Active learning with inductive counter examples

Juliette JACQUOT
(supervisor: Daniel STAN)

Technical Report n^o202406-techrep-JACQUOT, June 2024
revision bf5e93f

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 a regular language, which prevents the learning algorithms from terminating.

In this report, we propose the L_{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_{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

Contents

Chapter 1

Introduction

When active learning was introduced by Dana Angluin in 1987 [1], it was mostly seen as a theoretical example with no practical uses. Indeed, the L^* algorithm proposed by Angluin uses a perfect oracle, meaning that the oracle knows the membership of every possible word.

Such a condition is impossible to meet, because it could require infinite memory for most languages. And while several algorithms have been created to add onto the L^* algorithm, such as the $L^\#$ algorithm [9], they still face the same issue.

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 [5, 6], or adding a SAT solving step to an active learning algorithm [3, 7].

Some potential applications of language learning with an unknown membership are separating two languages [3] and solving Regular Model Checking problems [8]. 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 [2] 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 membership to a language can be used to prevent that issue, but it is possible to optimize the behavior of said unknown membership. 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.

In this report, we add the concept of unknown membership as well as inductive relations 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 Σ as an alphabet. Any word $w \in \Sigma^*$ can be defined as either the empty word ε or a combination of symbols of Σ. 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 $L \subset \Sigma^*$ is a subset of words of Σ^*. 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, Σ 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 δ 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.

2.3 The L^* algorithm

2.3.1 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.

	ε
ε	$+$
A	$-$
B	$+$

Figure 2.1: An observation table

2.3.2 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 ε, 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: $\forall s_1, s_2 \in S^2, s_1 \neq s_2 \Rightarrow \exists e \in E,\ T(s_1 \cdot e) \neq T(s_2 \cdot e)$.

2.3.3 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.

	ε	A
ε	$+$	$-$
A	$-$	$-$
AA	$-$	$+$
B	$+$	$-$
AB	$-$	$-$
AAA	$+$	$-$
AAB	$-$	$+$

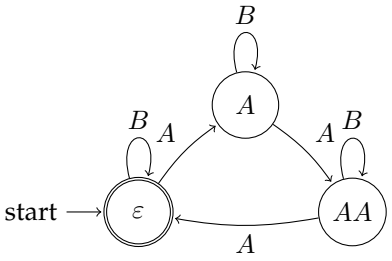

Figure 2.2: A closed and distinct observation table and its associated DFA

2.4 Regular Model Checking

2.4.1 The general problem

When given a Regular Model Checking problem [2], the different states can be represented with an alphabet Σ, 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 Post, and the inverse of that function is the function Pre. Therefore, the set of reachable states (resp. unsafe states) is represented by $\mathrm{Post}^*(\mathrm{Init}) = \bigcup_{n \in \mathbb{N}} \mathrm{Post}^n(\mathrm{Init})$ (resp. $\mathrm{Pre}^*(\mathrm{Bad}) = \bigcup_{n \in \mathbb{N}} \mathrm{Pre}^n(\mathrm{Bad})$).

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.

2.4.2 Constraints used

When solving a Regular Model Checking problem, we must make some assumptions about the Post function.

- Post is computable, meaning that for any given word $w \in \Sigma$, we are able to compute $\mathrm{Post}(w)$.
- For any regular language $L \subseteq \Sigma^*$, we can compute $\mathrm{Post}(L)$.

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 a few constraints to the Post function of the Regular Model Checking problem.

- The 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.
- The Post function must also be a bijection. Therefore, $\forall x \in \Sigma^*, \exists! y \in \Sigma^*, \mathrm{Post}(x) = y$ and $\forall y \in \Sigma^*, \exists! x \in \Sigma^*, \mathrm{Pre}(y) = x$. Contrary to the length-preservation constraint, the bijection constraint restricts the set of Regular Model Checking problems. For example, any algorithm using randomness can not be verified, because the use of randomness creates several ending states despite starting at the same state.

In some cases, a state is considered "blocked" and will not have any successors. In this case, we consider that the word itself is its own successor in our implementation. The same choice is made for words having no predecessors.

Chapter 3

Framework L_{ICE}

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, but 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 Post of sets we can represent, which is the case of regular languages. In fact, a potential implementation of 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 Init and Bad often do not contain all the words in Σ^*, 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 it would need to choose which invariant to learn arbitrarily. Furthermore, 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_{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. $\text{Post}(L) \subseteq L$

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 $\text{Post}(L) \subseteq L$, as seen in theorems 1 and 2.

Theorem 1. $\forall L \subseteq \Sigma^*, \forall\ \text{Init} \subseteq \Sigma^*, \forall\ \text{Post} : \Sigma^* \to \Sigma^*,\ (\text{Init} \subseteq L \text{ and } \text{Post}(L) \subseteq L) \Rightarrow \text{Post}^*(\text{Init}) \subseteq L$

Proof. Let's show that $\forall L \subseteq \Sigma^*$, $\text{Init} \subseteq L$ and $\text{Post}(L) \subseteq L \implies \text{Post}^*(\text{Init}) \subseteq L$ with an inductive reasoning.
Let $L \subseteq \Sigma^*$, $\text{Init} \subseteq \Sigma^*$ and $\text{Post} : \Sigma^* \to \Sigma^*$ such that $\text{Init} \subseteq L$ and $\text{Post}(L) \subseteq L$.
$\forall n \in \mathbb{N}$, let H_n be the following hypothesis: $\bigcup_{i=0}^{n} \text{Post}^i(\text{Init}) \subseteq L$.
$\text{Post}^0(\text{Init}) = \text{Init}$ 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.

$$
\begin{aligned}
\bigcup_{i=0}^{n} \text{Post}^i(\text{Init}) &\subseteq L \\
\text{Post}\left(\bigcup_{i=0}^{n} \text{Post}^i(\text{Init})\right) &\subseteq \text{Post}(L) \\
\text{Post}\left(\bigcup_{i=0}^{n} \text{Post}^i(\text{Init})\right) &\subseteq L && \text{because } \text{Post}(L) \subseteq L \\
\bigcup_{i=0}^{n} \text{Post}^{i+1}(\text{Init}) &\subseteq L \\
\bigcup_{j=1}^{n+1} \text{Post}^j(\text{Init}) &\subseteq L && \text{with } j = i+1 \\
\bigcup_{j=0}^{n+1} \text{Post}^j(\text{Init}) &\subseteq L && \text{because } \text{Post}^0(\text{Init}) = \text{Init} \subseteq L
\end{aligned}
$$

Therefore, if H_n is true, H_{n+1} is also true.
So, $\forall n \in \mathbb{N}, \bigcup_{i=0}^{n} \text{Post}^i(\text{Init}) \subseteq L$.
Since $\text{Post}^*(\text{Init}) = \bigcup_{i=0}^{+\infty} \text{Post}^i(\text{Init})$, we proved the following:

$$\forall L \subseteq \Sigma^*, \forall\ \text{Init} \subseteq \Sigma^*, \forall\ \text{Post} : \Sigma^* \to \Sigma^*,\ (\text{Init} \subseteq L \text{ and } \text{Post}(L) \subseteq L) \Rightarrow \text{Post}^*(\text{Init}) \subseteq L$$

□

Theorem 2. $\forall L \subseteq \Sigma^*, \forall\ \text{Bad} \subseteq \Sigma^*, \forall\ \text{Post} : \Sigma^* \to \Sigma^*,\ (L \cap \text{Bad} = \emptyset \text{ and } \text{Post}(L) \subseteq L) \Rightarrow L \cap \text{Pre}^*(\text{Bad}) = \emptyset$

Proof. Let's show that $\forall L \subseteq \Sigma^*$, $L \cap \text{Bad} = \emptyset$ and $\text{Post}(L) \subseteq L \implies L \cap \text{Pre}^*(\text{Bad}) = \emptyset$ with an inductive reasoning.
Let $L \subseteq \Sigma^*$, $\text{Bad} \subseteq \Sigma^*$ and $\text{Post} : \Sigma^* \to \Sigma^*$ such that $L \cap \text{Bad} = \emptyset$ and $\text{Post}(L) \subseteq L$.
$\forall n \in \mathbb{N}$, let H_n be the following hypothesis: $L \cap \left(\bigcup_{i=0}^{n} \text{Pre}^i(\text{Bad}) \right) = \emptyset$.
$\text{Pre}^0(\text{Bad}) = \text{Bad}$ 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, $w \in L \cap \left(\bigcup_{i=0}^{n} \text{Pre}^i(\text{Bad}) \right)$, which is a contradiction.
So, $L \cap \text{Pre}^{n+1}(\text{Bad}) = \emptyset$, therefore $L \cap \left(\bigcup_{i=0}^{n+1} \text{Pre}^i(\text{Bad}) \right) = \emptyset$.
Therefore, if H_n is true, H_{n+1} is also true.
So, $\forall n \in \mathbb{N}, L \cap \left(\bigcup_{i=0}^{n} \text{Pre}^i(\text{Bad}) \right) = \emptyset$
Since $\text{Pre}^*(\text{Bad}) = \bigcup_{i=0}^{+\infty} \text{Pre}^i(\text{Bad})$, we proved the following:

$$\forall L \subseteq \Sigma^*, \forall\, \text{Bad} \subseteq \Sigma^*, \forall\, \text{Post} : \Sigma^* \to \Sigma^*,\ (L \cap \text{Bad} = \emptyset \text{ and } \text{Post}(L) \subseteq L) \Rightarrow L \cap \text{Pre}^*(\text{Bad}) = \emptyset$$

□

3.3 The L_{ICE} learner

The L_{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_{ICE} learner also takes into account a function t to create an inductive invariant. This framework therefore adds several components to the learner, as seen in figure 3.2.

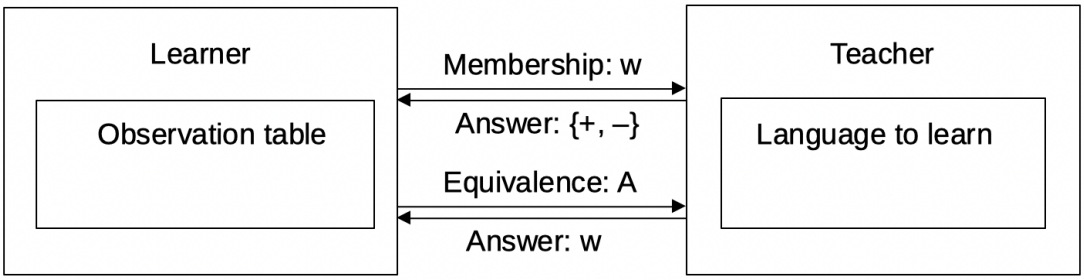

Figure 3.1: A representation of the L^* algorithm

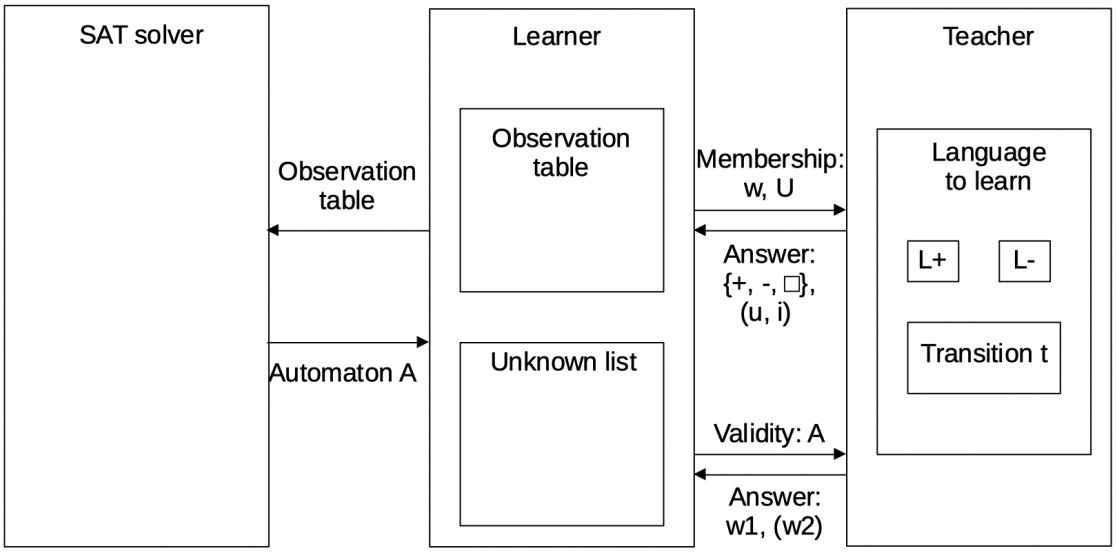

Figure 3.2: A representation of the L_{ICE} algorithm

One of those additions is the concept of an "unknown" membership. While the L_{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 relations between words. The learner builds a set U of words with an unknown membership, and creates series based on these words given a transition between words $t : \Sigma^* \to \Sigma^*$.

These relations can be given to the learner through both membership and validity queries, which replace the L^* learner's equivalence queries. More precisely, the L_{ICE} learner gives a list of words with an unknown membership during membership queries, which can then be used to find a potential relation. During validity queries, however, the learner can receive two words as a counter example: this is an inductive counter example [8].

The relation between words is stored as such: we have a set of words $U \subseteq T^{-1}(\square)$, which is used as a basis for all the relations between words. We can then define N_u for any word $u \in U$ as such: $N_u = \{n \mid n \in \mathbb{N}, \exists w \in S \cup (S \cdot \Sigma), w = t^n(u)\}$.

We end up with the following functions stored in the learner: $\forall u \in U, t_u : \begin{array}{ccc} N_u & \longrightarrow & S \cup (S \cdot \Sigma) \\ n & \longmapsto & t^n(u) \end{array}$

These relations between words are built during membership and validity requests. For example, the learner gives the set U to its teacher during membership request alongside a word $w \in \Sigma^*$, and may receive a word $u \in U$ and a number $i \in \mathbb{Z}$ in return. This means that the words w and u are linked as such: $w = t^i(u)$.

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

3.4 The SAT solver

We can see in figure 3.2 that there is a new component to the L_{ICE} learning process. Indeed, a SAT solver is added to the algorithm's learner and teacher. 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.

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'):

$$
\begin{array}{rll}
 & b_\varepsilon & (\eta_\varepsilon) \\
\forall s \in S, \forall a \in \{a \in \Sigma \mid sa \in S\}, & b_{sa} \Rightarrow b_s & (\eta_{sa}) \\
\forall w \in T^{-1}(\{+,-\}), & x_w \leftrightarrow T(w) & (\sigma_w) \\
\forall u \in U, \forall i,j \in N_u^2, i \leq j, & x_{t^i(u)} \Rightarrow x_{t^j(u)} & (\upsilon_{u,i,j}) \\
\forall s_1, s_2 \in S^2, \forall a \in \Sigma, & e_{s_1,a,s_2} \Leftrightarrow \bigwedge\limits_{e \in E} (x_{s_1 a e} \leftrightarrow x_{s_2 e}) & (\Psi_{s_1,a,s_2}) \\
\forall s \in S, a \in \Sigma, & b_s \Rightarrow \left(b_{sa} \vee \bigvee\limits_{s' \in S} e_{s,a,s'} \right) & (\Phi_{s,a}) \\
\forall s, s' \in S^2, \forall a \in \Sigma, sa \neq s', & b_{sa} \wedge b_{s'} \Rightarrow \neg e_{s,a,s'} & (\Delta_{s,s'})
\end{array}
$$

The new observation table (S', E, T') is defined as follows:

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

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

The η clauses are used to select the new set of accessor words $S' \subseteq S$, which must contain at least the word ε. In order to have to 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 σ and υ clauses are used to ensure that the new function T' follows all the constraints given to the function T. Indeed, if a word $w \in S \cup (S \cdot \Sigma)$ is defined such that $T(w) \in \{+,-\}$, then $T'(w) = T(w)$. That constraint is handled in the clauses σ.

Furthermore, the words with an unknown membership must follow some rules with all of their successors and predecessors using the transition function to ensure that the language is an inductive invariant. These rules are established in the υ clauses.

The clauses Ψ are used to establish that two words are in the same Myhill-Nerode equivalence class, by verifying that the rows of the observation table associated with those words are equal.

The clauses Φ and Δ are used to ensure that the new observation table can represent an automaton, with the Φ clauses used to verify whether the table is closed, and the Δ clauses used to verify that the table is distinct.

The new observation table can then be used to generate a hypothesis automaton as follows:

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

Sometimes, the learner can send a filled observation table with no possible solution. In that case, the SAT solver gives an UNSAT Core, which is a set of failing clauses. This set can then be used to modify the learner's observation table [7].

The SAT solver used in the L_{ICE} framework looks for Φ clauses in the UNSAT Core. When a clause $\Phi_{s,a}, s \in S, a \in \Sigma$ is found, it means that the table cannot be closed. The learner will therefore add the word $s \cdot a$ to the set of accessor words before filling its observation table once more.

3.5 The ICE teacher

The teacher has access to the sets Init and Bad, as well as the Pre and Post functions. It can then use Init, Bad, $\mathrm{Post}^*(\mathrm{Init})$ and $\mathrm{Pre}^*(\mathrm{Bad})$ to answer both membership and equivalence queries.

The sets Init and Bad are given to the teacher and can therefore be used immediately, but $\mathrm{Post}^*(\mathrm{Init})$ and $\mathrm{Pre}^*(\mathrm{Bad})$ both need to be computed. However, the latter two may be infinite sets, which prevents said computation from being possible.

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 creating the set $\mathrm{Pre}^*(w)$ (resp. $\mathrm{Post}^*(w)$), as shown in algorithm 1, and checking if at least one of its members is in Init (resp. in Bad).

Algorithm 1 Build $\mathrm{Post}^*(w)$ and $\mathrm{Pre}^*(w)$

```
function BUILDPOSTS(word)
    current ← word
    posts ← []
    while current ∉ posts do
        posts ← posts + [current]
        current ← Post(current)
    end while
    return posts
end function
function BUILDPRES(word)
    current ← word
    pres ← []
    while current ∉ pres do
        pres ← pres + [current]
        current ← Pre(current)
    end while
    return pres
end function
```

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 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 was in the language to learn, the ICE teacher instead looks at the relations between words to have more specific results. This allows the teacher to specify if the word must be in the language or can't 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 Post function. Therefore, if a word of unknown membership is the predecessor of another, their membership must be linked.

In the context of a membership query, as seen in algorithm 2, 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.

Algorithm 2 Answer a membership query

```
function MembershipAnswer(word, unknowns)
    if word ∈ Init then
        return (True, None, None)
    else if word ∈ Bad then
        return (False, None, None)
    end if
    pres ← BuildPres(word)
    for (i, preWord) ∈ Enumerate(pres) do
        if preWord ∈ Init then
            return (True, None, None)
        else if preWord ∈ unknowns then
            return (None, preWord, i)
        end if
    end for
    posts ← BuildPosts(word)
    for (i, postWord) ∈ Enumerate(posts) do
        if postWord ∈ Bad then
            return (False, None, None)
        else if postWord ∈ unknowns then
            return (None, postWord, −i)
        end if
    end for
    return (None, None, None)
end function
```

3.5.2 Validity queries

In the context of an validity query, the teacher is given an automaton representing a hypothesis for a valid language. The teacher will then verify that the set Init is included in the language, the set 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 3.

Algorithm 3 Answer a validity query

```
function ValidityAnswer(automaton)
    L ← GetLanguage(automaton)
    if Init ⊈ L then
        word ← GetWordIn(Init \ L)
        return (word, None)
    else if Bad ∩ L ≠ ∅ then
        word ← GetWordIn(Bad ∩ L)
        return (word, None)
    else if Post(L) ⊈ L then
        postWord ← GetWordIn(Post(L) \ L)
        pres ← BuildPres(postWord)
        for word ∈ pres do
            if word ∈ Init then
                return (postWord, None)
            end if
        end for
        posts ← BuildPosts(postWord)
        for word ∈ posts do
            if word ∈ Bad then
                return (Pre(postWord), None)
            end if
        end for
        return (Pre(postWord), postWord)
    end if
    return (None, None)
end function
```

While algorithm 3 is the theoretical process used to select a potential counter example, the current implementation of the ICE teacher only checks whether the language is valid for words up to a set length.

Chapter 4

Results

4.1 Example situation

In order to test our L_{ICE} framework, let's take an example of a Regular Model Checking problem.

Let's take a series of tokens, with 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.

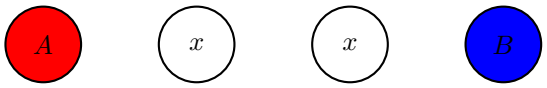

Figure 4.1: Example of a Regular Model Checking problem state

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. If we choose, the transition between states can be cyclical and let tokens go from one end of the series to the other. All those transitions are represented in figure 4.2.

While the initial problem starts with only one red and one blue tokens separated by an arbitrary number of empty tokens, it is possible to find states with more colored tokens during the learning process. We will consider two options to handle such a situation.

- Limited transition: If there are not exactly one token of each color, the state is considered "blocked" and won't go through any change when passed to the transition function.
- Unlimited transition: The number of colored tokens does not matter for the transition function, the state is considered blocked whenever one of the colored tokens is unable to move.

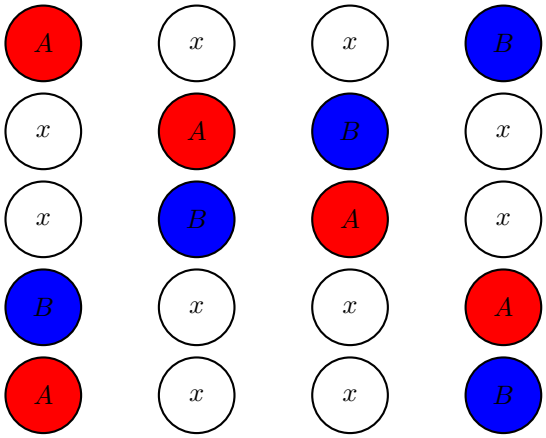

Figure 4.2: Transition between states

4.2 Framework Application

In order to test the L_{ICE} framework, four Regular Model Checking problems were created following the format described in the section above. The results of those tests are compiled in figure 4.3.

Index	Init	Bad	Post	States	Membership	Validity
1	$Ax(xx)^*B$	Bx^*A	Limited, non cyclical	3	122	6
2	$Ax(xx)^*B$	Bx^*A	Unlimited, non cyclical	3	108	4
3	$A(xx)^*B$	x^nAxBx^n	Limited, cyclical	2	27	3
4	$A(xx)^*B$	x^nAxBx^n	Unlimited, cyclical	2	27	3

Figure 4.3: Applications of the L_{ICE} framework

Chapter 5

Future work

While the current implementation of the L_{ICE} framework is working as intended, it could use several optimisations and improvements.

5.1 The SAT solver

The SAT solver used in the learning process of the L_{ICE} algorithm is currently created from scratch for each attempt at creating an automaton. Such an implementation is not optimal, because many clauses could have been conserved in between attempts.

For example, that is the case of the η clauses, which handle the set of accessor words, as well as the $sigma$ clauses which ensure that the words with a set membership are not changed after the completion of the table.

A potential way to improve the SAT solving would be to include the solver directly into the learner, and to use an incremental SAT [6] in that context. Doing so would allow the most used clauses to always be present in the SAT solver, and only some outliers would have to be added before every attempt at creating a validity query.

5.2 Handling counter examples

Another potential optimisation to the L_{ICE} learner would be to change the handling of the counter examples in order to reduce the number of membership queries.

Indeed, the current implementation of the L_{ICE} learner naively adds every suffix of every word it is given as a counter example in the set of distinguisher words. This process artificially creates more membership queries than might be necessary.

While there have been improvements in the handling of counter examples in the L^* algorithm, the addition of unknown memberships make applying the same principles more difficult in our situation.

An option for the handling of single counter examples might be to add more clauses to the SAT solver surrounding the behavior of the given counter example. This would reduce the amount of possible automata found in a given observation table until it is necessary to add a counter example as a distinguisher word.

5.3 Removing the bijection constraint

In our current context, any Regular Model Checking problem must have a bijective transition function for the L_{ICE} framework to word as intended. This is due to the way the relations between words of unknown membership are handled, with the help of a list.

The list structure currently used prevents a given application of the Post function to have more than one result, and likewise for the Pre function.

In order to remove that bijection constraint, the relation between words must be represented by another structure: one which allows a given word to have several successors and predecessors.

A potential option could be an oriented graph. If there is a path from one node to another, it means that the latter is a successor of the former.

Using a graph, it might even be possible to reduce the number of clauses used to define the behavior of unknown words. Indeed, taking into account only the direct neighbors of a node to create the υ clauses does not remove any information, because $\forall w_1, w_2, w_3 \in \Sigma^*$,
$w_1 \in L \Rightarrow w_2 \in L$ and $w_2 \in L \Rightarrow w_3 \in L$ means that $w_1 \in L \Rightarrow w_3 \in L$.

As such, adding only the direct neighbors of a node to the υ clauses will not affect the resolution of the SAT solver, since all the implications will still be indirectly present in the clauses.

Removing the bijection constraint would also allow for more extensive testing, since the Regular Model Checking benchmarks do not meet its criteria. The use of benchmarks [4] would allow us to observe the results of the L_{ICE} framework on varying models, and would be useful to make our model more efficient.

Chapter 6

Conclusion

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

However, the current L_{ICE} framework is still in its premises. For example, the framework currently only works when the transition function between states is bijective. Changing the handling of the SAT solver for the clauses involving words with an unknown membership to work with oriented graphs instead of lists should remove that constraint, and will be the next step in automating the solving of Regular Model Checking problems.

Chapter 7

Bibliography

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

[2] 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 8)

[3] 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)

[4] 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 21)

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

[6] 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. (pages 4 and 20)

[7] Moeller, M., Wiener, T., Solko-Breslin, A., Koch, C., Foster, N., and Silva, A. (2023). Automata Learning with an Incomplete Teacher. (pages 4 and 15)

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

[9] 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)