Approximating equivalence queries through membership queries

Alexis Pinson
(supervisor : Adrien Pommellet)

July 2024

Active learning consists in modelling as an automaton the behaviour of a black box system whose inner logic is unknown to the observer - be it an unknown program, a neural network, etc. To this end, the learner will rely on two different classes of queries: membership queries (”Does a word belong to the language accepted by the black box?”) and equivalence queries (”Does the automaton accurately model the system under learning?”). The former class of requests can be handled by merely running them on the black box. On the other hand, it is theoretically impossible to answer an equivalence query without knowing the inner workings of the system we are trying to model in the first place. For this very reason, actual active learning libraries tends to approximate them with membership queries.

Inspired by Joshua Moerman (2019) [Moe19]

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
alexis.pinson@epita.fr – http://www.lre.epita.fr/

Copying this document

Contents

1 Introduction

Model learning is an automated technique to construct a state-based model – often a type of automaton – from a black box system. The goal of this technique can be manifold: it can be used to reverse-engineer a system, to find bugs in it, to verify properties of the system, or to understand the system in one way or another. It is not just random testing: the information learned during the interaction with the system is actively used to guide following interactions. Additionally, the information learned can be inspected and analysed.

But before we get ahead of ourselves, we should first understand what we mean by learning, as learning means very different things to different people. In educational science, learning may involve concepts such as teaching, blended learning, and interdisciplinarity. Data scientists may think of data compression, feature extraction, and neural networks. In this article we are mostly concerned with software verification. But even in the field of verification several types of learning are relevant.

1.1 Model Learning

In the context of software verification, we often look at stateful computations with inputs and outputs. For this reason, it makes sense to look at words, or traces. For an alphabet Σ, we denote the set of words by Σ^*.

The learning problem is defined as follows. There is some fixed, but unknown, language $\mathcal{L} \subseteq \Sigma^*$. This language may define the behaviour of a software component, a property in model checking, a set of traces from a protocol, etc. We wish to infer a description of $\mathcal{L}$ after only having observed a small part of this language. For example, we may have seen hundred words belonging to the language and a few which do not belong to the language. Then concluding with a good description of $\mathcal{L}$ is difficult, as we are missing information about the infinitely many words we have not observed.

Such a learning problem can be stated and solved in a variety of ways. We will have a look at how a learning algorithm can interact with the software. So it makes sense to study a learning paradigm which allows for queries, and not just a data set of samples. A typical query learning framework was established by Angluin (1987) [D87]. In her framework, the learning algorithm may pose two types of queries to a teacher, or oracle:

Membership queries (MQ) The learner poses such a query by providing a word $w \in \Sigma^*$ to the teacher. The teacher will then reply whether $w \in \mathcal{L}$ or not. This type of query is often generalised to more general output, in these cases we consider $\mathcal{L} : \Sigma^* \to O$ and the teacher replies with $\mathcal{L}(w)$. In some papers, such a query is then called an *output query*.

Equivalence queries (EQ) The learner can provide a hypothesised description H of $\mathcal{L}$ to the teacher. If the hypothesis is correct, the teacher replies with yes. If, however, the hypothesis is incorrect, the teacher replies with no together with a counterexample, i.e., a word which is in $\mathcal{L}$ but not in the hypothesis or vice versa.

By posing many such queries, the learner algorithm is supposed to converge to a correct model. This type of learning is hence called *exact learning*. Angluin (1987) [D87] showed that one can do this efficiently for deterministic finite automata (DFAs), when $\mathcal{L}$ is in the class of regular languages.

It should be clear why this is called query learning or active learning. The learning algorithm initiates interaction with the teacher by posing queries, it may construct its own data points and ask for their corresponding label. Active learning is in contrast to passive learning where all observations are given to the algorithm up front.

1.2 Applications of Model Learning

This is a non-exhaustive list of applications of model learning.

Bug finding in protocols. A prominent example is by Fiterău-Broștean, et al. (2016) [RW16]. They learn models of TCP implementations – both clients and server sides. Interestingly, they found bugs in the (closed source) Windows implementation. Later, Fiterău-Broștean and Howar (2017) [FH17] also found a bug in the sliding window of the Linux implementation of TCP. Other protocols have been learned as well, such as the MQTT protocol by Tappler, et al. (2017) [KR17], TLS by de Ruiter and Poll (2015) [RE15], and SSH by Fiterău-Broștean, et al. (2017) [RP17]. Many of these applications reveal bugs by learning a model and consequently apply model checking. The combination of learning and model checking was first described by Peled, et al. (2002) [YM02].

Bug finding in smart cards Aarts, et al. (2013) [RE13] learn the software on smart cards of several Dutch and German banks. These cards use the EMV protocol, which is run on the card itself. So this is an example of a real black box system, where no other monitoring is possible and no code is available. No vulnerabilities were found, although each card had a slightly different state machine. The e.dentifier, a card reader implementing a challenge-response protocol, has been learned by Chalupar, et al. (2014) [ER14]. They built a Lego machine which could automatically press buttons and the researchers found a security flaw in this card reader.

Regression testing Hungar, et al. (2003) [OB03] describe the potential of automata learning in regression testing. The aim is not to find bugs, but to monitor the development process of a system. By considering the differences between models at different stages, one can generate regression tests.

Refactoring legacy software Model learning can also be used in order to verify refactored software. Schuts, et al. (2016) [JW16] have applied this at a project within Philips. They learn both an old version and a new version of the same component. By comparing the learned models, some differences could be seen. This gave developers opportunities to solve problems before replacing the old component with the new one.

2 Black Box Testing

An important step in automata learning is equivalence checking. Normally, this is abstracted away and done by an oracle, but we intend to implement such an oracle ourselves for our applications. Concretely, the problem we need to solve is that of conformance checking as it was first described by Moore (1956) [F56].

The problem is as follows: Given the description of a finite state machine and a black box system, does the system behave exactly as the description? We wish to determine this by running experiments on the system (as it is black box). It should be clear that this is a hopelessly difficult task, as an error can be hidden arbitrarily deep in the system. That is why we often assume some knowledge of the system. In this article we often assume a bound on the number of states of the system. Under these conditions, Moore (1956) [F56] already solved the problem. Unfortunately, his experiment is exponential in size, or in his own words: “fantastically large.”

Years later, Chow (1978) [S78] and Vasilevskii (1973) [P73] independently designed efficient experiments. In particular, the set of experiments is polynomial in the number of states. More background and other related problems, as well as their complexity results, are well exposed in a survey of Lee and Yannakakis (1994) [DM94].

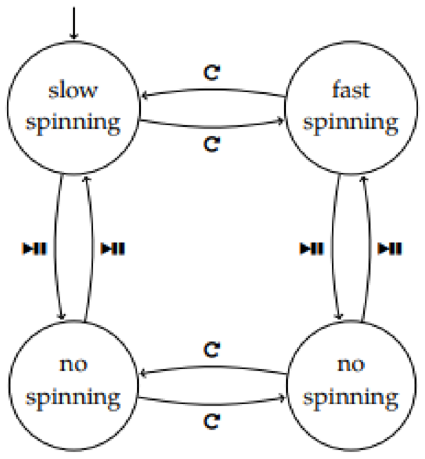


Figure 1: Behaviour of a record player modelled as a finite state machine.

To give an example of conformance checking, we model a record player as a finite state machine. We will not model the audible output – that would depend not only on the device, but also the record one chooses to play. Instead, the only observation we can make is looking at how fast the turntable spins. The device has two buttons: a start-stop button (⏯) and a speed button (↻) which toggles between $33\frac{1}{3}$ rpm and 45 rpm. When turned on, the system starts playing immediately at $33\frac{1}{3}$ rpm – this is useful for DJing. The intended behavior of the record player has four states as depicted in Figure 1.

Let us consider some faults which could be present in an implementation with four states. In Figure 2, two flawed record players are given. In the first (Figure 2a), the sequence ⏯↻↻ leads us to the wrong state. However, this is not immediately observable; the turntable is in a non-spinning state as it should be. The fault is only visible when we press ⏯once more: now the turntable is spinning fast instead of slow. The sequence ⏯↻↻⏯is a counterexample. In the second example (Figure 2b), the fault is again not immediately obvious: after pressing ↻⏯we are in the wrong state as observed by pressing ⏯. Here, the counterexample is ↻⏯⏯.

When a model of the implementation is given, it is not hard to find counterexamples. However, in a black box setting we do not have such a model. In order to test whether a black box system is equivalent to a model, we somehow need to test all possible counterexamples. In this example, a test suite should include sequences such as ⏯↻↻⏯and ↻⏯⏯.

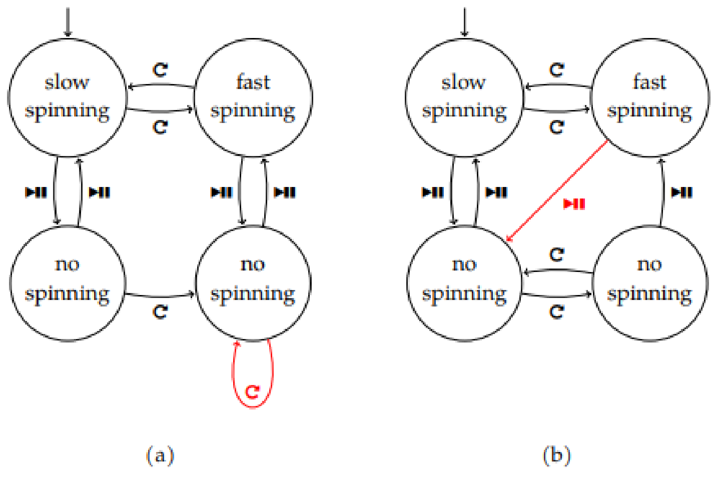


Figure 2: Two faulty record players

2.1 Mealy machines

We will focus on Mealy machines, as those capture many protocol specifications and reactive systems.

We fix finite alphabets I and O of inputs and outputs respectively. We use the usual notation for operations on sequences (also called words): uv for the concatenation of two sequences $u, v \in I^*$ and $|u|$ for the length of u. For a sequence $w = uv$ we say that u and v are a prefix and suffix respectively.

Definition A (deterministic and complete) Mealy machine M consists of a finite set of states S, an initial state $s_0 \in S$, and two functions:

- a transition function $\delta : S \times I \to S$, and
- an output function $\lambda : S \times I \to O$.

Both the transition function and output function are extended inductively to sequences as $\delta : S \times I^* \to S$ and $\lambda : S \times I^* \to O^*$:

$$
\begin{aligned}
&\delta(s, \epsilon) = s \\
&\lambda(s, \epsilon) = \epsilon \\
&\delta(s, aw) = \delta(\delta(s, a), w) \\
&\lambda(s, aw) = \lambda(s, a)\lambda(\delta(s, a), w)
\end{aligned}
$$

The behaviour of a state s is given by the output function $\lambda(s, -) : I^* \to O^*$. Two states s and t are equivalent if they have equal behaviours, written $s \sim t$, and two Mealy machines are equivalent if their initial states are equivalent.
An example Mealy Machine is given in Figure 3.

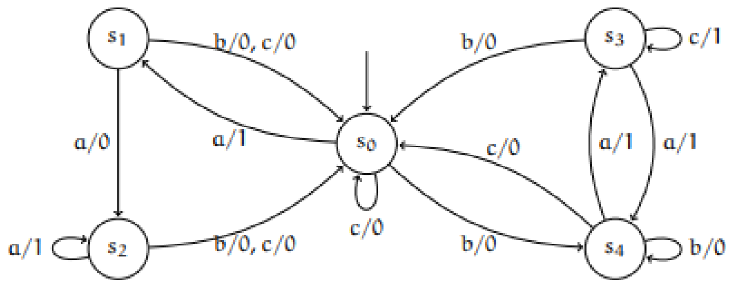


Figure 3: An example specification with input I = {a, b, c} and output O = {0, 1}

2.2 Completeness of test suites

In conformance testing we have a specification modelled as a Mealy machine and an implementation (the system under test, or SUT) which we assume to behave as a Mealy machine. Tests, or experiments, are generated from the specification and applied to the implementation. We assume that we can reset the implementation before every test. If the output is different from the specified output, then we know the implementation is flawed. The goals is to test as little as possible, while covering as much as possible.

A test suite is nothing more than a set of sequences. We do not have to encode outputs in the test suite, as those follow from the deterministic specification.

Definition A test suite is a finite subset $T \subseteq I^*$

A test $t \in T$ is called maximal if it is not a proper prefix of another test $s \in T$. We denote the set of maximal tests of T by $\max(T)$. The maximal tests are the only tests in T we actually have to apply to our SUT as we can record the intermediate outputs.

No test suite is complete. Consider the specification in Figure 4a. This machine will always outputs a cup of coffee – when given money. For any test suite we can make a faulty implementation which passes the test suite. A faulty implementation might look like Figure 4b, where the machine starts to output beers after n steps (signalling that it's the end of the day), where n is larger than the length of the longest sequence in the suite. This shows that no test-suite can be complete and it justifies the following definition.

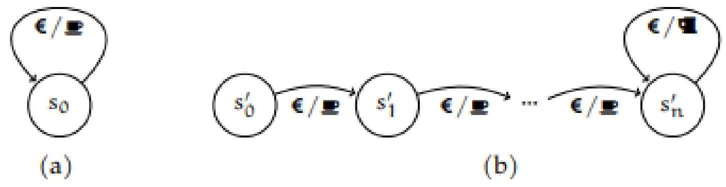


Figure 4: A basic example showing that finite test suites are incomplete. The implementation on the right will pass any test suite if we choose n big enough

Definition Let M be a Mealy machine and T be a test suite. We say that T is m-complete (for M) if for all inequivalent machines M' with at most m states there exists a $t \in T$ such that $\lambda(s_0, t) \neq \lambda'(s'_0, t)$.

We are often interested in the case of m-completeness, where $m = n + k$ for some $k \in \mathbb{N}$ and n is the number of states in the specification. Here k will stand for the number of extra states we can test.

Note the order of the quantifiers in the above definition. We ask for a single test suite which works for all implementations of bounded size. This is crucial for black box testing, as we do not know the implementation, so the test suite has to work for all of them

2.3 W-method

Before talking about W-method, you must know what a characterisation set is.

Definition Given two states s, t in M, we say that w is a separating sequence if $\lambda(s, w) \neq \lambda(t, w)$.

Definition A set of sequences W is called a characterisation set if it contains a separating sequence for each pair of inequivalent states in M.

We fix a state cover P and take the transition cover $Q = P \cdot I$.
After the work of Moore (1956) [F56], it was unclear whether a test suite of polynomial size could exist. He presented a finite test suite which was complete, however it was exponential in size. Both Chow (1978) [S78] and Vasilevskii (1973) [P73] independently prove that test suites of polynomial size exist. The W-method is a very structured test suite construction. It is called the W-method as the characterisation set is often called W.

Definition Given a characterisation set W, we define the W test suite as

$$T_W = (P \cup Q) \cdot I^{\leq k} \cdot W$$

This tests the machine in two phases. For simplicity, we explain these phases when $k = 0$. The first phase consists of the tests $P \cdot W$ and tests whether all states of the specification are (roughly) present in the implementation. The second phase is $Q \cdot W$ and tests whether the successor states are correct. Together, these two phases put enough constraints on the implementation to know that the implementation and specification coincide (provided that the implementation has no more states than the specification).

Let us compute the previous test suite on the specification in Figure 3. We will be testing without extra states, i.e., we construct 5-complete test suites. We start by defining the state and transition cover. For this, we take all shortest sequences from the initial state to the other states. This state cover is depicted in Figure 5. The transition cover is simply constructed by extending each access sequence with another symbol.

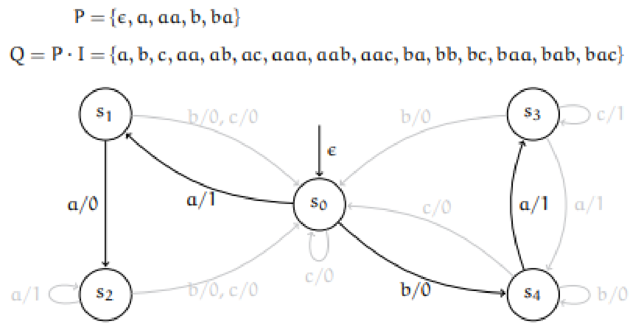


Figure 5: A state cover for the specification from Figure 3

The set $W = \{aa, ac, c\}$ is a characterisation set. The W-method, which simply combines $P \cup Q$ with W, gives the following test suite of size 169:

$$\begin{aligned} T_W = \{&aaaaa, aaaac, aaac, aabaa, aabac, aabc, aacaa, \\ &aacac, aacc, abaa, abac, abc, acaa, acac, acc, baaaa, \\ &baaac, baac, babaa, babac, babc, bacaa, bacac, bacc, \\ &bbaa, bbac, bbc, bcaa, bcac, bcc, caa, cac, cc\} \end{aligned}$$

3 Generation of W

In diverse areas of computer science and engineering, systems can be modelled by finite state machines (FSMs). One of the cornerstones of automata theory is minimisation of such machines – and many variations thereof. In this process one obtains an equivalent minimal FSM, where states are different if and only if they have different behaviour. The first to develop an algorithm for minimisation was Moore (1956) [F56]. His algorithm has a time complexity of $\mathcal{O}(mn)$, where m is the number of transitions, and n is the number of states of the FSM. Later, Hopcroft (1971) [E71] improved this bound to $\mathcal{O}(m \log n)$.

Minimisation algorithms can be used as a framework for deriving a set of separating sequences that show why states are inequivalent. The separating sequences in Moore's framework are of minimal length (Gill, 1962 [A62]). Obtaining minimal separating sequences in Hopcroft's framework, however, is a non-trivial task. In this part of this article, we present an algorithm for finding such minimal separating sequences for all pairs of inequivalent states of a FSM in $\mathcal{O}(m \log n)$ time.

Coincidentally, Bonchi and Pous (2013) [FD13] recently introduced a new algorithm for the equally fundamental problem of proving equivalence of states in non-deterministic automata. As both their and our work demonstrate, even classical problems in automata theory can still offer surprising research opportunities. Moreover, new ideas for well-studied problems may lead to algorithmic improvements that are of practical importance in a variety of applications.

One such application for our work is in conformance testing. Here, the goal is to test if a black box implementation of a system is functioning as described by a given FSM. It consists of applying sequences of inputs to the implementation, and comparing the output of the system to the output prescribed by the FSM. Minimal separating sequences are used in many test generation methods (Dorofeeva, et al., 2010 [RN10]). Therefore, our algorithm can be used to improve these methods.

In this part of the article, we will use only Mealy machines but all this algorithms can be adapted for DFA.

Definition States s and t are equivalent if $\lambda(s, x) = \lambda(t, x)$ for all $x \in I^*$.

We are interested in the case where s and t are not equivalent, i.e., inequivalent. If all pairs of distinct states of a machine M are inequivalent, then M is minimal.

Definition A separating sequence for states s and t in s is a sequence $x \in I^*$ such that $\lambda(s, x) \neq \lambda(t, x)$. We say x is minimal if $|y| \geq |x|$ for all separating sequences y for s and t.

A separating sequence always exists if two states are inequivalent, and there might be multiple minimal separating sequences. Our goal is to obtain minimal separating sequences for all pairs of inequivalent states of M

Both Moore's algorithm and Hopcroft's algorithm work by means of partition refinement.

A partition P of S is a set of pairwise disjoint non-empty subsets of S whose union is exactly S. Elements in P are called blocks. If P and P' are partitions of S, then P' is a refinement of P if every block of P' is contained in a block of P. A partition refinement algorithm constructs the finest partition under some constraint. In our context the constraint is that equivalent states belong to the same block.

Definition A partition is valid if equivalent states are in the same block.

Partition refinement algorithms for FSMs start with the trivial partition $P = \{S\}$, and iteratively refine P until it is the finest valid partition (where all states in a block are equivalent). The blocks of such a complete partition form the states of the minimised FSM, whose transition and output functions are well-defined because states in the same block are equivalent.

Let B be a block and a be an input. There are two possible reasons to split B (and hence refine the partition). First, we can split B with respect to output after a if the set $\lambda(B, a)$ contains more than one output. Second, we can split B with respect to the state after a if there is no single block B' containing the set $\delta(B, a)$. In both cases it is obvious what the new blocks are: in the first case each output in $\lambda(B, a)$ defines a new block, in the second case each block containing a state in $\delta(B, a)$ defines a new block. Both types of refinement preserve validity.

Partition refinement algorithms for FSMs first perform splits with respect to output, until there are no such splits to be performed. This is precisely the case when the partition is acceptable.

Definition A partition is acceptable if for all pairs s, t of states contained in the same block and for all inputs $a \in I$, $\lambda(s, a) = \lambda(t, a)$.

Any refinement of an acceptable partition is again acceptable. The algorithm continues performing splits with respect to state, until no such splits can be performed. This is exactly the case when the partition is stable.

Definition A partition is stable if it is acceptable and for any input $a \in I$ and states s and t that are in the same block, states $\delta(s, a)$ and $\delta(t, a)$ are also in the same block.

Since an FSM has only finitely many states, partition refinement will terminate. The output is the finest valid partition which is acceptable and stable. For a more formal treatment on partition refinement we refer to Gries (1973) [D73].

3.1 Naive approach

Both types of splits described above can be used to construct a separating sequence for the states that are split. In a split with respect to the output after a, this sequence is simply a. In a split with respect to the state after a, the sequence starts with an a and continues with the separating sequence for states in $\delta(B, a)$. In order to systematically keep track of this information, we maintain a splitting tree. The splitting tree was introduced by Lee and Yannakakis (1994) [DM94] as a data structure for maintaining the operational history of a partition refinement algorithm.

Definition A splitting tree for M is a rooted tree T with a finite set of nodes with the following properties:

- Each node u in T is labelled by a subset of S, denoted $l(u)$.
- The root is labelled by S.
- For each inner node u, $l(u)$ is partitioned by the labels of its children.
- Each inner node u is associated with a sequence $\sigma(u)$ that separates states contained in different children of u.

We use $C(u)$ to denote the set of children of a node u. The lowest common ancestor (lca) for a set $S' \subseteq S$ is the node u such that $S' \subseteq l(u)$ and $S' \nsubseteq l(v)$ for all $v \in C(u)$ and is denoted by $\text{lca}(S')$. For a pair of states s and t we use the shorthand $\text{lca}(s, t)$ for $\text{lca}(\{s, t\})$.

The labels $l(u)$ can be stored as a refinable partition data structure (Valmari & Lehtinen, 2008 [AP08]). This is an array containing a permutation of the states, ordered so that states in the same block are adjacent. The label $l(u)$ of a node then can be indicated by a slice of this array. If node u is split, some states in the slice $l(u)$ may be moved to create the labels of its children, but this will not change the set $l(u)$.

A splitting tree T can be used to record the history of a partition refinement algorithm because at any time the leaves of T define a partition on S, denoted $P(T)$. We say a splitting tree T is valid (resp. acceptable, stable, complete) if $P(T)$ is as such. A leaf can be expanded in one of two ways, corresponding to the two ways a block can be split. Given a leaf u and its block $B = l(u)$ we define the following two splits:

(split-output) Suppose there is an input a such that B can be split with respect to output after a. Then we set $\sigma(u) = a$, and we create a node for each subset of B that produces the same output x on a. These nodes are set to be children of u.

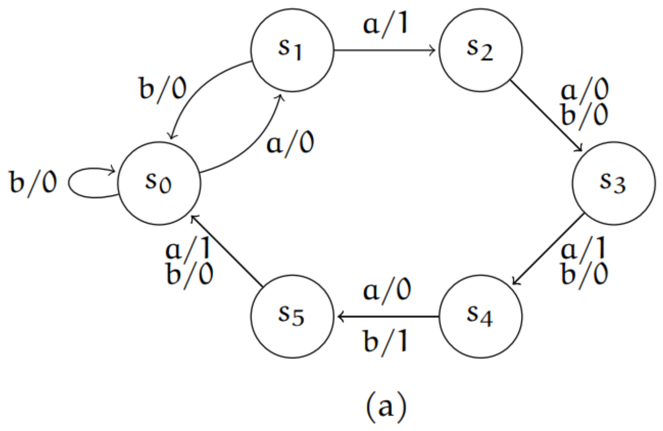
(split-state) Suppose there is an input a such that B can be split with respect to the state after a. Then instead of splitting B as described before, we proceed as follows. First, we locate the node $v = \text{lca}(\delta(B, a))$. Since v cannot be a leaf, it has at least two children whose labels contain elements of $\delta(B, a)$. We can use this information to expand the tree as follows. For each node w in $C(v)$ we create a child of u labelled $\{s \in B \mid \delta(s, a) \in l(w)\}$ if the label contains at least one state. Finally, we set $\sigma(u) = a\sigma(v)$.

A straight-forward adaptation of partition refinement for constructing a stable splitting tree for M is shown in Algorithm 1. The termination and the correctness of the algorithm are preserved. It follows directly that states are equivalent if and only if they are in the same label of a leaf node.

Algorithm 1 Constructing a stable splitting tree

```
Require: An FSM $M$
Ensure: A valid and stable splitting tree $T$
1: initialise $T$ to be a tree with a single node labelled $S$
2: repeat
3:     find $a \in I$, $B \in P(T)$ such that we can split $B$ with respect to output $\lambda(\cdot, a)$
4:     expand the $u \in T$ with $l(u) = B$ as described in (split-output)
5: until $P(T)$ is acceptable
6: repeat
7:     find $a \in I$, $B \in P(T)$ such that we can split $B$ with respect to state $\delta(\cdot, a)$
8:     expand the $u \in T$ with $l(u) = B$ as described in (split-state)
9: until $P(T)$ is stable
```

Example Figure 6 shows an FSM and a complete splitting tree for it. This tree is constructed by Algorithm 1 as follows. First, the root node is labelled by $\{s_0, \ldots, s_5\}$. The even and uneven states produce different outputs after a, hence the root node is split. Then we note that s_4 produces a different output after b than s_0 and s_2, so $\{s_0, s_2, s_4\}$ is split as well. At this point T is acceptable: no more leaves can be split with respect to output. Now, the states $\delta(\{s_1, s_3, s_5\}, a)$ are contained in different leaves of T. Therefore, $\{s_1, s_3, s_5\}$ is split into $\{s_1, s_5\}$ and $\{s_3\}$ and associated with sequence ab. At this point, $\delta(\{s_0, s_2\}, a)$ contains states that are in both children of $\{s_1, s_3, s_5\}$, so $\{s_0, s_2\}$ is split and the associated sequence is aab. We continue until T is complete.



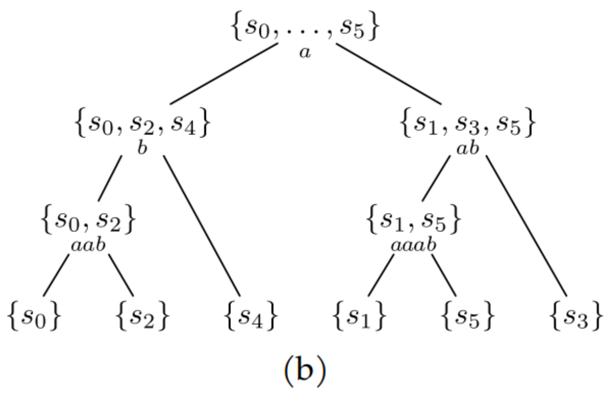


Figure 6: An FSM (a) and a complete splitting tree for it (b).

3.2 Minimal approach

In the previous section, we have described an algorithm for constructing a complete splitting tree. This algorithm is non-deterministic, as there is no prescribed order on the splits. In this section we order them to obtain minimal separating sequences.

Let u be a non-root inner node in a splitting tree, then the sequence $\sigma(u)$ can also be used to split the parent of u. This allows us to construct splitting trees where children will never have shorter sequences than their parents, as we can always split with those sequences first. Trees obtained in this way are guaranteed to be layered, which means that for all nodes u and all $u' \in C(u)$, $|\sigma(u)| \leq |\sigma(u')|$. Each layer consists of nodes for which the associated separating sequences have the same length.

Our approach for constructing minimal sequences is to ensure that each layer is as large as possible before continuing to the next one. This idea is expressed formally by the following definitions.

Definition A splitting tree T is k-stable if for all states s and t in the same leaf we have $\lambda(s, x) = \lambda(t, x)$ for all $x \in I^{\leq k}$.

Definition 9. A splitting tree T is minimal if for all states s and t in different leaves $\lambda(s, x) \neq \lambda(t, x)$ implies $|x| \geq |\sigma(\text{lca}(s, t))|$ for all $x \in I^*$.

Minimality of a splitting tree can be used to obtain minimal separating sequences for pairs of states. If the tree is in addition stable, we obtain minimal separating sequences for all inequivalent pairs of states. Note that if a minimal splitting tree is $(n-1)$-stable (n is the number of states of M), then it is stable.

This follows from the well-known fact that $n-1$ is an upper bound for the length of a minimal separating sequence (Moore, 1956 [F56])).

Algorithm 2 ensures a stable and minimal splitting tree. The first repeat-loop is the same as before (in Algorithm 1). Clearly, we obtain a 1-stable and minimal splitting tree here. It remains to show that we can extend this to a stable and minimal splitting tree. Algorithm 3 will perform precisely one such step towards stability, while maintaining minimality. Termination follows from the same reason as for Algorithm 1. Correctness for this algorithm is shown by the following key lemma. We will denote the input tree by T and the tree after performing Algorithm 3 by T'. Observe that T is an initial segment of T'.

Lemma Algorithm 3 ensures a $(k+1)$-stable minimal splitting tree.

Proof. Let us prove stability. Let s and t be in the same leaf of T' and let $x \in I^*$ be such that $\lambda(s,x) \neq \lambda(t,x)$. We show that $|x| > k+1$.

Suppose for the sake of contradiction that $|x| \leq k+1$. Let u be the leaf containing s and t and write $x = ax'$. We see that $\delta(s,a)$ and $\delta(t,a)$ are separated by k-stability of T. So the node $v = \mathrm{lca}(\delta(l(u),a))$ has children and an associated sequence $\sigma(v)$. There are two cases:

- $|\sigma(v)| < k$, then $a\sigma(v)$ separates s and t and is of length $\leq k$. This case contradicts the k-stability of T.
- $|\sigma(v)| = k$, then the loop in Algorithm 3 will consider this case and split. Note that this may not split s and t (it may occur that $a\sigma(v)$ splits different elements in $l(u)$). We can repeat the above argument inductively for the newly created leaf containing s and t. By finiteness of $l(u)$, the induction will stop and, in the end, s and t are split.

Both cases end in contradiction, so we conclude that $|x| > k+1$.

Let us now prove minimality. It suffices to consider only newly split states in T'. Let s and t be two states with $|\sigma(\mathrm{lca}(s,t))| = k+1$. Let $x \in I^*$ be a sequence such that $\lambda(s,x) \neq \lambda(t,x)$. We need to show that $|x| \geq k+1$. Since $x \neq \epsilon$ we can write $x = ax'$ and consider the states $s' = \delta(s,a)$ and $t' = \delta(t,a)$ which are separated by x'. Two things can happen:

- The states s' and t' are in the same leaf in T. Then by k-stability of T we get $\lambda(s',y) = \lambda(t',y)$ for all $y \in I^{\leq k}$. So $|x'| > k$.
- The states s' and t' are in different leaves in T and let $u = \mathrm{lca}(s',t')$. Then $a\sigma(u)$ separates s and t. Since s and t are in the same leaf in T we get $|a\sigma(u)| \geq k+1$ by k-stability. This means that $|\sigma(u)| \geq k$ and by minimality of T we get $|x'| \geq k$.

In both cases we have shown that $|x| \geq k+1$ as required. □

Example Figure 7(a) shows a stable and minimal splitting tree T for the machine in Figure 6. This tree is constructed by Algorithm 2 as follows. It executes the same as Algorithm 1 until we consider the node labelled $\{s_0, s_2\}$. At this point $k = 1$. We observe that the sequence of $\text{lca}(\delta(\{s_0, s_2\}, a))$ has length 2, which is too long, so we continue with the next input. We find that we can indeed split with respect to the state after b, so the associated sequence is ba. Continuing, we obtain the same partition as before, but with smaller witnesses.

The internal data structure (a refinable partition) is shown in Figure 7(b): the array with the permutation of the states is at the bottom, and every block includes an indication of the slice containing its label and a pointer to its parent (as our final algorithm needs to find the parent block, but never the child blocks).

Algorithm 2 Constructing a stable and minimal splitting tree.

```
Require: An FSM $M$ with $n$ states
Ensure: A stable, minimal splitting tree $T$
1: initialise $T$ to be a tree with a single node labelled $S$
2: repeat
3:     find $a \in I$, $B \in P(T)$ such that we can split $B$ with respect to output $\lambda(\cdot, a)$
4:     expand the $u \in T$ with $l(u) = B$ as described in (split-output)
5: until $P(T)$ is acceptable
6: for $k = 1$ to $n - 1$ do
7:     invoke Algorithm 3 on $T$ for $k$
8: end for
```

Algorithm 3 A step towards the stability of a splitting tree.

```
Require: A $k$-stable and minimal splitting tree $T$
Ensure: $T$ is a $(k + 1)$-stable, minimal splitting tree
1: for all leaves $u \in T$ and all inputs $a \in I$ do
2:     $v \leftarrow \text{lca}(\delta(l(u), a))$
3:     if $v$ is an inner node and $|\sigma(v)| = k$ then
4:         expand $u$ as described in (split-state) (this generates new leaves)
5:     end if
6: end for
```

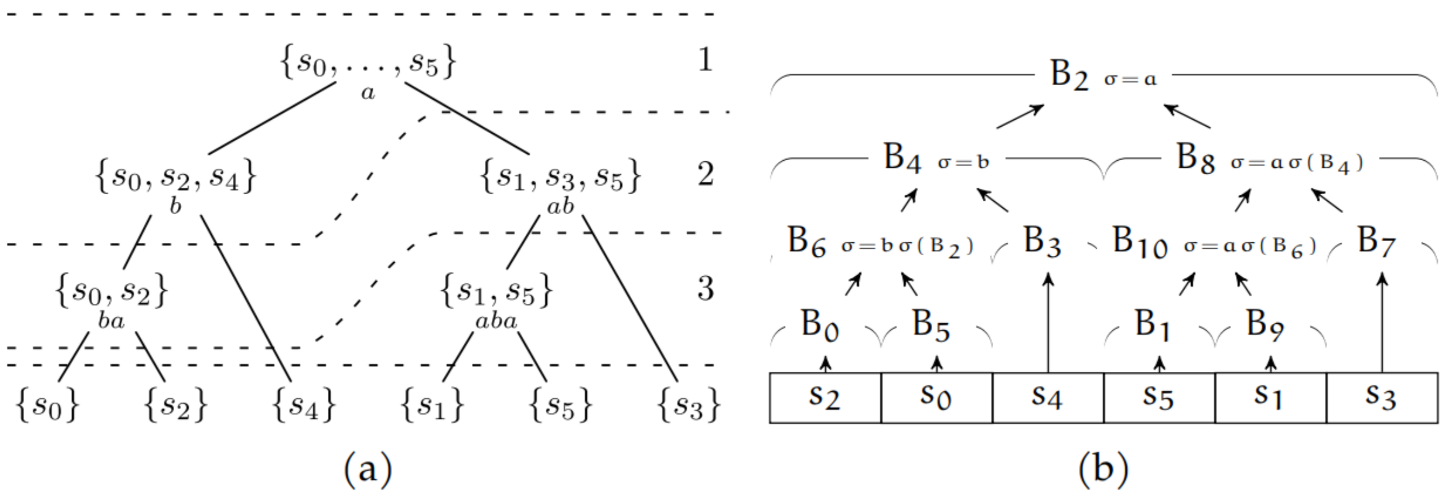


Figure 7: (a) A complete and minimal splitting tree for the FSM in Figure 6 and (b) its internal refinable partition data structure.

A splitting tree can be used to extract relevant information for a characterisation set for the W-method.

Definition A set $W \subset I^*$ is called a characterisation set if for every pair of inequivalent states s, t there is a sequence $w \in W$ such that $\lambda(s,w) \neq \lambda(t,w)$.

Lemma Let T be a complete splitting tree, then the set $\{\sigma(u) \mid u \in T\}$ is a characterisation set.

Proof. Let $W = \{\sigma(u) \mid u \in T\}$. Let $s,t \in S$ be inequivalent states, then by completeness s and t are contained in different leaves of T. Hence $u = \text{lca}(s,t)$ exists and $\sigma(u) \in W$ separates s and t. This shows that W is a characterisation set. □

Lemma A characterisation set with minimal length sequences can be constructed in time $\mathcal{O}(m \log n)$.

Proof. The sequences associated with the inner nodes of a splitting tree form a characterisation set. Such a tree can be constructed in time $\mathcal{O}(m \log n)$. Traversing the tree to obtain the characterisation set is linear in the number of nodes (and hence linear in the number of states). □

3.3 Learner aware approach

In this section, we will see how to make the most of the information from the learning algorithm. Indeed, let's suppose that the SUL (System Under Learning) is surrounded by a cache, meaning that if two identical queries are made, the second one incurs no real cost. An idea would be to make the most of this cache by leveraging the information from the learning algorithm to increase the chances of encountering queries that it has already processed.

Of course, this technique cannot work for all learning algorithms: in our case, we will base it on the L* algorithm.

To create this new set W, we will take the suffix set S from L* and minimize it. However, during the minimization, it is important to maintain the property of the characterization set. This is what Algorithm 4 does.

Algorithm 4 Constructing a W set learner aware

```
Require: A $W$ set given by the learning algorithm
Ensure: A minimal subset of $W$ which is still a characterisation set
 1: function WFROM($W$, $K = \emptyset$)
 2: Create a set $C$ for all sets that are still characterisation set
 3: Create a set $N$ for all states that are mandatory
 4: for all words $w$ in $W$ do
 5:     construct $F = W \setminus \{w\}$
 6:     if $F$ is a characterisation set then
 7:         add $F$ to $C$
 8:     else
 9:         add $w$ to $N$
10:     end if
11: end for
12: if $C = \emptyset$ then
13:     return $K \cup N$
14: end if
15: for all sets $W_i$ in $C$ do
16:     invoke WFROM($W_i$, $K \cup N$)
17: end for
18: return the $WFROM$ with the smallest length
```

The objective here is to separate the states into two categories: those that are mandatory (meaning if they are no longer present, then W is no longer a characterization set) and those that are not mandatory (meaning if they are no longer present, then W retains its property as a characterization set). The 'mandatory' states will be placed in what we will call a 'core', and we will make recursive calls with the sets deprived of the 'non-mandatory' words.

In this way, we achieve the smallest subset of the W provided by L* that is still a characterization set.

4 Results

	Size 5 DFA (k=3)	Size 10 DFA (k=5)
Minimal approach (total)	231	465
Leaner aware approach (total)	148	469
Minimal approach (equivalence)	216	437
Leaner aware approach (equivalence)	138	446

Table 1: Comparison of different approaches

Table 1 shows the number of queries obtained during the learning of DFA of size 5 and size 10. We have separated the total number of queries as well as the number of queries at the time of the final equivalence.

As you can notice, this table does not show clear results. Indeed, another point that we have not addressed and that would be important to implement is the prioritization order of the tests. An article done by Kruger L. (January 2024) [SJ24] demonstrates this specifically by using two types of prioritization: the first by random selection and the other by what it calls 'experts'.

5 Bibliography

References

[A62] Gill A. "Introduction to the theory of finite-state machines". In: *McGraw-Hill* (1962).

[AP08] Valmari A. and Lehtinen P. "Efficient Minimization of DFAs with Partial Transition Functions". In: *Symposium on Theoretical Aspects of Computer Science, STACS* (2008).

[D73] Gries D. "Describing an Algorithm by Hopcroft". In: *Acta Inf* (1973).

[D87] Augluin D. "Learning Regular Sets from Queries and Counterexamples". In: *Inf. Comput.* (1987).

[DM94] Lee D. and Yannakakis M. "Testing Finite-State Machines: State Identification and Verification". In: *IEEE Trans. Computers* (1994).

[E71] Hopcroft J. E. "An n log n algorithm for minimizing states in a finite automaton". In: *Theory of Machines and Computations - Proceedings of an International Symposium on the Theory of Machines and Computations* (1971).

[ER14] Chalupar G. Peherstorfer S. Poll E. and J. de Ruiter. "Automated Reverse Engineering using Lego". In: *8th USENIX Workshop on Offensive Technologies, WOOT* (2014).

[F56] Moore E. F. “Gedanken–experiments on Sequential Machines”. In: *Sequential Machines, Automata Studies, Annals of Mathematical Studies, no.34* (1956).

[FD13] Bonchi F. and Pous D. “Checking NFA equivalence with bisimulations up to congruence”. In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2013).

[FH17] P. Fiterău-Broştean and F. Howar. “Learning-Based Testing the Sliding Window Behavior of TCP Implementations”. In: *Critical Systems: Formal Methods and Automated Verification, Joint FMICS-AVoCS, Proceedings* (2017).

[JW16] Schuts M. Hooman J. and Vaandrager F. W. “Refactoring of Legacy Software Using Model Learning and Equivalence Checking: An Industrial Experience Report”. In: *Integrated Formal Methods - 12th International Conference, IFM, Proceedings* (2016).

[KR17] Tappler M. Aichernig B. K. and Bloem R. “Model-Based Testing IoT Communication via Active Automata Learning”. In: *ICST, Proceedings* (2017).

[Moe19] Joshua Moerman. “Nominal Techniques and Black Box Testing for Automata Learning”. In: *PhD thesis* (2019).

[OB03] Hungar H. Niese O. and Steffen B. “Domain-Specific Optimization in Automata Learning”. In: *n Computer Aided Verification, 15th International Conference, CAV, Proceedings* (2003).

[P73] Vasilevskii M. P. “Failure diagnosis of automata”. In: *Cybernetics and Systems Analysis* (1973).

[RE13] Aarts F. de Ruiter J. and Poll E. “Formal Models of Bank Cards for Free”. In: *ICST, Workshops Proceedings* (2013).

[RE15] de Ruiter J. and Poll E. “Protocol State Fuzzing of TLS Implementations”. In: *24th USENIX Security Symposium, USENIX Security* (2015).

[RN10] Dorofeeva R. El-Fakih K. Maag S. Cavalli A. R. and Yevtushenko N. “FSM-based conformance testing methods: A survey annotated with experimental evaluation”. In: *Information and Software Technology* (2010).

[RP17] Fiterău-Broştean P. Lenaerts T. Poll E. de Ruiter J. Vaandrager F. W. and Verleg P. “Model learning and model checking of SSH implementations”. In: *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium* (2017).

[RW16] Fiterău-Broştean P. Janssen R. and Vaandrager F. W. “Combining Model Learning and Model Checking to Analyze TCP Implementations”. In: *Computer Aided Verification - 28th International Conference, CAV, Proceedings, Part II* (2016).

[S78] Chow T. S. “Testing Software Design Modeled by Finite-State Machines”. In: *IEEE Trans. Software Eng* (1978).

[SJ24] Kruger L. Junges S. and Rot J. “Small Test Suites for Active Automata Learning”. In: *Institute for Computing and Information Sciences* (2024).

[YM02] Peled D. A. Vardi M. Y. and Yannakakis M. “Black Box Checking”. In: *. Journal of Automata, Languages and Combinatorics* (2002).