}$.
Let's show that $L \cap \text{Pre}^*(\text{Bad}) = \emptyset$ with an inductive reasoning.

$\forall n \in \mathbb{N}$, let $H_n$ be the following hypothesis: $L \cap \left(\bigcup_{i=0}^{n} \text{Pre}^*(\text{Bad})\right) = \emptyset$.

$\text{Pre}^0(\text{Bad}) = \text{Bad}$ and $L \cap \text{Bad} = \emptyset$, so the hypothesis $H_0$ is true.

Let $n \in \mathbb{N}$. Let's suppose that $H_n$ is true, and that $\exists w \in \text{Pre}^{n+1}(\text{Bad}), w \in L$.
Then, $\text{Post}(w) \in L$ because $\text{Post}(L) \subseteq L$ and $w \in L$.
However, $\text{Post}(w) \in \text{Pre}^n(\text{Bad})$ because $w \in \text{Pre}^{n+1}(\text{Bad})$.
Therefore, $\text{Post}(w) \in L \cap \left(\bigcup_{i=0}^{n} \text{Pre}^i(\text{Bad})\right)$, which is a contradiction with the hypothesis $H_n$.
So, $L \cap \text{Pre}^{n+1}(\text{Bad}) = \emptyset$, and $L \cap \left(\bigcup_{i=0}^{n+1} \text{Pre}^i(\text{Bad})\right) = \emptyset$.
Therefore, if $H_n$ is true, then $H_{n+1}$ is also true.

So, $\forall n \in \mathbb{N}, L \cap \left(\bigcup_{i=0}^{n} \text{Pre}^*(\text{Bad})\right) = \emptyset$.
Since $\text{Pre}^*(\text{Bad}) = \bigcup_{i=0}^{+\infty} \text{Pre}^i(\text{Bad})$, we proved that $L \cap \text{Pre}^*(\text{Bad}) = \emptyset$. □

## 3.3 The $L_{\text{ICE}}$ Learner

The $L_{\text{ICE}}$ learner is based on the $L^*$ learner, represented in figure 3.1, and is used to learn a language $L$ that includes a set of accepting words $L_+$, while excluding a set of words $L_-$. The $L_{\text{ICE}}$ learner also takes into account a function $t$ to create an inductive invariant. This framework therefore adds several components to the learner.

One of those additions is the concept of an "unknown" membership. While the $L_{\text{ICE}}$ learner still has an observation table $(S, E, T)$, $T$ is redefined as follows:

$$T : (S \cup (S \cdot \Sigma)) \cdot E \to \{+, -, \square\}$$

$$T(w) = \begin{cases} +, & w \in L_+ \\ -, & w \in L_- \\ \square, & w \notin L_+ \cup L_- \end{cases}$$

Another addition is a way to keep track of potential relationships between words. To do so, the learner builds a set $U$ of words with an unknown membership, and uses said set to create a directed graph that represents the transitions between words $t : \Sigma^* \rightarrow \Sigma^*$.

These relationships between words of unknown membership can be given to the learner through both membership and validity queries, which replace the $L^*$ learner's equivalence queries. More precisely, the $L_{\text{ICE}}$ learner gives a list of words with an unknown membership during membership queries, which can then be used to find a potential relationship. During validity queries, however, the learner can receive two words as a counter example: those two words form an inductive counter example [21].

The relation between words is stored as a directed graph: we have a set of words $U \subseteq T^{-1}(\square)$, which is used to define the nodes of the directed graph. For a given word $u_1 \in U$, we can define the set $U_{u_1} \subseteq U$ of successors of $u_1$. These successors are defined such that $\forall u_2 \in U_{u_1}, u_2 \in \text{Post}^*(u_1)$.

The relationships between words of unknown membership, represented by the edges of the directed graph, are built during membership and validity requests.

For example, the learner gives the set $U$ to its teacher during membership queries alongside a word $w \in \Sigma^*$, and may receive a word $u \in U$ and a boolean $b$ in return. Depending on the value of that boolean, we either have the relation $w \in \text{Post}^*(u)$ or $w \in \text{Pre}^*(u)$

Likewise, the answer to a validity query can be two words instead of one. In that case, the learner considers the last word of the answer as the direct successor of the first.

## 3.4 The SAT Solver

Another new component to the $L_{\text{ICE}}$ learning process is needed to properly handle the words of unknown membership when generating an automaton. Indeed, a SAT solver is added to the algorithm's learner. This SAT solver is used to modify the learner's observation table to make it compatible with automaton generation.

Once the learner's observation table is filled, it is given to the SAT solver in order to remove the unknown memberships, while taking into account any inductive relation between words. Furthermore, this SAT solver can also choose the set of words it can use as the accessor words for the observation table, which can allow it to search for smaller observation tables than the one it is given, and prevents the algorithm from searching an automaton with too many states for the given problem.

### 3.4.1 SAT clauses

The SAT solver is encoded as such:

- $x_w$: the word $w \in \Sigma^*$ is accepted by the language L
- $b_w$: the word $w \in \Sigma^*$ is amongst the set of accessor words of the observation table
- $e_{s_1,a,s_2}$: the words $s_1 \cdot a$ and $s_2$ are in the same Myhill-Nerode equivalence class

We then define the following clauses to create a new closed and distinct observation table $(S', E, T')$:

$$\begin{array}{rlr}
 & b_\varepsilon & (\eta_\varepsilon) \\
\forall s \in S, \forall a \in \{a \in \Sigma |\ s \cdot a \in S\} & b_{s \cdot a} \rightarrow b_s & (\eta_{s \cdot a}) \\
\forall u_1 \in U, \forall u_2 \in U_{u_1} & x_{u_1} \rightarrow x_{u_2} & (\upsilon_{u_1,u_2}) \\
\forall s_1, s_2 \in S^2, \forall a \in \Sigma, \forall e \in E, s_2 \neq s_1 \cdot a & e_{s_1,a,s_2} \rightarrow (x_{s_1 a e} \leftrightarrow x_{s_2 e}) & (\Psi_{s_1,a,s_2,e}) \\
\forall s, s_1, s_2 \in S^3, \forall a \in \Sigma, s_1 < s_2 & e_{s,a,s_1} \rightarrow \neg e_{s,a,s_2} & (\Delta_{s,s_1,s_2,a}) \\
\forall s \in S, \forall a \in \Sigma & b_s \rightarrow \bigvee_{s' \in S} e_{s,a,s'} & (\Phi_{s,a}) \\
\forall s_1, s_2 \in S^2, \forall a \in \Sigma & e_{s_1,a,s_2} \rightarrow b_{s_2} & (\sigma_{s_1,s_2,a})
\end{array}$$

Using those clauses, we can define the new observation table $(S', E, T')$ as follows:

- $S' = \{s \in S | b_s\}$
- $T' : \begin{array}{rcl} S \cdot (S \cup \Sigma) & \longrightarrow & \{+, -\} \\ w & \longmapsto & x_w \end{array}$

Each clause is used either to shape the new observation table, or to ensure that it is closed and distinct.

The $\eta$ clauses are used to select the new set of accessor words $S' \subseteq S$, which must contain at least the word $\varepsilon$. In order to have a correct format for the observation table, a word can be an accessor word only if it is either the empty word, or if its predecessor is also an accessor word.

The $\upsilon$ clauses are used to ensure that the new function $T'$ sets the correct memberships to words that are successors or predecessors of others using the transition function. Therefore, the $\upsilon$ clauses help reduce the number of potential automata by adding more verifications to ensure that the language associated with an automaton is an inductive invariant.

The $\Psi$ clauses are used to define what a Myhill-Nerode equivalence class is in the observation table. Indeed, in the $L^*$ algorithm, two words are considered to be in the same Myhill-Nerode equivalence class if their rows are equal in the observation table.

The $\Delta$ clauses are used to ensure that the new observation table is distinct. Meanwhile, the $\Phi$ clauses are used to ensure that the new observation table is closed.

Finally, the $\sigma$ clauses are used to make sure that the new observation table is not malformed, by limiting the Myhill-Nerode equivalence classes to the set of accessor words.

The new observation table created with those SAT clauses can then be used to generate a hypothesis automaton as follows:

$$(Q = \{[s] \mid s \in S, b_s\}, \quad \Sigma, \quad \delta([s], a) = [s'], s' \in S, e_{s,a,s'}, \quad q_0 = [\varepsilon], \quad F = \{[s] \mid s \in S, x_s\})$$

### 3.4.2 UNSAT Cores

Sometimes, the learner can have a filled observation table with no possible solution. In that case, the SAT solver can track some of its clauses and check whether removing one of these clauses would be enough to find a solution. Those clauses are used to form what is called an UNSAT core.

In the case of the observation table, the SAT solver tracks all the $\Phi$ clauses and uses them to create its UNSAT core. The learner will then consider each clause present in the UNSAT core, and will add all the words $w \cdot a, w, a \in \Sigma^* \times \Sigma$ such that the clause $\Phi_{w,a}$ is one of those clauses to the set of accessor words of the observation table.

## 3.5 The ICE Teacher

The teacher has access to automata representing the languages $\mathrm{Init}$ and $\mathrm{Bad}$, as well as the transducer $\mathrm{Post}$. It can use those languages and the transducer to compute everything it needs to answer the learner's membership and equivalence queries.

The automata $\mathrm{Init}$ and $\mathrm{Bad}$ are given to the teacher directly, and can therefore be used immediately. However, the languages $\mathrm{Post}^*(\mathrm{Init})$ and $\mathrm{Pre}^*(\mathrm{Bad})$ which can be used to answer membership queries need to be computed, despite them potentially being infinite and therefore not computable.

Therefore, when given a word $w \in \Sigma^*$, we check whether $w \in \mathrm{Post}^*(\mathrm{Init})$ (resp. $w \in \mathrm{Pre}^*(\mathrm{Bad})$) by computing the languages $\mathrm{Pre}^*(w)$ (resp. $\mathrm{Post}^*(w)$), and checking if at least one of the words in the language is also in $\mathrm{Init}$ (resp. $\mathrm{Bad}$).

Unlike $\mathrm{Post}^*(\mathrm{Init})$ and $\mathrm{Pre}^*(\mathrm{Bad})$, both $\mathrm{Post}^*(w)$ and $\mathrm{Pre}^*(w)$ are finite sets for a given word $w$ because the $\mathrm{Post}$ function is length preserving, and the number of possible combinations of letters with a given alphabet is finite for any given length of word.

### 3.5.1 Membership queries

While membership queries previously only checked whether a word is in the language to learn, the ICE teacher instead looks at the relationships between words to get more specific results. This allows the teacher to specify if the word must be in the language or cannot be in the language, as well as its connection with any word of unknown membership that is given to it.

In the context the ICE teacher is used in, we are trying to find an inductive invariant which is stable through the $\mathrm{Post}$ function. Therefore, if a word of unknown membership is the successor of another, their membership must be linked.

In the context of a membership query, as seen in algorithm 1, the teacher receives both the word of the membership query and a set of words with an unknown membership. This set of words is then used to establish a relation between the current word and the ones the teacher was given.

### 3.5.2 Validity queries

In the context of a validity query, the teacher is given an automaton representing a hypothesis for a valid language. The teacher will then verify that the set $\mathrm{Init}$ is included in the language, the set $\mathrm{Bad}$ is excluded from the language, and that the language is an inductive invariant.

If one of these conditions is not met, the teacher sends either a single counter example, or a couple of words as an inductive counter example. This process is showcased in algorithm 2

Algorithm 2 works by directly manipulating the automatons and transducer it is given. Therefore, this implementation of the ICE teacher can check the validity of all the language, instead of checking the validity for all words up to a set length.

## 3.6 The other teachers

While the $L_{\mathrm{ICE}}$ algorithm is the base goal of this project, we can study some slight variations of it by using the $L_{\mathrm{ICE}}$ learner and its SAT clauses, and creating a new teacher that uses validity

**Algorithm 1** Answer a validity query

```
function MembershipAnswer(word, unknowns)
    current_post ← {word}
    current_pre ← {word}
    all_post ← {}
    all_pre ← {}
    Init_L ← GetLanguage(Init)
    Bad_L ← GetLanguage(Bad)
    while current_post $\neq$ {} and current_pre $\neq$ {} do
        if Init_L $\cap$ current_pre $\neq \emptyset$ then
            return $(+, \text{None}, \text{None})$
        else if Bad_L $\cap$ current_post $\neq \emptyset$ then
            return $(-, \text{None}, \text{None})$
        else
            for $u \in$ unknowns do
                if $u \in$ current_post then
                    return $(\text{None}, u, -)$
                else if $u \in$ current_pre then
                    return $(\text{None}, u, +)$
                end if
            end for
        end if
        all_post ← all_post $\cup$ current_post
        all_pre ← all_pre $\cup$ current_pre
        current_post ← GetPost(current_post) $\setminus$ all_post
        current_pre ← GetPre(current_pre) $\setminus$ all_pre
    end while
    return (None, None, None)
end function
```

**Algorithm 2** Answer a validity query

```
function ValidityAnswer($L$)
    if Init $\not\subseteq L$ then
        word ← GetWordIn(Init $\setminus L$)
        return (word, None)
    else if Bad $\cap L \neq \emptyset$ then
        word ← GetWordIn(Bad $\cap L$)
        return (word, None)
    else if Post($L$) $\not\subseteq L$ then
        post_word ← GetWordIn(Post($L$) $\not\subseteq L$)
        pre_words ← GetPre(post_word)
        pre_word ← GetWordIn(pre_words)
        return (pre_word, post_word)
    end if
    return (None, None)
end function
```

queries but can't give an unknown membership to a word.
This teacher uses the same algorithms as the ICE teacher, and adapts the unknown memberships or inductive counter examples to remove the concept of an unknown membership.

If a word is given an unknown membership, the teacher answers with a default membership set arbitrarily ahead of time. Likewise, if the answer to a validity query is an inductive counter example, the teacher checks the memberships of both words and chooses the one that causes a contradiction with the hypothesis automaton.

Using this new teacher, we can create two new algorithms:

- $L_{\mathrm{init}}$: uses the $L_{\mathrm{ICE}}$ learner, and sets the default membership of words to $+$. This algorithm tries to learn an invariant $L$ that is similar to $\mathrm{Post}^*(\mathrm{Init})$, but can stop before the language $\mathrm{Post}^*(\mathrm{Init})$ is found if the hypothesis automaton is a valid invariant.
- $L_{\mathrm{bad}}$: uses the $L_{\mathrm{ICE}}$ learner, and sets the default membership of words to $-$. This algorithm tries to learn an invariant $L$ that is similar to $\mathrm{Pre}^*(\mathrm{Bad})$, but can stop before the language $\Sigma^* \setminus \mathrm{Pre}^*(\mathrm{Bad})$ is found if the hypothesis automaton is a valid invariant.

# Chapter 4

# Results

## 4.1 Example: Equi-distance

In order to test our frameworks, let's take an example of a Regular Model Checking problem: the "Equi-distance" problem.

In the equi-distance problem, we have a series of tokens of an arbitrary length, as seen in figure 4.1. Each token in the series is either red, blue or empty. The red tokens are represented by the letter $A$, the blue tokens by the letter $B$, and the empty tokens by the letter $x$.
The red tokens can move to the right, and the blue tokens can move to the left. The blue and red tokens can switch places if they are next to each other, but they are considered blocked if only one empty token is found between the two. Only one pair of red and blue tokens can move at a time, and a pair of tokens can only move if there are no colored tokens between the two.

In the context of the equi-distance problem, the system has the following components:

- $\text{Init} = Ax(xx)^*B$
- $\text{Bad} = Bx^*A$

The set of reachable states is $\text{Post}^*(\text{Init}) = \{x^nAx(xx)^*Bx^n, n \in \mathbb{N}\}$, and the set of unsafe states is $\text{Pre}^*(\text{Bad}) = \{x^nBx^*Ax^n + x^nA(xx)^*Bx^n, n \in \mathbb{N}\}$.
However, this causes the languages $\text{Post}^*(\text{Init})$ and $\Sigma^* \setminus \text{Pre}^*(\text{Bad})$ to not be regular languages, meaning that the usual language learning algorithms used in Regular Model Checking cannot terminate. Still, there is at least one invariant which fulfills all the required criteria: for example, the language $x^*Ax(xx)^*Bx^*$ is a potential solution to this Regular Model Checking problem.
This problem therefore tests whether our framework can find another invariant that can be used to solve a Regular Model Checking problem.

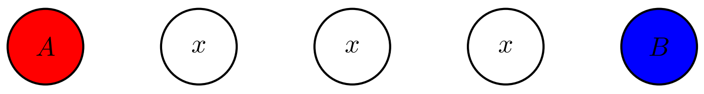

Figure 4.1: Example of a Regular Model Checking problem state

| Name | $\#_{\text{label}}$ | $S_{\text{init}}$ | $T_{\text{init}}$ | $S_{\text{bad}}$ | $T_{\text{bad}}$ | $S_{\text{post}}$ | $T_{\text{post}}$ |
|---|---|---|---|---|---|---|---|
| Bakery [10] | 3 | 3 | 3 | 3 | 9 | 5 | 19 |
| Burns [2] | 12 | 3 | 3 | 3 | 36 | 10 | 125 |
| Szymanski [22] | 11 | 9 | 9 | 13 | 40 | 118 | 412 |
| German [12] | 581 | 3 | 3 | 4 | 2112 | 17 | 9.5k |
| Dijkstra [2] | 42 | 1 | 1 | 3 | 126 | 13 | 827 |
| Dijkstra, ring [11] | 12 | 3 | 3 | 3 | 36 | 13 | 199 |
| Dining Crypto. [6] | 14 | 10 | 30 | 12 | 70 | 17 | 70 |
| Coffee Can [19] | 6 | 8 | 18 | 5 | 8 | 13 | 34 |
| Herman, linear [14] | 2 | 2 | 4 | 1 | 1 | 4 | 10 |
| Herman, ring [14] | 2 | 2 | 4 | 1 | 1 | 9 | 22 |
| Israeli-Jalfon [16] | 2 | 3 | 6 | 1 | 1 | 24 | 62 |
| Lehmann-Rabin [18] | 6 | 4 | 4 | 3 | 13 | 14 | 96 |
| LR Dining Philo. [19] | 4 | 4 | 4 | 3 | 4 | 3 | 10 |
| Mux Array [11] | 6 | 3 | 3 | 3 | 18 | 4 | 31 |
| Res. Allocator [9] | 3 | 3 | 3 | 4 | 9 | 7 | 25 |
| Kanban [3, 17] | 3 | 15 | 48 | 37 | 68 | 98 | 250 |
| Water Jugs [1] | 11 | 5 | 6 | 5 | 12 | 23 | 132 |
| Equi dist. | 3 | 4 | 4 | 3 | 3 | 9 | 18 |

Figure 4.2: Benchmark description. $\#_{\text{label}}$ stands for the size of the alphabet; $S_X$ and $T_X$ stand for the number of states and transitions in the given automata or transduce.r

## 4.2 Benchmark

In order to test our frameworks, we run each of them ten times on the benchmark[8] described in figure 4.2. The results given in figure 4.3 are the average of each run of the benchmark.

Based on the results seen in figure 4.3, we can see that the $L_{\text{ICE}}$ algorithm works as intended because a solution to the equi-distance problem was found. However, that algorithm is far from efficient, especially for problems with too many transition states.

Meanwhile, the $L_{\text{init}}$ and $L_{\text{bad}}$ algorithms are having fewer timeouts. They also tend to be faster than the $L_{\text{ICE}}$ algorithm, and do not use as many validity queries. However, we can still notice the impact of the number of states and transitions on the performance of both of these algorithms.

Indeed, almost every problem that caused a timeout for the $L_{\text{ICE}}$ algorithm still causes one for either the $L_{\text{init}}$ or the $L_{\text{bad}}$ algorithms, sometimes on both. Most of the time, the algorithm which times out is the one where its associated language has the highest number of transitions.

| RMC problem | $L_{\mathrm{ICE}}$ | | | |
|---|---|---|---|---|
| Name | T | $S_{\mathrm{inv}}$ | $M_{\mathrm{inv}}$ | $V_{\mathrm{inv}}$ |
| Bakery | 0.3 | 4.4 | 112 | 7.3 |
| Burns | 1.1 | 3.0 | 143 | 47.2 |
| Szymanski | t.o. | - | - | - |
| German | t.o. | - | - | - |
| Dijkstra | t.o. | - | - | - |
| Dijkstra, ring | t.o. | - | - | - |
| Dining Crypto. | t.o. | - | - | - |
| Coffee Can | $< 0.1$ | 3.0 | 38 | 7.2 |
| Herman, linear | $< 0.1$ | 2.0 | 5 | 2 |
| Herman, ring | $< 0.1$ | 2.0 | 5 | 1 |
| Israeli-Jalfon | $< 0.1$ | 2.0 | 11 | 5 |
| Lehmann-Rabin | 112.9 | 6.6 | 377 | 2966.3 |
| LR Dining Philo. | $< 0.1$ | 2.0 | 30 | 3.8 |
| Mux Array | 0.5 | 3.0 | 161 | 3 |
| Res. Allocator | 0.1 | 4.0 | 71 | 3 |
| Kanban | t.o. | - | - | - |
| Water Jugs | t.o. | - | - | - |
| Equi dist. | 5.5 | 4.0 | 58 | 787 |

| RMC problem | $L_{\mathrm{init}}$ | | | | $L_{\mathrm{bad}}$ | | | |
|---|---|---|---|---|---|---|---|---|
| Name | T | $S_{\mathrm{inv}}$ | $M_{\mathrm{inv}}$ | $V_{\mathrm{inv}}$ | T | $S_{\mathrm{inv}}$ | $M_{\mathrm{inv}}$ | $V_{\mathrm{inv}}$ |
| Bakery | 0.8 | 6.0 | 157 | 5 | $< 0.1$ | 4.0 | 31 | 2 |
| Burns | 5.9 | 5.0 | 143 | 3 | 0.2 | 3.0 | 87 | 2 |
| Szymanski | t.o. | - | - | - | 182.8 | 15.6 | 3949.0 | 42 |
| German | t.o. | - | - | - | t.o. | - | - | - |
| Dijkstra | 1.3 | 2.0 | 85 | 1 | t.o. | - | - | - |
| Dijkstra, ring | 515.9 | 18.0 | 3071 | 33 | 23.2 | 9.0 | 1409 | 10 |
| Dining Crypto. | t.o. | - | - | - | t.o. | - | - | - |
| Coffee Can | $< 0.1$ | 4.0 | 31 | 2 | 0.2 | 5.0 | 38 | 2 |
| Herman, linear | $< 0.1$ | 2.0 | 5 | 1 | $< 0.1$ | 2.0 | 5 | 1 |
| Herman, ring | $< 0.1$ | 2.0 | 5 | 1 | $< 0.1$ | 2.0 | 5 | 1 |
| Israeli-Jalfon | $< 0.1$ | 4.0 | 19 | 5 | $< 0.1$ | 2.0 | 5 | 1 |
| Lehmann-Rabin | 3.3 | 5.0 | 193.0 | 2 | 0.2 | 4.0 | 67 | 2 |
| LR Dining Philo. | 0.5 | 6.0 | 108 | 2 | 0.1 | 3.0 | 43 | 2 |
| Mux Array | 0.6 | 6.0 | 99 | 6 | 0.1 | 4.0 | 67 | 2 |
| Res. Allocator | 0.2 | 5.0 | 71 | 3 | 1.4 | 7.0 | 232 | 10.9 |
| Kanban | 141.7 | 33.0 | 2251 | 41 | t.o. | - | - | - |
| Water Jugs | t.o. | - | - | - | 393.8 | 29.1 | 5198.3 | 17 |
| Equi dist. | 4.1 | 10.0 | 294 | 26 | 6.9 | 9.0 | 523.3 | 68 |

Figure 4.3: Benchmark results for the $L_{\mathrm{ICE}}$, $L_{\mathrm{init}}$ and $L_{\mathrm{bad}}$ algorithms. The times are in seconds; $S_X$ and $T_X$ stand for the number of states and transitions in the learned invariant. "t.o." stands for timeout, and is used when a run lasted longer than 10 minutes.

# Chapter 5

# Future work

While the current implementations of the $L_{\text{ICE}}$, $L_{\text{init}}$ and $L_{\text{bad}}$ algorithms are working as intended, some improvements can still be made.

## 5.1 The SAT solver

The SAT clauses can be rebuilt either after each validity query, or only when we get an UNSAT core.

When the SAT solver is rebuilt only when we get an UNSAT core, like in the $L_{\text{ICE}}$ algorithm, the number of validity queries tends to skyrocket. Indeed, when a counter example is given, the learner can add one clause to its SAT solver, which causes successive validity queries to be very similar until every possible automaton is tested. With the current implementation of the framework, this causes many redundant counter examples.

In the $L_{\text{init}}$ and $L_{\text{bad}}$ algorithms, the SAT clauses are instead rebuilt after each validity query. This prevents the redundant counter examples, but can lengthen the learning process by adding too many variables when a solution could be found without them. Indeed, since several prefixes can be added whenever we find an UNSAT core, we might add some rows that do not bring more information and only add more variables to the SAT solver.

Since both membership and validity queries can be costly in the current implementation of the teacher, especially when the transitions between words are complex, we need to find a balance between the two implementations of the SAT solver.

## 5.2 Handling the counter examples

Another improvement that can be made to the learning frameworks has to do with the handling of the counter examples. Indeed, the addition of unknown memberships makes some improvements of the $L^*$ algorithm more difficult to apply in our observation table. For example, when a word is given as a counter example to a validity query, all the suffixes of that word are added to the observation table, which is far from optimal.

# Chapter 6

# Conclusion

In order to solve a Regular Model Checking problem, this report presents the $L_{\mathrm{ICE}}$ framework, an active learning algorithm based on the $L^*$ algorithm. By adding the concept of an unknown membership to a language, and inductive relation between words of unknown membership and a SAT solver, this framework is capable of solving simple Regular Model Checking problems.

This report also introduces two variations of the $L_{\mathrm{ICE}}$ framework, $L_{\mathrm{init}}$ and $L_{\mathrm{bad}}$, which try to approximate the languages $\mathrm{Post}^*(\mathrm{Init})$ and $\Sigma^* \setminus \mathrm{Pre}^*(\mathrm{Bad})$ respectively. In these frameworks, the concept of an unknown membership to a language is removed, but the concept of validity queries remains. Based on which framework is used, a solution to the Regular Model Checking problem can be found faster, and might even consist of smaller automatons than with the $L_{\mathrm{ICE}}$ algorithm.

# Chapter 7

# Bibliography

[1] (2024). Water pouring puzzle. Page Version ID: 1242579170. (page 19)

[2] Abdulla, P., Delzanno, G., Henda, N., and Rezine, A. (2007). Regular Model Checking Without Transducers (On Efficient Verification of Parameterized Systems). pages 721–736. (page 19)

[3] Abdulla, P. A., Haziza, F., and Holík, L. (2013). All for the Price of Few. In Giacobazzi, R., Berdine, J., and Mastroeni, I., editors, *Verification, Model Checking, and Abstract Interpretation*, pages 476–495, Berlin, Heidelberg. Springer. (page 19)

[4] Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106. (page 4)

[5] Bouajjani, A., Jonsson, B., Nilsson, M., and Touili, T. (2000). Regular Model Checking. In Emerson, E. A. and Sistla, A. P., editors, *Computer Aided Verification*, pages 403–418, Berlin, Heidelberg. Springer. (pages 4 and 9)

[6] Chaum, D. (1988). The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75. (page 19)

[7] Chen, Y.-F., Farzan, A., Clarke, E. M., Tsay, Y.-K., and Wang, B.-Y. (2009). Learning Minimal Separating DFA's for Compositional Verification. In Kowalewski, S. and Philippou, A., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 31–45, Berlin, Heidelberg. Springer. (page 4)

[8] Chen, Y.-F., Hong, C.-D., Lin, A. W., and Rummer, P. (2017). Learning to prove safety over parameterised concurrent systems. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 76–83, Vienna. IEEE. (page 19)

[9] Donaldson, A. F. (2007). *Automatic techniques for detecting and exploiting symmetry in model checking*. PhD, University of Glasgow. (page 19)

[10] Fokkink, W. (2013). *Distributed algorithms: an intuitive approach*. The MIT Press, Cambridge, Massachusetts. (page 19)

[11] Fribourg, L. and Olsén, H. (1997). Reachability sets of parameterized rings as regular languages. *Electronic Notes in Theoretical Computer Science*, 9:40. (page 19)

[12] German, S. M. and Sistla, A. P. (1992). Reasoning about systems with many processes. *J. ACM*, 39(3):675–735. (page 19)

[13] Grinchtein, O., Leucker, M., and Piterman, N. (2006). Inferring Network Invariants Automatically. pages 483–497. (page 4)

[14] Herman, T. (1990). Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67. (page 19)

[15] Heule, M. J. H. and Verwer, S. (2010). Exact DFA Identification Using SAT Solvers. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Sempere, J. M., and García, P., editors, *Grammatical Inference: Theoretical Results and Applications*, volume 6339, pages 66–79. Springer Berlin Heidelberg, Berlin, Heidelberg. Series Title: Lecture Notes in Computer Science. (page 4)

[16] Israeli, A. and Jalfon, M. (1990). Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 119–131, Quebec City Quebec Canada. ACM. (page 19)

[17] Kaiser, A., Kroening, D., and Wahl, T. (2010). Dynamic Cutoff Detection in Parameterized Concurrent Programs. In Touili, T., Cook, B., and Jackson, P., editors, *Computer Aided Verification*, pages 645–659, Berlin, Heidelberg. Springer. (page 19)

[18] Lehmann, D. and Rabin, M. O. (1981). On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 133–138, New York, NY, USA. Association for Computing Machinery. (page 19)

[19] Lin, A. W. and Ruemmer, P. (2016). Liveness of Randomised Parameterised Systems under Arbitrary Schedulers (Technical Report). arXiv:1606.01451 [cs]. (page 19)

[20] Moeller, M., Wiener, T., Solko-Breslin, A., Koch, C., Foster, N., and Silva, A. (2023). Automata Learning with an Incomplete Teacher. (page 4)

[21] Neider, D. (2014). Applications of automata learning in verification and synthesis. (pages 4 and 13)

[22] Szymanski, B. K. (1988). A simple solution to Lamport's concurrent programming problem with linear wait. In *Proceedings of the 2nd international conference on Supercomputing*, ICS '88, pages 621–626, New York, NY, USA. Association for Computing Machinery. (page 19)

[23] Vaandrager, F., Garhewal, B., Rot, J., and Wißmann, T. (2022). A New Approach for Active Automata Learning Based on Apartness. arXiv:2107.05419 [cs]. (page 4)