Conflict-Aware Active Automata Learning

Tiago Ferreira Léo Henry Raquel Fernandes da Silva
University College London
London, UK
{t.ferreira,leo.henry,raquel.silva.20}@ucl.ac.uk

Alexandra Silva
Cornell University
Ithaca, NY, USA
alexandra.silva@cornell.edu

Active automata learning algorithms cannot easily handle *conflict* in the observation data (different outputs observed for the same inputs). This inherent inability to recover after a conflict impairs their effective applicability in scenarios where noise is present or the system under learning is mutating.

We propose the Conflict-Aware Active Automata Learning (C$\mathcal{E}$AL) framework to enable handling conflicting information during the learning process. The core idea is to consider the so-called observation tree as a first-class citizen in the learning process. Though this idea is explored in recent work, we take it to its full effect by enabling its use with any existing learner and minimizing the number of tests performed on the system under learning, specially in the face of conflicts. We evaluate C$\mathcal{E}$AL in a large set of benchmarks, covering over 30 different realistic targets, and over 18,000 different scenarios. The results of the evaluation show that C$\mathcal{E}$AL is a suitable alternative framework for closed-box learning that can better handle noise and mutations.

1 Introduction

Formal methods have a long history of success in the analysis of critical systems through abstract models. These methods are rapidly expanding their range of applications and recent years saw an increase in industrial teams applying them to (large-scale) software [6, 9, 10, 12, 13, 25]. The applicability of such methods is limited by the availability of good models, which require time and expert knowledge to be hand-crafted and updated. To overcome this issue, a research area on automatic inference of models, called *model learning* [32], has gained popularity. Broadly, there are two classes of model learning: *passive learning*, which attempts to infer a formal model from a static log, and *active learning*, where interaction with the system is allowed to refine knowledge during the inference.

In this paper, we focus on active learning, motivated by its successful use in verification tasks, e.g. in analyzing network protocol implementations, as TCP [18], SSH [19], and QUIC [15], or understanding the timing behavior of modern CPUs [34]. Current state-of-the-art active learning algorithms rely on the *Minimally Adequate Teacher* (MAT) framework [4], which formalizes a process with two agents: a *learner* and a *teacher*. The learner tries to infer a formal model of a system, and the teacher is omniscient of the system, being able to answer queries on potential behaviors and the correctness of the learned model. MAT assumes that the interactions between both agents are perfect and deterministic.

Learning In Practice Interactions with the *System Under Learning* (SUL) are often non-deterministic in some way, e.g. the communications can be noisy (i.e. query answers do not only reflect the actual system output, but are instead a consequence of its interaction with the environment), or the SUL itself can change during learning. This can lead to *conflicts*, which we define in the following way:

A *conflict* appears when a query's answer formally contradicts a previous query in a way that cannot be expressed by a model of the target class.

A. Achilleos and D. Della Monica (Eds.): Fourteenth
International Symposium on Games, Automata, Logics,
and Formal Verification (GandALF 2023).
EPTCS 390, 2023, pp. 150–167, doi:10.4204/EPTCS.390.10

Current MAT Learners cannot handle the conflicts that arise during learning. Thus, when used in practice, MAT learner implementations use artifacts to circumvent conflicting observations.

For example, in the case of noise, each interaction has a chance of diverging from its usual behavior. To handle this, MAT learners repeat each query n times and majority-vote the result. They aim to guess an n sufficiently large to prevent *any* noisy observation from reaching the learner, but small enough to let the computation finish before timeout. As a consequence, noise threatens both *efficiency* and *correctness* of learning. We provide a framework alleviating this issue without tailoring it to specific MAT learners.

Irrespective of the nature of the conflicts detected, dealing with them requires the ability to *backtrack* certain decisions that were made based on what is now considered incorrect information. This pinpoints the issue with current MAT learners: there is no notion of information storage other than the internal data structure that the learners use to build the model, which is not easily updatable in the face of conflict. This structure in fact needs to be fully rebuilt if a conflict is found, generating many superfluous (and expensive!) queries to the SUL. Separating the learning process from the information gathered through the queries allows us to *retain* all the previous non-conflicting information. This alleviates the main cost of conflict handling: the unnecessary repetition of tests on the system. The Learner then only needs to rebuild its data structure based on the information already available.

Contribution Based on the ideas above, this paper proposes the *Conflict Aware Active Automata Learning* (C$\mathcal{E}$AL, pronounced *seal*) framework. Any existing MAT learner can be used in C$\mathcal{E}$AL. When a conflict arises, we provide a method for updating the learner's internal state — without making assumptions on its data-structure — so that it remains conflict-free while removing only inconsistent information.

> In a nutshell, this paper aims to provide classic MAT learners with a way to recover from conflicts caused by either noise or potential mutations of the system.

At the heart of C$\mathcal{E}$AL is the use of an *observation tree*, a data structure (external to the learner) used to store information gathered from the SUL. It can be efficiently updated and used by the learner to construct its own internal data structure. When a conflict appears, we update the observation tree to reflect our knowledge, while the learner's data structure is *pruned* to a conflict-free point and then expanded from the observation tree. Crucially, the learner uses *the observations already stored in the tree* without requiring tests on the SUL for already observed behaviors. C$\mathcal{E}$AL's main features are:

- The SUL is a first-class citizen, instead of being abstracted. C$\mathcal{E}$AL notably does not rely on *equivalence queries*, replacing them with either a check of the stored knowledge (when sufficient) or an *equivalence test*, using an m-complete testing algorithm (e.g. the Wp-method [20] or Hybrid-ADS [26,29]).
- The information obtained through tests on the SUL is stored in an observation tree managed by a new *Reviser* agent that is responsible for handling the conflicts and answering the learner's queries like a teacher. Providing a teacher interface is an important aspect as it enables the use of any MAT-based algorithm seamlessly, only requiring the ability to restart a classic MAT learner.
- The Reviser alone interacts with the SUL by means of tests meant to expand its observation tree.

Crucially, C$\mathcal{E}$AL is less abstract than MAT, representing directly the objects and challenges of *practical* active learning, while still allowing the design of Learners to enjoy the simplifying abstraction of MAT.

After some preliminaries in Section 2 we formalize and prove the above claims in Section 3. We evaluate C$\mathcal{E}$AL in Section 4 using a broad range of experiments [27]. We compare several state-of-the-art algorithms (namely $\mathtt{L}^\star$ [4], `KV` [24], `TTT` [22] and $\mathtt{L}^\#$ [33]) for targets of different sizes and different levels of noise, while varying the controllable parameters for both MAT and C$\mathcal{E}$AL. The experimental

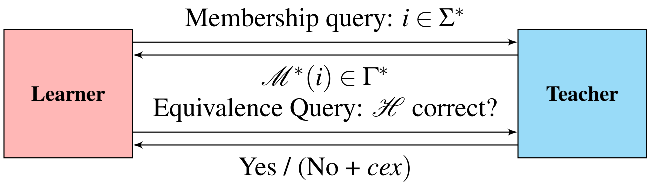

Membership query: $i \in \Sigma^*$
Learner
$\mathscr{M}^*(i) \in \Gamma^*$
Equivalence Query: $\mathscr{H}$ correct?
Teacher
Yes / (No + *cex*)

Figure 1: The Minimally Adequate Teacher framework.

results show that in the case of noise, CƐAL allows us to drastically reduce the number of repeats required to learn correct models by handling some conflicts in the information it gathers from the system. This allows CƐAL to achieve a success rate of 95.5% compared to MAT's 79.5% in our experiments.

A long version of this paper, complete with the appendices presenting the proofs, some more formalization, extensive experimental results and some more discussion can be found in [16]. References are given when it can be useful.

2 Preliminaries

In this section, we recall Mealy machines and MAT. Fix an alphabet A (a finite set of symbols). The set of finite words is denoted A^*, the empty word ε, and the set of non-empty words by A^+. The length of a word $w \in A^*$ is denoted $|w|$, its sets of prefixes by $\mathsf{prefixes}(w)$, its k-th element by $w[k]$ and the subword from the i-th to the j-th element by $w[i,j]$. The concatenation of word w with symbol a is denoted by wa.

Mealy Machines For the rest of the paper, we fix an input and output alphabet pair (Σ,Γ). A *Mealy machine* over alphabets (Σ,Γ) is a tuple $\mathscr{M} = (Q, q_0, \delta, \lambda)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \to Q$ is a transition function and $\lambda : Q \times \Sigma \to \Gamma$ an output function. Mealy machines assign *output words* ($o \in \Gamma^*$) to *input words* ($i \in \Sigma^*$) — one reads input letters using δ and collects all output letters given by λ. This is achieved using inductive extensions of δ and λ:

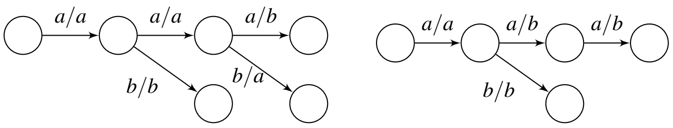
$$\delta^* : Q \times \Sigma^* \to Q \qquad \delta^*(q,\varepsilon) = q \qquad \delta^*(q,ia) = \delta(\delta^*(q,i),a)$$
$$\lambda^+ : Q \times \Sigma^+ \to \Gamma \qquad \lambda^+(q,ia) = \lambda(\delta^*(q,i),a)$$

We now build the semantics function $\mathscr{M}^* : \Sigma^* \to \Gamma^*$ given by

$$\mathscr{M}^*(a_1 \cdots a_j) = b_1 \cdots b_j \quad \text{where } b_k = \lambda^+(q_0, a_1 \cdots a_k), \text{ for all } k = 1, \ldots, j.$$

Note the preservation of length of input words in output. When the functions δ and λ are partial we call the Mealy machine $\mathscr{M}$ partial. A partial tree-shaped Mealy machine is called an *observation tree*.

Example 1. *On the left below is the tree representing the tests* $\{(aaa,aab),(aab,aaa),(ab,ab)\}$ *and on the right the tree representing* $\{(aaa,abb),(ab,ab)\}$.



Active Model Learning Active learning is a process in which a *learner* can interact with an omniscient *teacher* to build a model of an unknown system. Formally, this type of learning uses the *Minimally Adequate Teacher* (MAT) framework [4] (see Fig. 1). The teacher is supposed to have enough knowledge about the target machine $\mathscr{M}$ to be able to answer two types of queries:

Membership The learner sends an input word i to the teacher, who answers with the output word $\mathscr{M}^*(i)$.

Equivalence The learner proposes a hypothesis model $\mathscr{H}$. The teacher either confirms the model as correct or provides a counterexample $cex \in \Sigma^*$ such that $\mathscr{H}^*(cex) \neq \mathscr{M}^*(cex)$.

The MAT framework is an interesting abstraction to design algorithms and conduct proofs, and has been the basis for active model learning since its introduction (see e.g. `L*` [4], `KV` [24], `TTT` [22] or `L#` [33]). The teacher abstracts the system under learning (SUL), which complicates discussions on the practical interfaces between the learner and the SUL during applications. MAT does not separate the learner's core features (i.e. choosing the queries to be made and building hypotheses) from the storage of observations. This has led the community to resort to *caches*, often implemented through observation trees, to access observations directly. Being mostly tricks to avoid repeating queries, caches are rarely discussed in the literature (although used during experiments), which had so far delayed a discussion on the practical implications of a proper handling of observations. This paper addresses this.

Noise on communications The term *noise* is usually used to described a wide range (if not any form) of perturbations that can happen between the designed agent (in our case the Learner) and the SUL. In the case of this study we are primarily interested in the classification between *input* and *output* noise.

Output noise We call output noise a perturbation that only affects what our agent sees from the world, i.e. the outputs of the SUL. Formally, this kind of noise can be represented as a non-deterministic function of $\mathscr{M}^*(i)$ returning a different output word of same size.

Input noise This kind of noise instead affects the query i inputted into the SUL, so that a different input word i' of same size is processed instead.

Noise can have different levels of *structure*, being generated by different kinds of models or probability distributions. As this paper strives for a generic approach, no assumption is made on the structure of noise. Furthermore, experiments will use generic noise that has a fixed probability *for each symbol* of the word, taken in sequence, to replace it with a random one according to a uniform distribution. One notable restriction of our approach is that it does not target adversarial modifications — such as an attacker trying to change the Learner's hypotheses.

Remark 1. *We do not further formalize noise, as it stems for very practical considerations that may require a wide array of different formalizations. The method we propose is* generic *and aims to demonstrate that paying attention to noise and* conflicts *allows significant efficiency gains without any specialization towards a specific model of noise.*

3 Conflict Aware Active Automata Learning

We now introduce our alternative to MAT in practice — the *Conflict-Aware Active Automata Learning* (C$\mathcal{E}$AL) framework. C$\mathcal{E}$AL's main features are as follows:

- The SUL is a first class citizen, allowing for clearer practical discussions and modularity.
- The information obtained through tests on the SUL is stored in a new *Reviser* agent that handles the conflicts and answers the learner's queries like a teacher.
- The Reviser alone interacts with the SUL by means of tests meant to expand its observation tree. The learner's queries are answered from the observation tree.

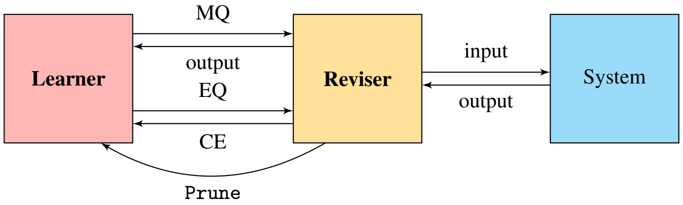


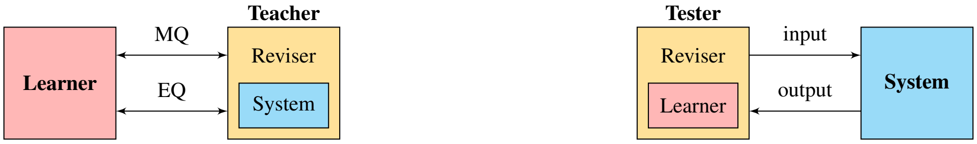
Figure 2: Simplified view of the C$\mathcal{E}$AL framework. See Fig. 4 and 5 for more detail.

3.1 Framework Overview

C$\mathcal{E}$AL (Fig. 2) is centered around three agents — the Learner, the System (SUL), and the Reviser — and the interfaces between them. The **Learner** plays the same general role as in MAT. Crucially, any MAT learner can be used in C$\mathcal{E}$AL (e.g. $\mathtt{L}^\star$, $\mathtt{KV}$, $\mathtt{TTT}$, $\mathtt{L}^\#$). The Learner *does not have* to store the information obtained from tests on the system. It focuses on the questions "What is the next query to make?" and "How is the hypothesis built?". The **System** is the System Under Learning, together with its environment (e.g. noise). The **Reviser** handles knowledge and conflicts. It answers the question "What do we know about the system?". It is set between the Learner and the System with interfaces to both of them.

C$\mathcal{E}$AL is designed to improve the practical learning of reactive systems like Mealy machines. As such, it makes use of features that are core to such models, like causality and closure under inputs and outputs. However, the main ideas behind C$\mathcal{E}$AL's philosophy and separation of concerns can be adapted to learn other types of automata, such as acceptors like DFAs.

Remark 2. *The Reviser acts as a MAT teacher w.r.t. the Learner, answering membership and equivalence queries, with the added ability to* `Prune` *the learner to place it in a state coherent with the Reviser's information. On the System's view, the Reviser acts as a tester, providing input sequences (tests) and recording the system output. The outside views of the learner and system in* C$\mathcal{E}$AL *are illustrated below.*



(a) Learner's view

(b) System's view

Figure 3: Reviser's interfaces

The **interfaces** on the Learner side are similar to MAT: the Learner can perform membership queries (MQ) and equivalence queries (EQ) on the Reviser, with the latter potentially resulting in a counterexample (CE). Note that the queries are sent to the Reviser and not directly to the SUL: a crucial design choice. This allows us to control the information that the Learner obtains, and reuse the information in the Reviser with no new tests. Formally, C$\mathcal{E}$AL provides the following functions as module interfaces:

- $\mathtt{MQ} : \Sigma^* \to (\Gamma^* \cup \{\mathtt{Prune}\})$ the membership query of the learner that the Reviser has to implement. It varies from the MAT function as the Reviser may return a command to prune the learner's state instead of an output.
- $\mathtt{EQ} : \mathsf{Mealy} \to ((\Sigma^* \times \Gamma^*) \cup \{\mathtt{Prune}\})$ the equivalence query of the learner that the Reviser has to implement. It may return "`Prune` " instead of "Yes".
- $\mathtt{System} : \Sigma^* \to (\Sigma^* \times \Gamma^*)$ is a call to the system for a specific test. The system returns the corresponding behavior (input and output), with the effect of noise applied.

In the interface mentioned above, an `EQ` can never return "Yes" as in MAT. This work is left to the Reviser, that will halt the learning process according to the termination criterion chosen (see Section 3.2). The Prune signal does not require us to modify the code of a MAT Learner, as it can be implemented by restarting the Learner without requiring further access to the Learner's internals. The Reviser's caching of observations ensures that this operation does not add to the query complexity of the process.

Remark 3. *The main cost of learning comes from* unit interactions with the system — *each individual symbol that is inputted into or outputted by the system — as these tests are generally costly to perform and that cost cannot be compensated.*

3.2 The Reviser

The Reviser agent is the core of the C$\mathcal{E}$AL framework. It concretizes its main idea: taking the storage and handling of observations *out* of the Learner's prerogatives. Its task is to update the observation tree $\mathscr{T}$ on which a Learner is trained. We will assume the following interface is made available to the Reviser:

Definition 1 (Operations on observation trees)**.** *Given an observation tree $\mathscr{T}$, we define the following functions to access and modify $\mathscr{T}$:*

- $\textsc{Lookup}_{\mathscr{T}} : \Sigma^* \to (\Gamma^* \cup \{\textsc{Null}\})$ *receives an input word i and returns output o if (i,o) is present in the observation tree $\mathscr{T}$. Otherwise,* $\textsc{Null}$ *is returned.*
- $\textsc{Update}_{\mathscr{T}} : (\Sigma^* \times \Gamma^*) \to \mathbb{2}$ *updates the observation tree $\mathscr{T}$ to take into account a new query pair, revoking conflicting information if necessary. Returns $\top$ if the new information conflicts with $\mathscr{T}$.*

Note that the function $\textsc{Update}$ is the only one that alters the tree and handles conflicts. Implementations of these functions are given in [16, Appendix A]. Using these functions, we can now define the *language* of an observation tree as the set of observations that it can transmit to a Learner, and provide a formal definition of a conflict as a non-additive change to the language of $\mathscr{T}$.

Definition 2. *Given an observation tree $\mathscr{T}$, we call* language *of $\mathscr{T}$ the following set*

$$\mathscr{L}_{\mathscr{T}} = \{(i, \textsc{Lookup}_{\mathscr{T}}(i)) \in \Sigma^* \times \Gamma^* \mid \textsc{Lookup}_{\mathscr{T}}(i) \in \Gamma^*\} \ .$$

Definition 3. *An observation (i,o)* conflicts *with an observation tree $\mathscr{T}$ when the tree $\mathscr{U}$ obtained by calling $\textsc{Update}_{\mathscr{T}}(i,o)$ satisfies $\exists (i',o') \in \mathscr{L}_{\mathscr{T}},\ o' \neq \textsc{Lookup}_{\mathscr{U}}(i')$. Two observations (i,o) and (i',o')* conflict*, written $(i,o) \lightning (i',o')$, when there is an input word $i'' \in \mathsf{prefixes}(i) \cap \mathsf{prefixes}(i')$ such that $o[|i''|] \neq o'[|i''|]$.*

Note that a conflict appears not between the System and the Reviser, but signifies that the Reviser wants to update its answer to some information previously given to the Learner.

Definition 4 (Reviser)**.** *The Reviser contains an observation tree $\mathscr{T}$ and implements four operations:*

① $\textsc{Apply}_{\mathscr{T}} : (\Sigma^* \times \Gamma^*) \to (\Gamma^* \cup \{\texttt{Prune}\})$ *updates $\mathscr{T}$ with the observation gained from a system test. It then either returns the query output or prunes the learner if a conflict is detected.*

② $\textsc{Read}_{\mathscr{T}} : \Sigma^* \to (\Gamma^* \cup \{\texttt{Prune}\})$ *looks in $\mathscr{T}$ for a query answer and either returns it or tests the system if necessary. Note that if a test is performed, then $\mathscr{T}$ is updated accordingly.*

Algorithm 1: $\textsc{Apply}_{\mathscr{T}}(i,o)$

```
Data: $(i,o)$ trace from the SUL.
if $\textsc{Update}_{\mathscr{T}}(i,o)$ then
    return Prune;
return $o$;
```

Algorithm 2: $\textsc{Read}_{\mathscr{T}}(i)$

```
Data: The queried string $i$.
$o \leftarrow \textsc{Lookup}_{\mathscr{T}}(i)$;
if $o \neq \textsc{Null}$ then return $o$ ;
return $\textsc{Apply}_{\mathscr{T}}(\texttt{System}(i))$;
```

③ $\textsc{Check}_{\mathscr{T}} : \mathsf{Mealy} \to ((\Sigma^* \times \Gamma^*) \cup \{\textsc{Null}\})$ *performs a consistency check of a given Mealy machine hypothesis against the observation tree* $\mathscr{T}$. *Returns a counterexample if found or* NULL *if no divergences are found.*

④ $\textsc{Test}_{\mathscr{T}} : \mathsf{Mealy} \to ((\Sigma^* \times \Gamma^*) \cup \{\texttt{Prune}\})$ *is the function used to look for counterexamples in the System. It takes a hypothesis proposed by the learner and coherent with the observation tree, and tests the SUL until a counterexample or a conflict is found. The tests are taken from* `sampleWord` *which is instantiated by an off-the-shelf test suite generating algorithm (e.g. the Wp-method [20] or Hybrid-ADS [26, 29]) in practice.*

Algorithm 3: $\textsc{Check}_{\mathscr{T}}(\mathscr{H})$

```
Data: Hypothesis $\mathscr{H}$
for $(i,o) \in \mathscr{T}$ do
    if $\mathscr{H}^*(i) \neq o$ then
        return $(i,o)$;
return NULL;
```

Algorithm 4: $\textsc{Test}_{\mathscr{T}}(\mathscr{H})$

```
Data: A hypothesis $\mathscr{H}$ coherent with $\mathscr{T}$.
while $\top$ do
    $w \leftarrow$ sampleWord();
    $(i,o) \leftarrow$ System($w$);
    if $\textsc{Apply}_{\mathscr{T}}(i,o) =$ Prune then return
     Prune ;
    if $\mathscr{H}(i) \neq o$ then return (i,o) ;
```

Crucially, the above functions rely on the observation tree's interface to handle the conflict as they arise, forwarding the `Prune` command to the Learner when needed.

Update Strategies At the core of dealing with conflicts is the idea of identifying information that will be sacrificed for the sake of cohesion. The way this is achieved depends largely on the type of conflict, and the *meaning* of observing such a conflict. We propose two ways to resolving conflicts in C$\mathcal{E}$AL:

① `Most Recent`: When a conflict is identified, the most recently observed (freshest) query information is committed to the observation tree, and the previous one suppressed, if needed. This approach makes sense, for example if the target system has mutated and we are only interested in capturing the most up-to-date behavior, or as a base default strategy. We define $\mathsf{prefixes}(i,o) = \{(i[1,n], o[1,n]) \mid 0 \leq n \leq |i|\}$.

Proposition 1. *In the case of the* `Most Recent` *update strategy, given a stream of tests* $((i_k, o_k)_{k \in \mathbb{N}})$, *at any step* $K \in \mathbb{N}$*:* $\mathscr{L}_T = \{\mathsf{prefixes}(i_k, o_k) \mid 0 \leq k \leq K \land \nexists k < l \leq K,\ s.t.\ (i_k, o_k) \lightning (i_l, o_l)\}$.

Example 2. *In Example 1, the right-hand tree is the result of observing* (aaa, abb) *starting from the left-hand tree. Notice that the sets prefixes of the sets of observations in Example 1 verify Proposition 1.*

② `Most Frequent`: When two possible output sequences conflict for a given input sequence, the most frequently observed one is returned to the Learner. This information can be obtained passively by keeping track of naturally occurring repetitions of queries, or actively by specifying a sample size on which the frequency is estimated. This approach makes sense for example for conflicts that are due to unwanted statistical noise in the observations.

We define $\mathsf{Count}(i,o) = |\{k \mid (i,o) \in \mathscr{P}(\mathsf{prefixes})(\{(i_k, o_k)_{k \in K}\})\}|$ as the number of observations of which (i,o) is a prefix in an observation stream $(i_k, o_k)_{k \in \mathbb{N}}$ considered at step K.

Proposition 2. *In the case of the* `Most Frequent` *update strategy, given a stream of tests* $((i_k, o_k)_{k \in \mathbb{N}})$, *at any step* $K \in \mathbb{N}$*:*

$$mf((i_k, o_k), (i_l, o_l)) \triangleq \mathsf{Count}(i_k, o_k) < \mathsf{Count}(i_l, o_l)\ \lor\ (\mathsf{Count}(i_k, o_k) = \mathsf{Count}(i_l, o_l)\ \land\ k < l)$$

$$\mathscr{L}_{\mathscr{T}} = \{\mathsf{prefixes}(i_k, o_k) \mid k \leq K \land\ \nexists k < l \leq K,\ s.t.\ (i_k, o_k) \lightning (i_l, o_l)\ \land mf((i_k, o_k), (i_l, o_l))\}$$

We present implementations of UPDATE and LOOKUP fitting these two strategies in [16, Appendix A], and proofs of the above properties in [16, Appendix B].

Remark 4. *An observation (i,o) conflicting with an observation tree $\mathscr{T}$ implies that $(i,o) \lightning (i',o')$ for some $(i',o') \in \mathscr{L}_{\mathscr{T}}$. The other implication is not always true, e.g. for the `Most Frequent` update strategy.*

Termination The termination criteria of C$\mathcal{E}$AL are the same as those used in MAT in practice: in our experiment, we terminate when our currently selected hypothesis has survived for a fixed number of tests that is deemed sufficient, or if a predefined limit number of queries is reached.

Hypothesis Selection Active automata learning involves the production of a sequence of hypotheses that are refined over time, with the goal of converging towards a correct one. As such, a key characteristic of different approaches to automata learning is how a final model is to be selected, out of the many hypotheses. In the case of MAT this is simple: Learning produces a sequence of ever more accurate models, until termination occurs with a positive equivalence query. It is then logical to pick the most recently produced hypothesis as the final model. However, when dealing with conflicts and different update strategies, this is no longer necessarily the case for C$\mathcal{E}$AL. In particular, when it comes to electing a model out of a sequence of hypothesis, C$\mathcal{E}$AL has two options:

- `Most Recent`: This hypothesis selection strategy is the one known classically: the most recently produced hypothesis is the one to be elected as final. This strategy is sensible in the case of learning with no noise, or in the case of learning targets that evolve over time.
- `Most Frequent`: In this selection strategy, the sequence of hypotheses is analyzed to elect a final model. We count the frequency of each unique model (up to language equivalence) over the sequence, and elect the most frequently occurring one. This strategy makes sense when dealing with noise, as we may be producing (rarely) hypotheses that capture noisy behavior that is fixed over time. As such, we want to select not the latest model produced, but the one that is the most stable. This strategy can be implemented efficiently in practice (using hash fingerprints and counters, for example) and on-the-fly during learning, allowing us to not have to store the whole sequence of hypotheses as it is produced.

3.3 Interface Implementation

We now explain how to build the interface described in Sec. 3.1 using the Reviser. This mostly amounts to implementing membership and equivalence queries, as the testing interface is simply composed of calls to `System`. **Membership** queries can be defined, for $i \in \Sigma^*$, as $\mathtt{MQ}(i) = \textsc{Read}_{\mathscr{T}}(i)$. When $\mathscr{T}$ does not have the answer to this particular query, $\textsc{Read}_{\mathscr{T}}$ sends it through to the SUL (with the call to `System`) and the result is applied in $\mathscr{T}$. This process is illustrated in Fig. 4.

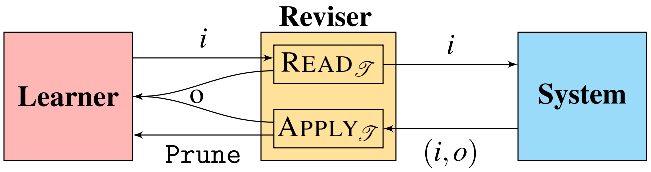


Figure 4: Implementation of Membership Queries (`MQ`) in the Reviser.

Equivalence queries are handled in two steps. First, the hypothesis given by the learner is checked against the observation tree using $\textsc{Check}_{\mathscr{T}}$. If a counterexample is found, it is returned. Otherwise, the

Reviser tests the System to discover new information and update the tree. If a counterexample is found, either it is returned to the learner or, if a conflict arose, the learner is pruned. (Fig. 5).

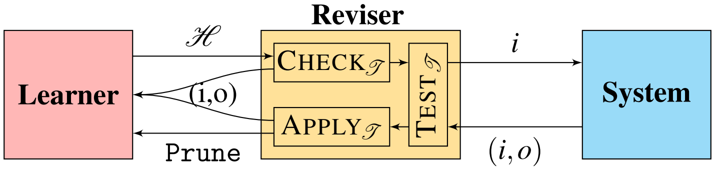


Figure 5: Implementation of Equivalence Queries (EQ) in the Reviser.

Remark 5 (Modularity)**.** *We present* CƐAL *in the case of* black-box *learning where we do not have access to any information from the system but can interact with it. However, the framework is fully modular, as can be seen from the high-level functions presented in Section 3.2: one can interface any model-checking (or other) method before the calls to system in* TEST$_{\mathscr{T}}$ *and* READ$_{\mathscr{T}}$ *when related models (specifications, parts of the System...) are available, allowing* CƐAL *to perform gray-box or even white-box learning.* CƐAL *focuses on the* storage *of information, without restrictions on its acquisition.*

Correctness Proposition 1 and Proposition 2 characterize the language of the Reviser, and the following results describe its interactions with the Learner and the System (as proved in [16, Appendix B]).

Lemma 1. *During an execution of* CƐAL*, all tests on the System are integrated in $\mathscr{T}$ through* UPDATE$_{\mathscr{T}}$*, and the Learner queries are answered according to $\mathscr{L}_{\mathscr{T}}$.*

Proposition 3. `Prune` *is sent to the Learner exactly when a new observation conflicts with $\mathscr{T}$.*

3.4 Optimizations

CƐAL allows us to implement two main optimizations, both in the framework and its algorithms.

Query Caching One of the direct benefits of the presence of the Reviser is that the learner does not have to cache membership queries to avoid repeating them, as it obtains knowledge through the Reviser's data structure. It especially offers a good basis to discuss algorithms that are based on the observation tree themselves [30, 33], for which the Learner's data structure is very limited.

Specialized Pruning In order to fully support the simplistic interface of a classic MAT learner, we restart it — at no extra query cost (see Section 3.1) — when the `Prune` signal is sent. For specific Learner data structures, the time-complexity of this operation can be enhanced by suppressing only the part of the data structure that is impacted by the conflict (instead of restarting it completely). This optimization, however, will be specific to each learning algorithm. In the same way as the Learner can read the Reviser' observation tree, a CƐAL-specific Learner could compute its internal data-structure directly from it after a pruning without requiring restarts.

4 Evaluation

We introduced CƐAL to extend the power of classic MAT learners into environments that may cause conflicts, for instance caused by *noise*. In practice, each symbol inputted in or outputted by the SUL can be noisy, making longer queries more likely to have noisy results overall. Recall that poorly guessing the number n of query repetitions can lead to a learning failure (if noise is integrated in the system), or a timeout (if too many repetitions are made). Hence noise does not only affect the *efficiency*, but also the *success rate* of the learning, i.e., the proportion of runs that end with a correct hypothesis.

It is then important to measure how well C$\mathcal{E}$AL can avert these negative effects. In particular, we are interested in testing if C$\mathcal{E}$AL's approach is sufficient to allow for an improvement of learning environments plagued by noise[1]. We evaluate this through the following research question:

> Across different realistic model learning targets, types of noise, and noise levels, does C$\mathcal{E}$AL provide a better learning environment in terms of both success rates and number of tests issued when compared to the state-of-the-art MAT-based approaches?

4.1 Experiments

We first present the experimental setup and the outlines of the experimental results. We focus on the difference between *uncontrollable parameters*, i.e., that are part of the target and its environment and can't be altered by the learning setup, and the *controllable* ones, that are chosen when designing a learning session. *Uncontrollable parameters* are related to the SUL and its environment.

Realistic Targets We run our experiments over a range of 36 Mealy machines representing real world systems from previous successful model learning applications [27]. These range in size between 4 and 66 states, with alphabet sizes between 7 and 22 input symbols.

Different Types of Noise We run the targets on different types of realistic simulated noise, namely input and output noise as described above.

Different Levels of Noise We run the above mentioned noise types over 3 different levels: 0.01, 0.05, and 0.1. These indicate the probability that each symbol has of being noisy.

Given the above constraints, we aim to reach the best performing learning session by manipulating the following *controllable parameters*:

Framework We run each experiment under both MAT and C$\mathcal{E}$AL.

Algorithm We run each experiment under `L#`, `TTT`, `KV`, and `L*`. We use the implementations of these algorithms provided in LearnLib [23]. Notably, `L*` is implemented with Rivest & Schapire's improvements [28] and we re-implemented `L#` completely to incorporate it into the LearnLib library.

Number of Repeats We compare different numbers of repeats used in majority voting test results to remove noise in MAT, or in sampling frequencies for the update strategy in C$\mathcal{E}$AL. We use one of 3 different levels of repeats, in pairs of (*min_repeats*, *max_repeats*): $(5,10),(10,20),(20,30)$[2].

Each experiment uses the following settings to enhance its learning, independent of the above mentioned variables. Firstly, caching of previously observed queries is done wherever possible in MAT, and the `Most Recent` update and `Most Frequent` hypothesis selection strategies are used in C$\mathcal{E}$AL, for simplicity. Secondly, the Hybrid-ADS [26] equivalence testing algorithm is used for all runs as we found it to be the best performing for our experiments. Thirdly, each independent experiment is performed with 100 runs, and its results averaged for consistency. And finally, each run is allowed to use up to 10 million queries before an unsuccessful timeout is declared.

[1] We compare success rates and number of tests issued instead of running times as to make hardware-agnostic benchmarks that capture the main factors in both efficiency and correctness.

[2] Each test is repeated *min_repeats* times and then if at least 80% of the queries agree the result is returned. Else, the query is repeated until it is the case or *max_repeats* is reached at which point the majority answer is returned.

Due to the vast number of variables considered, we are unable to fully describe the result of the over 18,000 distinct experiments, and close to 2 million runs that we have performed. However this is not required to rigorously answer our research question. What we have to consider is, for each combination of our independent variables (target, noise type, and noise level), which framework allows for the most efficient learning configuration.

The graphs below (Figs. 6 - 11) summarize, for each target and for all levels and types of noise, the success rates and number of symbols tested of the best controllable parameter profile for both MAT and CℰAL. We have also included all the data used, as well as conclusions in [16, Appendix C].

4.2 Analysis

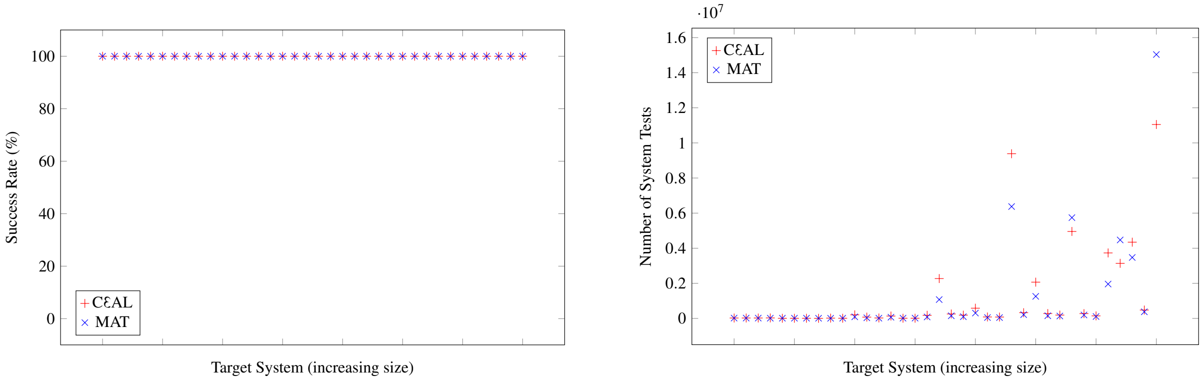
We now analyze the results of these experiments to draw some high-level conclusions about how MAT and CℰAL compare and answer our research question.

Success Rates First and foremost, we discuss the impact of different parameters on the success rates of the experiments. We can see from the graphs (Figs. 6a - 11a) that, as expected, while the exact type of noise does not have a significant impact on success rates and test counts, the *level of noise* does. In particular, at a very low level of 0.01% (Figs. 6a, 9a) both frameworks are capable of maintaining perfect success rates. However, once the level increases to 0.05% (Figs. 7a, 10a), MAT's success rate starts to fluctuate, more so in bigger targets. CℰAL too seems to be slightly affected by an increase in noise, but overall maintains a success rate close to 100%. Once the noise is increased to its highest level, 0.1% (Figs. 8a, 11a), we can see that MAT's success rates reduce significantly, while CℰAL's tend to stay high for a great number of targets, until they inevitably decrease when faced with massive targets at this level of noise. CℰAL **manages to stay consistently reliable in the face of these large alphabets**.

Efficiency Let us now turn our attention to the system test count graphs (Figs. 6b - 11b). Overall we see an expected picture: Larger systems require more tests to be learned. A particular caveat to notice however, is that while MAT appears to have quite efficient runs on large noisy targets, their respective success rates are considerably lower. Although efficiency of learning is certainly important, it is of low use if at the end the reported hypothesis is not correct. This result is expected: If a learning run fails due to, for example, high noise not being fully filtered out, the MAT learner will collapse before it finishes running. This leaves a final test count that is quite low, but also gives us an incorrect hypothesis.

Overall Results Perhaps most importantly, CℰAL **provides the most efficient *correct* configuration in 70% of the experiments**, having a better success rate than MAT or the same with a lower average number of tests used. We provide this result for each individual experiment in [16, Appendix C]. In particular, in every experiment CℰAL performs with a success rate that is at least as high as MAT's, often outperforming it. In addition to this, experiments ran with CℰAL had an overall success rate of 95.5% compared to MAT's 79.5% success rate. This alone has allowed CℰAL to perform successful runs that no configuration of MAT was able to perform, namely learning moderate to large targets at 0.1% noise.

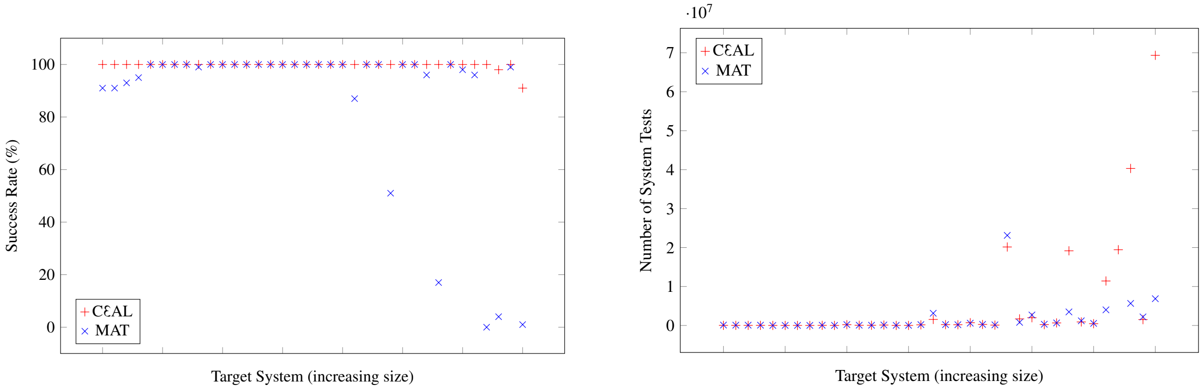
We found that a lot of the improvements provided by CℰAL are commonly a consequence of it being more successful when running at a *lower number of repeats* when compared to the ones required by MAT. This solidifies our initial hypothesis of there being a benefit in reducing the number of repeats used when learning noisy targets. The above provides enough supporting evidence to answer our research question positively: CℰAL **provides a better learning environment in terms of both success rates and number of tests issued when compared to the state-of-the-art MAT-based approaches**.



(a) Success rate (higher is better)

(b) Average test use (lower is better)

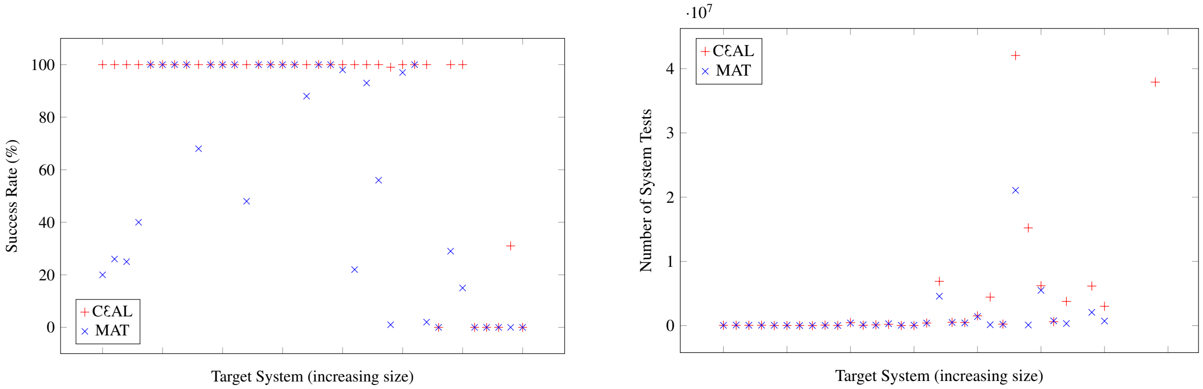
Figure 6: **INPUT** noise at **0.01%**.



(a) Success rate (higher is better)

(b) Average test use (lower is better)

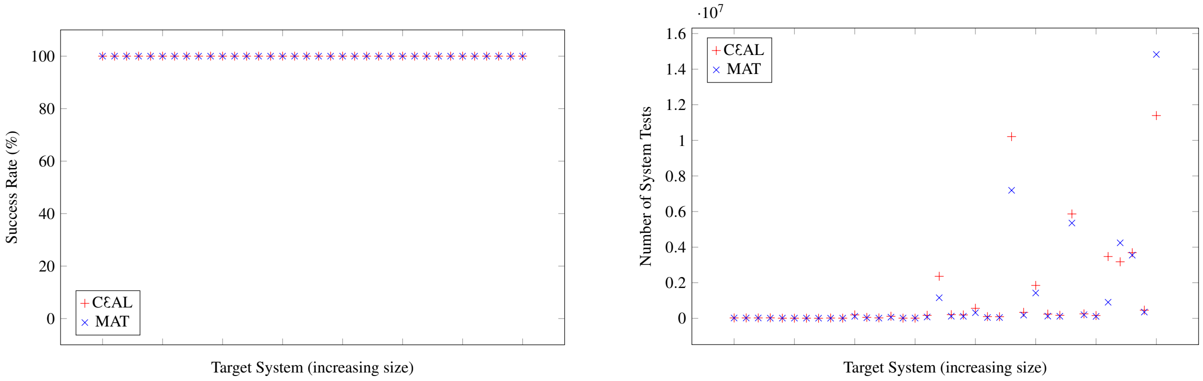
Figure 7: **INPUT** noise at **0.05%**.



(a) Success rate (higher is better)

(b) Average test use (lower is better)

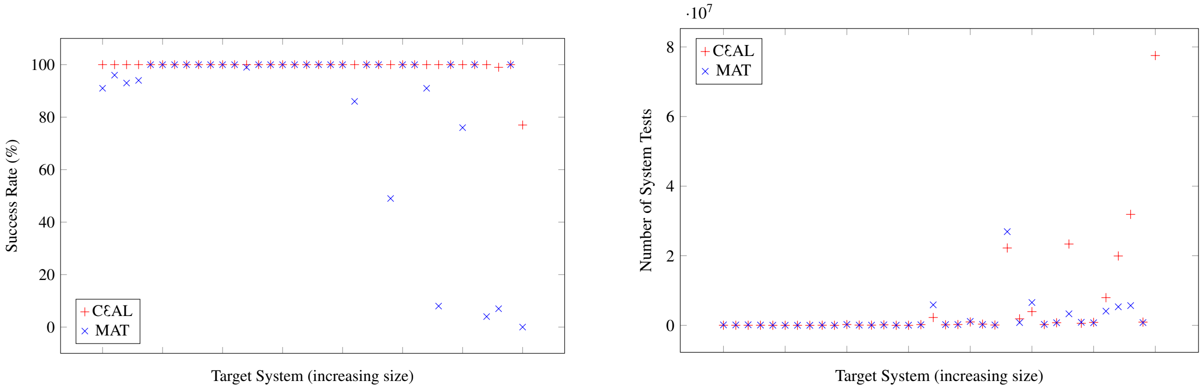
Figure 8: **INPUT** noise at **0.1%**.



(a) Success rate (higher is better)

(b) Average test use (lower is better)

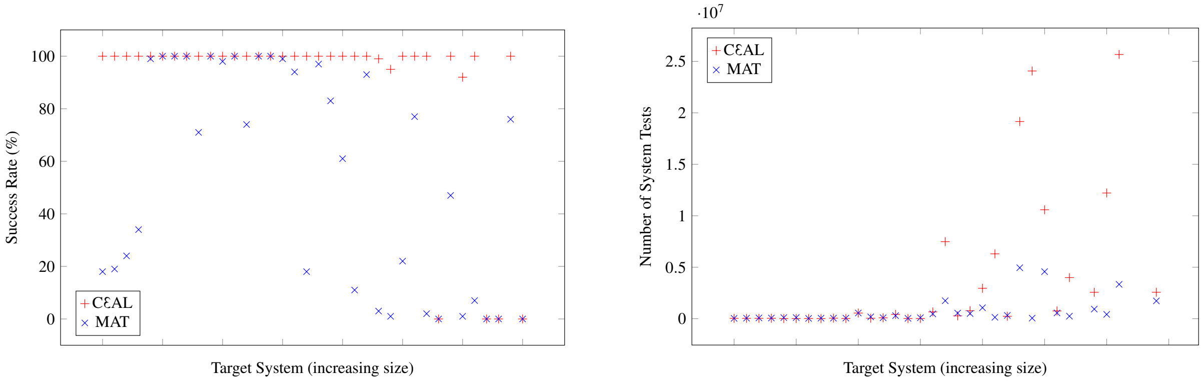
Figure 9: **OUTPUT** noise at **0.01%**.



(a) Success rate (higher is better)

(b) Average test use (lower is better)

Figure 10: **OUTPUT** noise at **0.05%**.



(a) Success rate (higher is better)

(b) Average test use (lower is better)

Figure 11: **OUTPUT** noise at **0.1%**.

Other Findings We report on other interesting findings in [16, Appendix D]. Two results, however, are of particular significance: As already accepted by the community [5], we can confirm that indeed most of the tests spent in learning are used to realize equivalence queries. In particular, we found that **equivalence tests account for 89.1% of tests in MAT, 59.8% in** C$\mathcal{E}$AL**, and 66.3% on average**.

Perhaps more surprisingly, the commonly accepted ordering of Learner efficiencies did not surface in our experiments. From theoretically most performant to least we have $\mathtt{L}^{\#}$, TTT, KV, and $\mathtt{L}^{\star}$, based on complexity analyses in MAT. Through our experiments we have found that, at least in our particular case of black-box learning (i.e. learning from SUL tests only) of Mealy machines with noise and randomized testing algorithms, this ordering cannot be seen in the results. The best configurations for each framework were not consistent on which algorithm performed the best. Not only was there no clear ”winning” algorithm, we found no pattern based on noise, target and alphabet size, or number of repeats that had a strong enough correlation to the better performance of any one algorithm.

We believe that this is not a flaw of the complexity analyses themselves. It is simply that complexity analyses in MAT abstract away the biggest cost of learning: equivalence tests. It may be that more recent algorithms have a theoretical (and membership query based) advantage over classic algorithms, however the nature of randomized equivalence oracles seems to be a bigger agent of chaos, and a good or bad run of the equivalence oracle quickly overshadows the small advantage that some algorithms may have.

5 Related Work

There has been extensive work on finding ways of applying classic learning algorithms like $\mathtt{L}^{\star}$ [4], KV [24], TTT [22], and $\mathtt{L}^{\#}$ [33] to real world systems such as passports [2], network protocol implementations [15, 18, 19], and bank cards [1]. All these works rely on ad-hoc implementations of noise handling which is inefficient and not formalized in the MAT framework. One of the goals of our framework, which replaces the teacher with the SUL and the Reviser, is to discuss how noise can be dealt with in the learning process, independently of the type of Learner being used. The LearnLib library [23] provides *caches* that can be placed in the learning environment to avoid the repetition of queries, much like observation trees. Note that our Reviser agent goes further than LearnLib as it provides the ability to act as membership and equivalence oracles, test the system, and act on conflicts by pruning the learner’s data structure in an efficient (query-wise) and correct manner.

There has also been previous work in improving the efficiency of model learning strategies for mutating targets by reusing previously learned behavior, using *adaptive* learning algorithms [11,14,17,21,35]. These algorithms work by being able to start learning with pre-seeded information of previous runs that has been confirmed to still apply in the current target, or by being able to filter this information themselves if it is found to no longer apply. Additionally, there has been some work on *Lifelong Learning* [7], where model learning and model checking are used together to run over the development lifecycle of a system. This allows for the quick discovery of bugs in the development cycle. However, when these are found and corrected, learning needs to be *manually restarted*.

Our model learning framework improves on these two lines of work, being able to autonomously correct itself when faced with conflicts. It can do so without any notification of mutations in the system, allowing it to be applied to complete closed-box systems, unlike the current state-of-the-art adaptive algorithm [17]. Additionally, it is capable of continuously checking for changes in the system, much like Lifelong Learning, but requiring no human interaction on system changes. These characteristics make it resilient to real world noise, allowing the learner to correct itself as it identifies the correct behavior.

Our work participates in the current trend trying to link learning to testing, which spans communities, e.g. formal approaches [3], genetic approaches [31], and fuzzing [36]. In this context, C$\mathcal{E}$AL provides a modular framework upon which other techniques can be added. In active model learning, this trend also matches the interest in observation-tree based algorithms [30, 33], which we instantiate in C$\mathcal{E}$AL. The role of observation structures in learning and testing is a long-standing lore [8] that can be leveraged to enhance the learning approach and its modularity with testing methods.

6 Conclusion and Future Work

This paper explores efficient ways to handle conflicts during active learning. We build on the idea that recovering from conflicts is best done by splitting information collection and the construction of the Learner's data structure, two operations that are conflated in MAT.

We introduce the Conflict Aware Active Automata Learning (C$\mathcal{E}$AL) framework as an alternative to MAT. C$\mathcal{E}$AL directly represents the SUL and introduces a *Reviser* tasked with testing it, storing and curating the observations. C$\mathcal{E}$AL provides a way to accept *some* conflicts to reach the Learner and to recover from them without requiring to test the SUL anew.

To test the efficiency of C$\mathcal{E}$AL, we conducted a large body of experiments on real targets using several state-of-the-art algorithms. We found that **not only does** C$\mathcal{E}$AL **always improves on MAT in terms of success rates**, obtaining an overall **success rate of 95,5% against MAT's 79,5%** it most importantly **enables the learning of previously un-learnable SULs**, typically complex systems plagued with a high level of noise. Our experiments further put into light the impact of equivalence tests, both in terms of variability of the results and sheer cost, with an average of **66.3% of testing cost spent on equivalence**.

In the future, we would like to explore the use and design of testing algorithms for active learning, as their efficiency seems to be able to overshadow the difference between learning algorithms. C$\mathcal{E}$AL's modular nature also allows us to seamlessly build a *gray-box* environment i.e. to gain information from different sources in the Reviser (e.g. specifications, access to source code). This would offset the cost of equivalence queries by using cheaper sources of observations when searching for counterexamples.

Assessing the efficiency of C$\mathcal{E}$AL on a real case of mutating targets would be of interest, as an evaluation, as an opportunity to fine-tune the framework for such task, and as a demonstration of the improved reach of active learning. Similarly, testing the `Most Frequent` update strategy in practice against high noise levels would be of interest.

Acknowledgements

This work was partially supported by the EPSRC Standard Grant CLeVer (EP/S028641/1), ERC grant Autoprobe (no. 101002697), and a Royal Society Wolfson Fellowship. The authors would like to also thank Denis Timm and the TSG and HPC groups at UCL Computer Science department for their support in running the experiments in this paper.

References

[1] F. Aarts, J. De Ruiter, and E. Poll. Formal Models of Bank Cards for Free. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 461–468, March 2013. doi:10.1109/ICSTW.2013.60.

[2] Fides Aarts, Julien Schmaltz, and Frits Vaandrager. Inference and Abstraction of the Biometric Passport. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, Lecture Notes in Computer Science, pages 673–686, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-16558-0_54.

[3] Bernhard K. Aichernig, Wojciech Mostowski, Mohammad Reza Mousavi, Martin Tappler, and Masoumeh Taromirad. *Model Learning and Model-Based Testing*, pages 74 – 100. Lecture Notes in Computer Science. Springer Nature, 7 2018. doi:10.1007/978-3-319-96562-8_3.

[4] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, November 1987. URL: http://www.sciencedirect.com/science/article/pii/0890540187900526, doi:10.1016/0890-5401(87)90052-6.

[5] Kousar Aslam, Loek Cleophas, Ramon Schiffelers, and Mark van den Brand. Interface protocol inference to aid understanding legacy software components. *Software and Systems Modeling*, 19(6):1519–1540, November 2020. doi:10.1007/s10270-020-00809-2.

[6] John Backes, Byron Cook, Andrew Gacek, Neha Rungta, and Michael W. Whalen. One-click formal methods. In *ICST 2020*, 2019. URL: https://www.amazon.science/publications/one-click-formal-methods.

[7] Alexander Bainczyk, Bernhard Steffen, and Falk Howar. Lifelong Learning of Reactive Systems in Practice. In Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, and Einar Broch Johnsen, editors, *The Logic of Software. A Tasting Menu of Formal Methods*, volume 13360, pages 38–53. Springer International Publishing, Cham, 2022. Series Title: Lecture Notes in Computer Science. doi:10.1007/978-3-031-08166-8_3.

[8] Therese Berg, Olga Grinchtein, Bengt Jonsson, Martin Leucker, Harald Raffelt, and Bernhard Steffen. On the correspondence between conformance testing and regular inference. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering*, pages 175–189, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:978-3-540-31984-9_14.

[9] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, pages 836–850, Virtual Event Germany, October 2021. ACM. doi:10.1145/3477132.3483540.

[10] Quentin Carbonneaux, Noam Zilberstein, Christoph Klee, Peter W. O'Hearn, and Francesco Zappa Nardelli. Applying formal verification to microkernel IPC at meta. In Andrei Popescu and Steve Zdancewic, editors, *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 116–129. ACM, 2022. doi:10.1145/3497775.3503681.

[11] Sagar Chaki, Edmund Clarke, Natasha Sharygina, and Nishant Sinha. Verification of evolving software via component substitutability analysis. *Formal Methods in System Design*, 32(3):235–266, June 2008. doi:10.1007/s10703-008-0053-x.

[12] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. Continuous formal verification of amazon s2n. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 430–446, Cham, 2018. Springer International Publishing. doi:10.1007/9783319961422_26.

[13] Byron Cook, Kareem Khazem, Daniel Kroening, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. Model checking boot code from aws data centers. In *CAV 2018*, 2018. URL: https://www.amazon.science/publications/model-checking-boot-code-from-aws-data-centers.

[14] Carlos Diego Nascimento Damasceno, Mohammad Reza Mousavi, and Adenilso da Silva Simão. Learning to Reuse: Adaptive Model Learning for Evolving Systems. In *IFM*, volume 11918 of *LNCS*, pages 138–156. Springer, 2019. `doi:10.1007/978-3-030-34968-4_8`.

[15] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. Prognosis: closed-box analysis of network protocol implementations. In *SIGCOMM*, pages 762–774. ACM, 2021. `doi:10.1145/3452296.3472938`.

[16] Tiago Ferreira, Léo Henry, Raquel Fernandes da Silva, and Alexandra Silva. Conflict-aware active automata learning, 2023. `arXiv:2308.14781`.

[17] Tiago Ferreira, Gerco van Heerdt, and Alexandra Silva. *Tree-Based Adaptive Model Learning*, pages 164–179. Springer Nature Switzerland, Cham, 2022. `doi:10.1007/978-3-031-15629-8_10`.

[18] Paul Fiterau-Brostean, Ramon Janssen, and Frits W. Vaandrager. Combining Model Learning and Model Checking to Analyze TCP Implementations. In *CAV*, volume 9780 of *LNCS*, pages 454–471. Springer, 2016. `doi:10.1007/978-3-319-41540-6_25`.

[19] Paul Fiterau-Brostean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits W. Vaandrager, and Patrick Verleg. Model learning and model checking of SSH implementations. In *SPIN*, pages 142–151. ACM, 2017. `doi:10.1145/3092282.3092289`.

[20] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991. `doi:10.1109/32.87284`.

[21] David Huistra, Jeroen Meijer, and Jaco van de Pol. Adaptive Learning for Learn-Based Regression Testing. In Falk Howar and Jiří Barnat, editors, *Formal Methods for Industrial Critical Systems*, volume 11119, pages 162–177. Springer International Publishing, Cham, 2018. Series Title: Lecture Notes in Computer Science. `doi:10.1007/978-3-030-00244-2_11`.

[22] Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, volume 8734, pages 307–322. Springer International Publishing, Cham, 2014. Series Title: Lecture Notes in Computer Science. `doi:10.1007/978-3-319-11164-3_26`.

[23] Malte Isberner, Falk Howar, and Bernhard Steffen. The Open-Source LearnLib. In *CAV*, volume 9206 of *LNCS*, pages 487–495, 2015. `doi:10.1007/978-3-319-21690-4_32`.

[24] Michael J. Kearns and Umesh Virkumar Vazirani. *An introduction to computational learning theory*. MIT Press, Cambridge, Mass, 1994. `doi:10.7551/mitpress/3897.001.0001`.

[25] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.*, 6(OOPSLA1), apr 2022. `doi:10.1145/3527325`.

[26] D. Lee and M. Yannakakis. Testing finite-state machines: state identification and verification. *IEEE Transactions on Computers*, 43(3):306–320, 1994. `doi:10.1109/12.272431`.

[27] Daniel Neider, Rick Smetsers, Frits Vaandrager, and Harco Kuppens. Benchmarks for Automata Learning and Conformance Testing. In Tiziana Margaria, Susanne Graf, and Kim G. Larsen, editors, *Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*, pages 390–416. Springer International Publishing, Cham, 2019. `doi:10.1007/978-3-030-22348-9_23`.

[28] Ronald L. Rivest and Robert E. Schapire. Inference of Finite Automata Using Homing Sequences. *Inf. Comput.*, 103:299–347, 1993. `doi:10.1006/inco.1993.1021`.

[29] Wouter Smeenk, Joshua Moerman, Frits Vaandrager, and David N. Jansen. Applying automata learning to embedded control software. In Michael Butler, Sylvain Conchon, and Fatiha Zaïdi, editors, *Formal Methods and Software Engineering*, pages 67–83, Cham, 2015. Springer International Publishing. `doi:10.1007/978-3-319-25423-4_5`.

[30] Michal Soucha and Kirill Bogdanov. Observation Tree Approach: Active Learning Relying on Testing. *The Computer Journal*, 63(9):1298–1310, 07 2019. `doi:10.1093/comjnl/bxz056`.

[31] Martin Tappler, Bernhard K. Aichernig, Kim Guldstrand Larsen, and Florian Lorber. Time to learn - learning timed automata from tests. In Étienne André and Mariëlle Stoelinga, editors, *Formal Modeling and Analysis of Timed Systems - 17th International Conference, FORMATS 2019, Amsterdam, The Netherlands, August 27-29, 2019, Proceedings*, volume 11750 of *Lecture Notes in Computer Science*, pages 216–235. Springer, 2019. `doi:10.1007/978-3-030-29662-9\_13`.

[32] Frits W. Vaandrager. Model learning. *Commun. ACM*, 60:86–95, 2017. `doi:10.1145/2967606`.

[33] Frits W. Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A New Approach for Active Automata Learning Based on Apartness. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 223–243. Springer, 2022. `doi:10.1007/978-3-030-99524-9_12`.

[34] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. CacheQuery: Learning Replacement Policies from Hardware Caches. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 519–532, New York, NY, USA, 2020. Association for Computing Machinery. event-place: London, UK. `doi:10.1145/3385412.3386008`.

[35] Stephan Windmüller, Johannes Neubauer, Bernhard Steffen, Falk Howar, and Oliver Bauer. Active Continuous Quality Control. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering*, CBSE '13, pages 111–120, New York, NY, USA, 2013. Association for Computing Machinery. event-place: Vancouver, British Columbia, Canada. `doi:10.1145/2465449.2465469`.

[36] Andreas Zeller. Learning the language of failure. 2020 CASA Disinguished Lecture, 2020. URL: `https://andreas-zeller.info/assets/CASA-2020-Learning-the-Language-of-Failure.pdf`.