Feedback Rich.

Relevant Literature

Wesley van der Lee

October 12, 2017

1 Model Inference

Draft 3, ext rev SS.

1.1 Terminology

This chapter discusses notorious algorithms for inferring state machines. In order to keep consistency between all discussed related work, this section establishes a formal mathematical notation which will be used in the continuance of this document. The notation will be consistent with the notation proposed by Sipser [1].

-> remainder.

The deterministic finite automaton (DFA) U is used to formalize state machines. An example DFA $\mathcal{A}$ is depicted in Figure 1. As can be seen, a DFA contains states, letters, transitions, a start state and accepting states. A DFA can thus be formalized by a 5-tuple $U = (Q, \Sigma, \epsilon, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \to Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

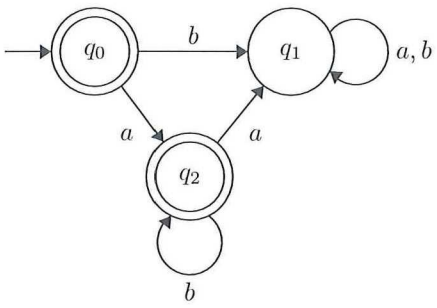


Figure 1: Example DFA $\mathcal{A}$.

The example DFA $\mathcal{A}$ shows three states: q_0, q_1 and q_2 depicted by the circles. Double edged circles indicate that the state is an accepting state, which in the example of $\mathcal{A}$ is the case for q_0 and q_2. Transitions from one state to another are indicated by arrows and their corresponding input letter. From this follows that the input alphabet of $\mathcal{A}$ solely consists of the elements a and b. In general,

a state $q' \in Q$ which is the result of a transition from another state $q \in Q$ with input letter $a \in \Sigma$, is also called the a-successor of q, i.e. $q' = \delta(q, a)$ is the a-successor of q. In the example of DFA $\mathcal{A}$, state q_1 is the b-successor of state q_0. The successor-notation can also be extended for words by defining $\delta(q, \varepsilon) = q$ and $\delta(q, wa) = \delta(\delta(q, w), a)$ for $q \in Q$, $a \in \Sigma$ and $w \in \Sigma^*$. For words $w \in \Sigma^*$ the transition function $\delta(q, w)$ implies the extended transition function: $\delta(q, w) = \delta(q_0, w)$ unless specifically specified. Moreover, by also utilizing the notation of Vazirani et al. [2], a state of a general DFA U can be denoted as $U[w]$ for $w \in \Sigma^*$, where $U[w]$ corresponds to the state in U reached by w, i.e. $U[w] = \delta(q_0, w)$. The singleton transition without an associating input letter, indicates the empty transition, which points to the initial starting state of $\mathcal{A}$.

Furthermore, for words $w \in \Sigma^*$ the state $U[w]$ of a DFA U either results in an accepting state or a rejecting state. Accepting states are modeled with a double edged node, whereas rejecting states are modeled with a single edged node. For states $q \in Q$ that are accepting states, it also holds that $q \in F$. The accepting states of $\mathcal{A}$ are q_0 and q_2. The set of all words $w \subseteq \Sigma^*$ that DFA U accepts, is called the language of U indicated by $L(U)$. A language can be infinite as Σ^* is unbounded. This is the case for DFA $\mathcal{A}$ as it accepts an input existing of any number of b's after a single a, i.e. $\{\varepsilon, a, ab, abb, abbb, \cdots, ab*\}$. The λ-function also evaluates if an input sequence is accepted by a DFA. For a word $w \in \Sigma^*$ the λ-evaluation $\lambda(w)$ returns 1 iff the DFA in question accepts word w, that is if the extended transition function for q_0 concludes in an accepting state. The function $\lambda(w)$ returns 0 if the extended transition function results in a rejecting state.

Example 1.1. DFA $\mathcal{A}$ from Figure 1, can be formally written as the 5-tuple $\mathcal{A} = \{Q, \Sigma, \delta, q_0, F\}$ where

$Q = \{q_0, q_1, q_2\}$,
$\Sigma = \{a, b\}$,
δ is described as:

	a	b
q_0	q_2	q_1
q_1	q_1	q_1
q_2	q_1	q_2

q_0 is the starting state, and
$F = \{q_0, q_2\}$.

Σ^* is the set of words over symbols in Σ, including the empty word ε. The *-notation follows from the unary operation, which works by attaching any number of strings in Σ together: $\Sigma^* = \{x_1, x_2 \ldots, x_{k-1}, x_k | k \geq 0$ and each $x_i \in \Sigma\}$. For all input sequences In $\mathcal{A}$ one can see that word $w_1 = abbb \in \Sigma^*$, because w_1 leads to an accepting state[1]. A word $w_2 = abab$ can be identified as not a member of Σ^*: $w_2 \notin L(\mathcal{A})$[2]. For words $w, w' \in \Sigma^*$, $w \cdot w'$ is a concatenation-operation of the two words. The concatenation-operation of two words can also be written by omitting the operator $\cdot$ and just write ww'.

[1] $\delta(q_0, abbb) = \delta(\delta(q_0, a), bbb) = \delta(\delta(\delta(q_0, a), b), bb) = \delta(\delta(\delta(\delta(q_0, a), b), b), b) = \delta(\delta(\delta(q_1, b), b), b) = \delta(\delta(q_1, b), b) = \delta(q_1, b) = q_2 \in F \rightarrow abbb \in L(\mathcal{A})$.

[2] $\delta(q_0, abab) = \delta(\delta(q_0, a), bab) = \delta(\delta(\delta(q_0, a), b), ab) = \delta(\delta(\delta(\delta(q_0, a), b), a), b) = \delta(\delta(\delta(q_2, b), a), b) = \delta(\delta(q_2, a), b) = \delta(q_1, b) = q_1 \notin F \rightarrow abab \notin L(\mathcal{A})$.

1.2 Learning Regular Sets from Queries and Counterexamples

Learning Regular Sets from Queries and Counterexamples by Dana Angluin [3] forms the basis of many modern state machine inference algorithms. Her research introduces the polynomial L* algorithm for learning a regular set, a task which before was computationally intractable because it was proven to be NP-hard [4]. A regular set represents the value of expressions that describe languages, or in other words regular expressions. Expressions are regular if they are created with regular operators, such as union and intersection[1]. The reason to infer a regular is that if a set is regular, it can be modeled by a DFA.

Example 1.2. The regular expression to express DFA $\mathcal{A}$ is $(\varepsilon \cup ab^*)$, since it accepts the empty string and a single a followed by any number of b's.

The basic idea of the L* algorithm is a learner whose goal is to create a equitable conjecture by utilization of an expert system, the Minimally Adequate Teacher (MAT). A conjecture is a hypothesized DFA that approximates the SUT's behavior and is either equivalent or not. The establishment of a conjecture is achieved by posing two types of queries to the MAT:

- **membership queries** that answer *yes* or *no* for an input word w depending on whether w is a member of the to be hypothesized DFA. This is equivalent to the *lambda*-evaluation for a word w that depicts whether w is recognized by the set U: $\lambda(w) \rightarrow \{0, 1\}$.
- **equivalence queries** that take as input a conjecture DFA and then answers *yes* if the conjecture is equal to the set U. If this is not the case, the MAT provides a counterexample, which is a string w' in the symmetric difference of the conjecture and the unknown language.

The learner keeps track of the queried strings, classified by either a member or non-member of the unknown regular set U. This information is organized in an observation table that consists of three fields: a nonempty finite prefix-closed set S of strings, a nonempty finite suffix-closed set E of strings and a finite function T that maps all entries that are formed by concatenating the prefix and suffix together, see Figure 2. A word u recognized by the set U can be typically of the form $u = sae$, for $s \in S$, $e \in E$ and $a \in \Sigma$, meaning a word starts with a prefix and ends with a suffix. Sometimes a one-letter extension in Σ is added to the prefix, they identify transitions. The latter will become apparent in the running example of the L* algorithm. Angluin applies the notation of words in the form $u = sae$ as $u \in ((S \cup S \cdot \Sigma) \cdot E)$, thus if $u \in U$ then $T(u)$ will return 1 and 0 otherwise. Function T thus corresponds to the earlier mentioned λ-function: $T(w) \mathrel{\widehat{=}} \lambda(w)$. To maintain consistency with Angluin's work, the function T notation will be used for the continuance of this section. In conclusion an observation table can be denoted as a 3-tuple (S, E, T).

To clarify the terminology, imagine the example where a learner is required to learn the behavior of DFA $\mathcal{A}$ described in Figure 1. Initially any information except $\mathcal{A}$'s alphabet Σ is unknown to the learner, but the learner has access to a MAT that can answer membership and equivalence queries. The learner starts by first creating the observation table. Set S initially contains the input alphabet of $\mathcal{A}$ and the empty string ε as one-letter prefixes. Thus $S = \{\varepsilon\} \cup \Sigma = \{\varepsilon, a, b\}$

	ε
ε	1
b	0
a	1

(a)

	ε
ε	1
b	0
a	1
ba	0
bb	0

(b)

	ε	b
ε	1	0
b	0	0
a	1	1
ba	0	0
bb	0	0
aa	0	0
ab	1	1

(c)

Figure 2: Gradually growing observation tables corresponding to various steps of the L* algorithm: (a) initial observation table T_0, (b) observation table T_1 that is a closed and consistent version of the initial observation table, (c) final observation table T_2 describing DFA $\mathcal{A}$.

The commencing distinguishing suffix is ε, therefore set $E = \{\varepsilon\}$. The learner then fills the entries in observation table $S \times E$ with the output of membership queries, depending on whether an entry $e \in S \times E$ is accepted by $\mathcal{A}$ or not. This leads to the initial observation table depicted in Figure 2a. Note that the table vertically distinguishes two sections that are separated by an additional line. That is, set S is divided into two subsets. The top section of S represents all distinct rows with respect to the outcome of T. Since $row(\varepsilon) = 1$ and $row(b) = 0$, those distinct rows are put in the top part of S. $Row(a)$ results to 1, which is equivalent to $row(\varepsilon)$, hence $row(a)$ is put in the bottom part of S.

The learner queries for the right amount of data from the MAT, by ensuring that the observation table is both closed and consistent. An observation table is *closed* if for every entry $e \in S \times \Sigma$, there exists an element $s \in S$ such that $row(e) = row(s)$. If the table is not closed, it lacks information for transitions. Table 2a is not closed, because this property is not ensured for the state $\mathcal{A}[b]$. The table clearly depicts at least two states, because there are two distinct rows. However $\mathcal{A}[b]$ requires information on where to transit from that state. Hence the access sequence of the state $\mathcal{A}[b]$ appended with one-letter extensions are added to the set of prefixes. In other words $\{ba, bb\}$ are to be added to the set S. This results in a bigger observation table, depicted in Table 2b. The learner identifies which suffix distinguishes the rows and adds the word to the set S. An observation table is *consistent* when for all $s_1, s_2 \in S$ where $row(s_1) = row(s_2)$, if for all a in the alphabet Σ holds that $row(s_1 \cdot a) = row(s_2 \cdot a)$. If the table is not consistent, the language would not non-deterministic and can thus not be modeled by a deterministic finite automaton. The learner identifies for which $s_1, s_2 \in S$ distinguishes the result of output T and adds the word to the set E. Given the observation table in 2b, the observation table is consistent.

If (S, E, T) is closed and consistent, an acceptor $H(S, E, T)$ can be defined which is consistent with function T. H then forms the hypothesized model based on the contents of the observation table. A conjecture $H = (\Sigma, Q, \delta, q_0, F)$ can be modeled as follows:

1. $Q = \{row(s) | s \in S_H\}$
2. $q_0 = row(\varepsilon)$
3. $F = \{row(s) | s \in S_H \text{ and } T_H(s, \varepsilon) = 1\}$

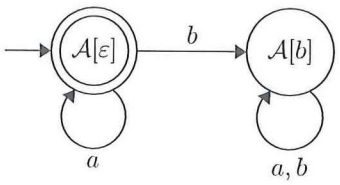


Figure 3: Hypothesized conjecture DFA H_0 corresponding to observation table T_1 (Table 2b).

4. $\delta(row(s), a) = row(sa)$

Example 1.3. A hypothesized DFA H_0 generated according to the steps above consistent to the observation table 2b that has the following contents: $Q = \{\mathcal{A}[\varepsilon], \mathcal{A}[b]\}$, $q_0 = \mathcal{A}[\varepsilon]$, $F = \mathcal{A}[\varepsilon]$. Conjecture H_0 is visualized in Figure 3. The learner then verifies the conjecture to the MAT and receives a counterexample back, indicating that the conjecture is inequivalent to U. Although the learner was unable to see this, the model depicted in Figure 3 clearly differs from the SUT shown in Figure 1. Suppose that the learner received the counterexample word $w = ab$, which is a valid counterexample because $\lambda_{\mathcal{A}}(ab) = 1$, $\lambda_{H_0}(ab) = 0$, thus $\lambda_{\mathcal{A}}(ab) \neq \lambda_{H_0}(ab)$ holds.

Analysis of the counterexample $w = ab$ shows that if suffix b is added, H_0 predicts the wrong output. Element b is therefore identified as a distinguishing suffix and hence added to set E. The observation table must now be updated again: the new entries caused by the appearance of the b-column are filled and in order to make the table closed again, new prefixes for the new state $\mathcal{A}[a]$ have to be determined. This results in the observation table depicted in Table 2c. Following the steps to establish a DFA from an observation table yields the original DFA shown in Figure 1 which is the correct DFA that corresponds to the observation table T_2 shown in Table 2c. After the learner poses an equivalence query with this DFA, the MAT answers *yes*, indicating that the correct DFA has been inferred and the learner can stop learning.

1.3 Minimal DFAs

Up until now, the only focus was for the conjecture to be consistent with T. Since a DFA with the state size equal to the number of observations, that is each observation leads to a new case state in a conjecture, such a DFA would be incorrect because it possibly incorrectly models unforeseen future observations. An hypothesized conjecture H should thus not only be consistent with T, but also have the smallest set of states to model the SUT's behaviour. The L* algorithm only learns a DFA consistent with T and that has the smallest set of states. Angluin proves this as follows. Let q_0 be the starting state ($row(\varepsilon)$) and δ be the transition function from one state to another in the acceptor, then for $s \in S$ and $a \in \Sigma$ holds that because $\delta(row(s), a) = row(s \cdot a)$ follows $\forall s \in (S \cup S \cdot \Sigma) : \delta(q_0, s) = row(s)$, thus the closed property ensures that any row in the observation table corresponds with a valid path in the acceptor. $\forall s \in (S \cup S \cdot \Sigma) \forall e \in E \rightarrow \delta(q_0, s \cdot e)$ is an accepting state if and only if $T(s \cdot e) = 1$, thus due to the consistency with finite function T, a word will be accepted by

the acceptor if it is in the regular set. To see that $H(S, E, T)$ is the acceptor with the least states, one must note that any other acceptor H' consistent with T is either isomorphic or equivalent to $H(S, E, T)$ or contains more states.

The algorithm L* is listed in Listing 1.

```
1  S = E = {λ}
2  (S,E,T) ← MQ(λ) ∪ ∀a ∈ A : MQ(a) # (S,E,T) is the observation table
3
4  While M is incorrect: # M is the conjecture
5    While (S,E,T) is not consistent or not closed
6      if (S,E,T) is not consistent:
7          ∃(s1,s2) ∈ S, a ∈ A, e ∈ E : row(s1) = row(s2) and T(s1 · a · e) ≠ T(s2 · a · e)
8          E ← a · e
9          extend T to (S ∪ S · A) · E using MQ
10     if (S,E,T) is not closed:
11         ∃s1 ∈ S, a ∈ A : ∀ row(s1 · a) ≠ row(s)
12         S ← s1 · a
13         extend T to (S ∪ S · A) · E using MQ
14   M = M(S,E,T)  # (S,E,T) is closed and consistent
15   if Teacher replies with a counter-example t:
16     add t and all its prefixes to S
17     extend T to (S ∪ S · A) · E using MQ
18 Halt and output M
```

Listing 1: The L* Algorithm

1.4 Counterexample Decomposition

In the previous section, the learner was tasked to infer the behavior of set $\mathcal{A}$ illustrated in Figure 1. At some point the learner inferred an incorrect conjecture H_0, depicted in Figure 3. The key step to improve H_0 towards $\mathcal{A}$ was to utilize a given counterexample word $w = ab$. This section discusses how the counterexample can be decomposed by the learner and elaborates on the process of how this decomposition leads towards the appearance of a new state.

Suppose at some point the learner poses an equivalence query to the teacher for an incorrect conjecture H and recieves a counterexample word w. Word w exists of a prefix and a suffix part. Let the suffix be av, then since w is a counterexample it implies that for any two access sequences $u, u' \in U$ reach the same state in H, we have: $\lambda(ua \cdot v) \neq \lambda(u' \cdot v)$. This becomes apparent in Figure 4a where input word uav leads to a different state than word $u'v$. This is noticeable since both states yield a different output. One could digest this even further by concluding that the state with access sequence u trailed with letter a is a different state in comparison to the state with access sequence ua. In other words $\lambda^A(\lfloor u \rfloor_H a \cdot v) \neq \lambda^A(\lfloor ua \rfloor_H \cdot v)$.

Since a word $w = \langle prefix, suffix \rangle$ and the suffix is modeled as av, a counterexample word w can also be decomposed in a three-tuple $w = \langle u, a, v \rangle$. Element v is called a distinguishing suffix, because it distinguishes the states $U[ua]$ and $U[u']$ from each other. Angluin's L* algorithm adds v to the set E, such that in the observation table the rows ua and u' differ from each other. This results in one more distinct row in the observation table, hence that state q is split into two states: one state q_1 that leads uav to q' and one state q_2 that leads $u'v$ to q''. The latter is depicted in Figure 4b.

Example 1.4. In the running example of the former section the counterexample word $w = ab$ led to the appearance of a new state in the conjecture.

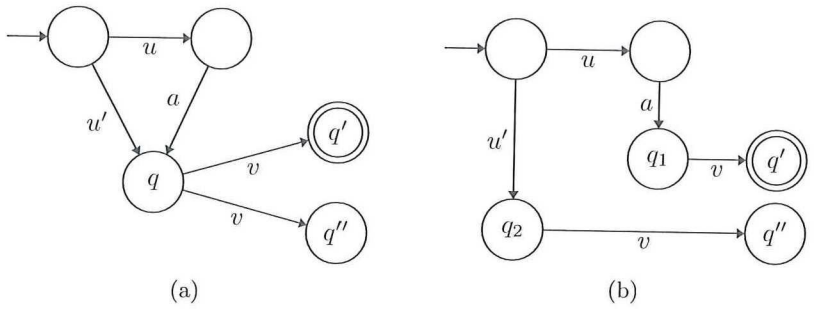


Figure 4: Formal progression of an incorrect conjecture: (a) inconsistent model for distinguishing suffix v from state q, (b) consistent model after splitting q into new states q_1 and q_2.

Word w is a valid counterexample because $\lambda_H(ab) \neq \lambda_{\mathcal{A}}(ab)$. According to the decomposition from above, this must be true because $\lambda_H(ua \cdot v) \neq \lambda_H(u' \cdot v)$ with the counterexample decomposition $w = \langle u, a, v \rangle = \langle \varepsilon, a, b \rangle$. In Figure 3 $H[ua \cdot v] = H[\varepsilon \cdot a \cdot b] = H[ab]$ is equal to the state $H[u'v] = H[\varepsilon \cdot b] = H[b]$, whilst $\lambda_A(ab) \neq \lambda_{\mathcal{A}}(b)$. The state $H[\varepsilon \cdot a] = H[a]$ should thus be added. This is achieved by splitting $H[\varepsilon]$, because both $H[\varepsilon]$ and $H[a]$ have the same output for λ. This results into two new states: $H[\varepsilon]$ and $H[a]$.

1.5 The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning

In *The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning* [5] Isberner et al. recognizes that one of the computational expensive disadvantages of Angluin's L* algorithm is a consequence of redundant entries in the observation table. The redundant entries are a result of suboptimal and possibly excessively long counterexamples provided by the MAT to the learner's equivalence queries. The prefix-part and the suffix-part of the counterexample contain elements that do not contribute to the discovery of new states even the distinguishing suffix possibly contain redundant elements. The L* algorithm can only establish a conjecture when the observation table is closed and consistent. As a result the learner also poses membership queries to fill out each new entry field that is created by the product of the counterexample's redundant prefix and suffix elements. To illustrate the presence of the redundant fields: in the observation table of the employed example DFA shown in Table 2c, the value 0 in the ε-column suffices to distinguish the $\mathcal{A}[b]$-state from the other two states. The remaining fields for these particular rows denoted by the b-suffix do not contribute to the distinctiveness of these rows, hence the suffix to distinguish the state $\mathcal{A}[b]$ from $\mathcal{A}[\varepsilon]$ and $\mathcal{A}[a]$ are redundant.

In order to overcome performance impacts caused by redundancies in counterexamples, Isberner et al. proposes the TTT algorithm. The TTT algorithm does so by utilization of a redundancy-free organization of observations in a discrimination tree. A discrimination tree adopts two sets $S, E \subset \Sigma^*$ that are non-empty and finite like Angluin's L* algorithm does, the only difference being that S consists of state access strings opposed to prefixes. Set E still contains

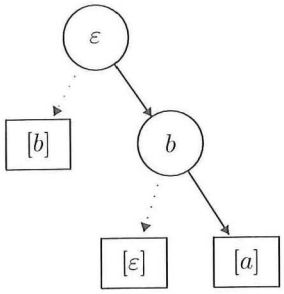


Figure 5: Discrimination Tree DT corresponding to DFA $\mathcal{A}$.

distinguishing suffixes, which are referred to by Isberner as discriminators. The tree can then be modeled by a rooted binary tree where leafs are labeled with access strings in S and inner nodes are labeled with suffixes in E. The two children of an inner node correspond to labels $\ell \in \{\top, \bot\}$ that represent the λ-evaluation. Other studies apply different terminology, like $\ell \in \{+, -\}$, $\ell \in \{1, 0\}$ or adopt consistency in the direction their children have: a child to the left coincides with $\ell = \top$ and a child to the right coincides with $\ell = \bot$ [5, 6].

The discrimination tree is shaped in a way such that words can be *sifted* through the tree. Sifting is required in order to determine the transitions when establishing the transitions of a conjecture. The process of sifting is administered to a word $w \in \Sigma^*$ that starts at the root of the tree. For every inner node $v \in E$ of the discrimination tree the sifting process passes one branches to the $\top$- or $\bot$- child depending on the value $\lambda(w \cdot v)$ until a leaf node is reached. The leaf node then indicates the resulting state with the corresponding access sequence for word w. The relation of the leaf node to its direct parent, either $\top$ or $\bot$, depict whether w is accepted or not. Each pair of states have then exactly one distinguishing suffix, which is the lowest common ancestor of the two leafs.

The discrimination tree DT corresponding to $\mathcal{A}$ is depicted in Figure 5. How the TTT algorithm gradually builds this tree will be discussed in the end of this section. Note that the state names q_0, q_1 and q_2 are replaced by their access sequence notation: $[\varepsilon], [b]$ and $[a]$ respectively.

Example 1.5. Sifting is used to establish the transitions of a conjecture from a discrimination tree. If for example the b-transition from state $\mathcal{A}[a]$ should be determined i.e. $\delta(\mathcal{A}[a], b) = ?$, one needs the access sequence for state $\mathcal{A}[a]$ which is a, and the one-letter extension which in this case is b. Together they form the word $w = ab$ and this is sifted through DT. Starting at the root of the tree, one must evaluate $\lambda(ab \cdot \varepsilon)$ which results to 1. Thus one follows the $\top$-branch and finds the inner-node b. Then $\lambda(ab \cdot b)$ is evaluated, which also results to 1. Again following the $\top$-branch of the tree, results in the leaf $[a]$, at which point the process shows that $\delta(\mathcal{A}[a], b) = \mathcal{A}[a]$.

The TTT algorithm follows the following steps in order to infer a correct model:

1. Hypothesis Construction
2. Hypothesis Refinement

3. Hypothesis Stabilization
4. Discriminator Finalization

1. Hypothesis Construction
The initial discrimination tree is constructed by evaluating $\lambda(\varepsilon)$. This results in either a tree with as root node ε and a single leaf with access string $[\varepsilon]$ on either the $\top$- or $\bot$-child of the root node depending on the λ-evaluation. A conjecture DFA is established where:

1. Q are all the leaf nodes in the discrimination tree,
2. Σ is already known,
3. δ is determined by sifting all words $w \in Q \times \Sigma$,
4. q_0 is $[\varepsilon]$ and
5. F are all leaf nodes that are a $\top$-child in the tree.

Example 1.6. Following the example where DFA $\mathcal{A}$ from Figure 1 should be learned again, but now according to the TTT algorithm, the algorithm starts with evaluating $\lambda(q_0, \varepsilon)$. This results to 1, thus an initial discrimination tree is established where the $\top$-child points to the initial state, since it is an accepting state. The initial discrimination tree DT_0 is depicted in Figure 6a. In order to create an initial conjecture DFA, one takes all leafs from DT_0, in this case only $H[\varepsilon]$ and determines transitions by sifting the access sequence with all one-letter extensions. Thus εa and εb are sifted in order to respectively determine the a and b transitions from the initial state $H[\varepsilon]$. Furthermore, a state $q \in Q^H$ is in F^H if the associated leaf is on the $\top$-side of the ε-node. The initial state is the only state in the hypothesis, hence $S = \{\varepsilon\}$, and since q_0 is an accepting state, the q_0-leaf is the $\top$-child of the ε-root. The initial conjecture DFA H_0 corresponding to DT_0 is depicted in Figure 6b.

2. Hypothesis Refinement
The initial hypothesis is likely to be inequivalent to the SUT, in which case a counterexample will be given by the teacher. As discussed in section 1.4, a counterexample can be decomposed as $\langle u, a, v\rangle$. The a-part is called a breaking point, because the conjecture predicts ambiguous results for the v-part after a. This breaking point is added as a node in the tree, depending on the evaluation of $\lambda(ua)$ on the $\top$- or $\bot$ branch.

Example 1.7. Suppose that conjecture H_0 was submitted as an equivalence query to the teacher and the learner recieves a counterexample $w = b$. Word w is a valid counterexample since $\lambda^{\mathcal{A}}(b) = 0 \neq \lambda^{H}(b) = 1$. As discussed in section 1.4, a counterexample can be decomposed as $\langle u, a, v\rangle$. Word w can thus also be read as $w = \varepsilon b \varepsilon$, thus the complete decomposition of the counterexample is: $u = \varepsilon, a = b$ and $v = \varepsilon$. Since state $H\lfloor ua \rfloor = H\lfloor \varepsilon b \rfloor = H\lfloor b \rfloor = q_0^{H_0}$ should be split to a new state $\lfloor u \rfloor_H a \to \lfloor \varepsilon \rfloor_H b$, a new state q_1 is introduced that can be reached by the transition $\delta(q_0, b)$. The outbound transitions from the new state are determined by sifting the access sequence of the new state with Σ. These adjustments are depicted in DT_1 and H_1 (Figure 6c and 6d).

3. Hypothesis Stabilization
Although the previous step of hypothesis refinement constructed a discrimination tree and a DFA conjecture that are consistent with each other, this does not necessarily need to be the case. There is a possibility that for a word that can

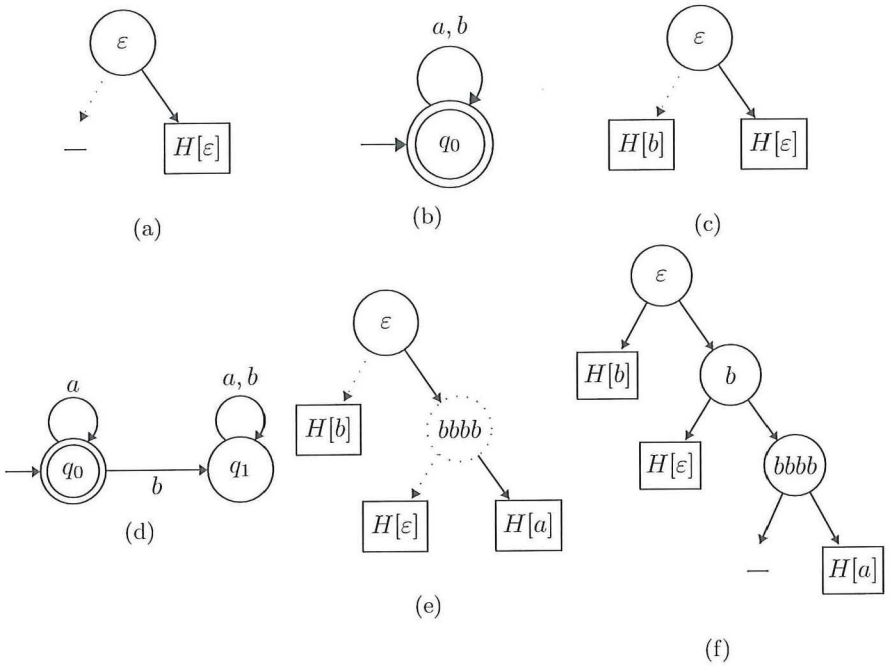


Figure 6: Evolution of discrimination trees and conjectures towards learning the DFA $\mathcal{A}$ with the TTT algorithm: (a): initial discrimination tree DT_0, (b) conjecture DFA H_0 corresponding to DT_0, (c) discrimination tree DT_1 after processing counterexample $w = b$, (d) the conjecture DFA H_1 corresponding to DT_1, (e) discrimination tree a temporary node, (f) discrimination tree where th temporary node is pushed down.

be formed in $w \in S \times E$, a $\top$-child predicts the output 1 but the hypothesized conjecture 0 or a $\bot$-child predicts the output 0, but the hypothesized conjecture 1. Word w then forms a new counterexample, and is dealt with likewise the former counterexample. This step is done until the hypothesis is stable and the discrimination tree and the conjecture are consistent.

4. Hypothesis Finalization

Often the counterexample retrieved is not minimal, such that the counterexample contains redundant information. The discriminator is added as a new node in the discrimination tree, but the algorithm adds non-elementary discriminators as a temporary node. The subtree generated with the temporary node as root must be split by subsequential replacement of the temporary discriminator closest to the subtree's root by a final discriminator. New final discriminators v' are obtained by prepending a symbol $a \in \Sigma$ to an existing final discriminator or in other words adding an additional parent node in the discrimination tree above the temporary node.

Example 1.8. Suppose H_1 was subjected to an equivalence query, which resulted in the counterexample *abbbb*. Because $\lambda^A(\lfloor u \rfloor_H a \cdot v) \neq \lambda^A(\lfloor ua \rfloor_H \cdot v)$ only holds for $a = a$, the corresponding $\langle u, a, v \rangle$-decomposition of this counterexample thus is $\langle \varepsilon, a, bbbb \rangle$. Splitting $H\lfloor a \rfloor = q_0^{H_1}$ into a new state $\lfloor \varepsilon \rfloor_H a$ a new state q_2 is introduced that can be reached by the transition $\delta(q_0, a)$. The discriminator *bbbb* is added as a temporary node, indicated by the dashed surroundings depicted in Figure 6e. The algorithm finds b to be a final discriminator, as for all elements in Σ this yields the same behavior, hence b is added as a node above the temporary discriminator, shown in Figure 6f. Finally, since the temporary discriminator does not provide distinguishable behavior compared to the new final discriminator, the temporary discriminator is removed from the discrimination tree, which results in *DT* Figure 5. Following the steps as before to construct a DFA from the discrimination tree results in the example DFA $\mathcal{A}$.

1.6 Equivalence Testing

A key step of model learning algorithms are equivalence queries that test the equality between a hypothesized model and the actual SUT. In particular equivalence queries either depict a positive result indicating that both models are equal or provide a symmetric difference of the hypothesis and the unknown model of the SUT. This conclusion is drawn after running a series of exhaustive or trivial test cases. This section discusses the two most popular DFA conformance testing methods: the *RandomWalk* and the *W-method*. In addition, since the W-method is an improvement over RandomWalk but has a gradual performance, a recently developed method for finding separating sequences for all pairs of states will also be discussed as a conformance testing method. Let the SUT's behavior be modeled by the DFA $\mathcal{A} = \{Q, \Sigma, \delta, q_0, F\}$ and let H be the hypothesized model of $\mathcal{A}$.

RandomWalk

To test whether two DFAs $\mathcal{A}$ and H are equivalent, one could perform a series of random 'walks' over H and compare if the SUT yields the same output. A walk is an arbitrary input sequence that either concludes in an accepting state or an a rejecting state. If for enough sequences both $\mathcal{A}$ and H return the same output, the two DFAs are equivalent. If one test case fails, then the sequence functions

as a counter example and refinement of H starts until the RandomWalk oracle is used again.

W-method

The W-method was first proposed by Chow in *Testing Software Design Modeled by Finite-State Machines* [7] as a method of testing the correctness of control structures that can be modeled by a finite state machine. The method embodies a test suite development strategy based on a *state cover set* of inputs P and a characterization set W of input sequences that can distinguish between the behaviors of every pair of states. A state cover set is any set of input sequences for which hold that if $q, q' \in Q | \delta(q, x, q') \in \delta$ and $x \in \Sigma$, there are input sequences p and $p \cdot x$ such that p forces the machine into state q from its initial state q_0. In addition, the state cover set also includes the empty word ε. Set P is thus composed of all short prefixes for state identification in case all observations are stored in an observation table or access sequences of all states any other case.

Chow constructs the state cover set P with the aid of a *testing tree* of the DFA H. A testing tree of H depicts H's control flow in a linear and non-cyclic manner. The tree is constructed y induction as follows:

1. The root of the testing tree is the initial state of H.
2. Suppose the tree is built to a level k. Level $(k+1)$ is built by examining all nodes on the k'th level from left to right. A node is terminated, meaning that it's branch will halt at that node, if the node appeared at a lower level j for $j \leq k$. The labels correspond to the transition symbol between the states.

The characterization set W consists of input sequences that distinguishes the behavior between every pair of states in a minimal automaton. In other words, for every two distinct states $q, q' \in Q$, W contains at least one input sequence that produces different outputs when applied from q and q' respectively. Gill [8] describes definitions and methods for constructing such sets.

Example 1.9. A testing tree for conjecture DFA H_1 (Fig. 6d) is constructed as follows:

1. The root of the tree is the initial state of H_1: q_0.
2. The next branch is created by determining the resulting state for each input symbol: a, b. Since on input a from q_0 one goes to q_0, e.g. $\delta(q_0, a) = q_0$, one node labeled q_0 is connected to the root node. The b-successor of q_0 is q_1, so a new node labeled q_1 is added and connected to the root node. Because this label differs from the layer above, this branch continues to grow. Because both the a- and b-successor of q_1 result in the q_1 state, two children are added, both labeled with q_1.

The testing tree corresponding to conjecture H_1 from Figure 6d is depicted in Figure 7. From this figure one can derive that there are 3 branches, as there are 3 leafs, all that can be reached with inputs a, ba, bb. The state cover set can thus be defined as $P = \{\varepsilon, a, ba, bb\}$.

Example 1.10. In the example of conjecture H_1, finding a characterization set is trivial, as the only pair of states is the pair q_0 and q_1 and their output differes for input symbol a as $\delta(q_0, a) = q_0 \in F$ and $\delta(q_1, a) = q_1 \notin F$, thus the output for a differs in both states and $W = \{a\}$.

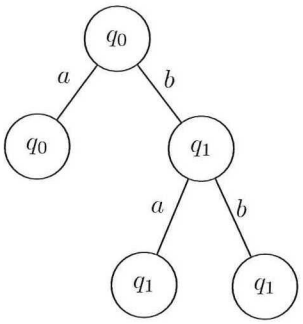


Figure 7: The testing tree conform conjecture DFA H_1.

One drawback of the W-method is that it requires knowledge about the maximum number of states m that a correct version conjecture might have. Chow solves this variable as to be determined by human judgement. Since m functions as an upper-bound estimation, m can be any number as long as it is larger or equal to the number of states in the hypothesized conjecture n. The test cases are then derived in $P \cdot (W \cdot \bigcup_{i=0}^{m-n} \Sigma^i)$. Chow summarizes $W \cdot \bigcup_{i=0}^{m-n} \Sigma^i$ to be the set Z and because $\Sigma^0 = \{\varepsilon\}$, Z can be written as $W \cup \Sigma \cdot W \cup \cdots \cup \Sigma^{m-n} \cdot W$.

Example 1.11. To exemplify the test suite $P \cdot Z$ that would be generated for the hypothesized conjecture H_1, one should recall that $P = \{\varepsilon, a, ba, bb\}$, $W = \{a\}$ and $\Sigma = \{a, b\}$. Because $Q = \{q_0, q_1\}$ the size of Q: $|Q| = n = 2$. Let m then be an arbitrary estimation that satisfies $m \leq n$, one could choose $m = 4$. Set Z can then be determined by $Z = W \cdot \bigcup_{i=0}^{m-n} \Sigma^i = \{a\} \cdot \bigcup_{i=0}^{2} \{a, b\}^i = \{a\} \cup \{a, b\}^1 \cdot \{a\} \cup \{a, b\}^2 \cdot \{a\} = \{a\} \cup \{a, b\} \cdot \{a\} \cup \{aa, ab, ba, bb\} \cdot \{a\} = \{a\} \cup \{aa, ba\} \cup \{aaa, aba, baa, bba\} = \{a, aa, ba, aaa, aba, baa, bba\}$.

The test suite $P \cdot Z$ created with the W-method is run on both the SUT and the hypothesized model. If the generated test suite is executed in the order depicted above, input sequence *aba* yields a different result. This input sequence is then provided as a counterexample to the learner by the equivalence oracle. In example 1.3 the learner recieved the counterexample word $w = ab$ which demonstrates the same difference in behaviour as the test input sequence *aba* generated with Chow's method.

Minimal Separating Sequences for All Pairs of states

One drawback of the W-method is that the number of test cases rapidly grows in the size of the alphabet, number of states, the maximum depth and especially the length of the characterizing set. Instead of each combination in the permutation of the alphabet up until a certain depth, one can utilize smarter techniques for finding separating sequences. In *Minimal Separating Sequences for All Pairs of States*[9] Smetsers et al. propose an improved modification based on Hopcroft's framework [10] for finding the shortest input sequence that distinguishes two inequivalent states in a DFA. Minimal separating sequences play an central role in conformance testing methods and hence can be applied to establish an equivalence oracle for learning automata. Separating sequences function as an input for test suite generation, which like Chow's W-method can determine whether a hypothesized conjecture is equivalent to an abstraction of a system

under test.

Smetsers et al. identify minimal separating sequences by systematically refining state partitions to ensure a minimal DFA minimal access sequence. The operational refinement information is maintained in a tree-like data structure called a splitting tree, which was first introduced by Lee and Yannakakis[11]. The continuous of this section elaborates on how Smetsers utilize partitions and splitting trees to determine the minimal separating sequences.

Let the SUT's behavior be abstracted to the DFA $\mathcal{A} = \{Q, \Sigma, \delta, q_0, F\}$ and let H be the hypothesized model of $\mathcal{A}$. A state *partition* P of Q is a set of pairwise disjoint non-empty subsets of Q whose union is exactly Q. Each subset in P is called a block. If P and P' are partitions of Q and every block of P' is contained in a block of P, then P' is called a refinement of P. The algorithm starts with the trivial partition $P = \{Q\}$ and refine P until the partition is valid, that is when all equivalent states are in the same block. Let B be a block in P and a be an input. The partition refinement algorithm splits blocks because of two reasons. B can be split with respect to the output after a if the set $\lambda(B, a)$ contains more than one output. In this instance each distinct output in $\lambda(B, a)$ defines a new block. Alternatively B can be split with respect to the successing state after input a. In the latter instance each block that contains a state in $\delta(B, a)$ defines a new block. The refinement process is continued until for all pairs of states $q, q' \in Q$ that are contained in the same block and for all inputs $a \in \Sigma$ hold that $\lambda(q, a) = \lambda(q', a)$. At this point the partition is classified as *acceptable*. Final refinement is reached when a partition is acceptable and for all $a \in \Sigma$ hold that for all $q, q' \in Q$ in the same block, the new states $\delta(q, a)$ and $\delta(q', a)$ are also in the same block. At this final point, the partition is classified as *stable*.

Separating sequences are determined by the type of split and the information is maintained in a splitting tree. Smetser redefines the splitting tree, in order to be applicable for the situation where it stores minimal separating sequences, as follows:

Definition 1 (*Splitting Tree*). A splitting tree for $\mathcal{A}$ is a rooted tree T with a finite set of nodes with the following properties:

- Each node in T is labelled by a subset of Q, denoted $l(u)$
- Each leaf nodes u, $l(u)$ corresponds to a block in the stale partition P.
- Each non-leaf node u, $l(u)$ is partitioned by the labels of its children, thus the root node if labeled Q.
- Each non-leaf node u is associated with a sequence $\sigma(u)$ that separates states contained in different children of u.

$C(u)$ denotes the set of children of a node u in T and the lowest common ancestor for a set $Q' \subseteq Q$ is a node u denoted by $lca(Q')$. For a pair of states, the shorthand notation $lca(s, t)$ is used instead of $lca(\{s, t\})$ to denote the lowest common ancestor of s and t. At any given time, the labels of the leafs of T, denoted as $P(T)$ together form a partition of Q. A tree T is valid, iff $P(T)$ is valid as well. A leaf u within block $B = l(u)$ can be split in the same way partition blocks are split, either based on output or the consecutive state for an input a. If the block is split based on output, $\sigma(u)$ is set to a and a new

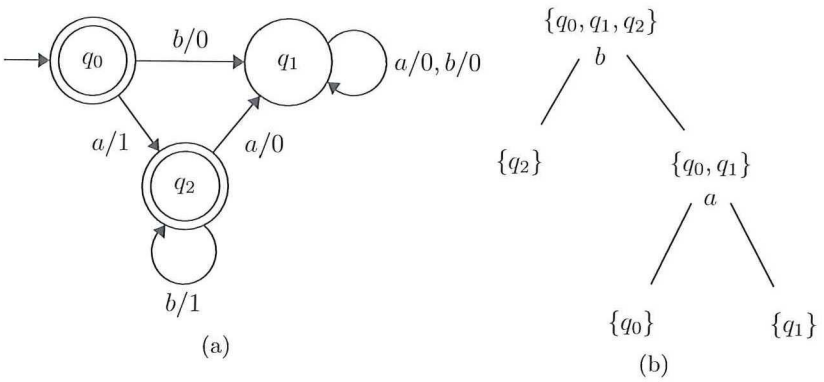


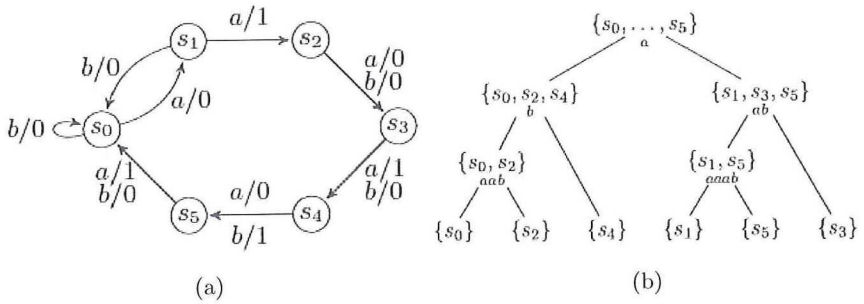
Figure 8: (a): $\mathcal{A}'$ the mealy machine representation of $\mathcal{A}$ (b): splitting tree representation of $\mathcal{A}'$

node for each subset of B that produces the same output for a are appended as children for u. If the block is split based on the consecutive state, then the node $v = lca(\delta(B, a))$ has at least two children whose labels contain elements of $\delta(B, a)$. This information is utilized to create a new child of u labelled $\{s \in B | \delta(s, a) \in l(w)\}$ for each node w in $C(v)$. The state seperator $\sigma(u)$ is set to $a \cdot \sigma(v)$.

In order to create a stable splitting tree for the example DFA $\mathcal{A}$ shown in Figure 1 one should note that the partition refinement algorithm only works for Mealy Machines. The reason for this is because splitting with respect to output only works if every transition produces an output, which is not the case for a general DFA. To map the algorithm to the running example, $\mathcal{A}$ can be regarded as an mealy machine where transitions output 1 if the resulting state is an accepting state and 0 if the resulting state is a rejecting state. This results in a Mealy Machine version of $\mathcal{A}$ depicted in Figure 8a.

Example 1.12. Figure 8b shows the splitting tree for mealy machine $\mathcal{A}'$, it is generated by Smetsers' algorithm as follows. The first step is setting the root node to Q, hence the root node is labelled $\{q_0, q_1, q_2\}$. Since q_2 gives another output for input b opposed to the output of states q_0 and q_1, the root node is split based on this output after b. The node labeled $\{q_2\}$ cannot be split anymore, as it contains one single element, the algorithm determines that states q_0 and q_1 yield a different output for input a. Since the nodes cannot be split anymore, the tree is complete.

The algorithm for generating splitting trees can be used to obtain separating sequences, but they are not necessarily minimal separating sequences. This is the case when child nodes have a smaller sequence than their parents. The sequences can be shortened, as the parents' can be split first. This does not appear in the elementary example from Figure 8b, but it shows in Smetser's example DFA and splitting tree (Figures 9a and 9b. The labels of the child node are arbitrarily larger than the labels of the parent nodes. Splitting trees that are obtained in a way that ensures that each k'th level has a label of size k, are *layered* splitting trees. Each layer in the tree consists of nodes for which the associated separating sequences have the same length. During the construction



(a) (b)

Figure 9: (a): Smetsers example mealy machine and (b) complete splitting tree for the mealy machine.

of the splitting tree, each layer should be as large as possible, before continuing to the next one. The following two definitions aid the process of obtaining such splitting trees:

Definition 2 (*k-stable Splitting Trees*). 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 3 (*Minimal Splitting Trees*). A splitting tree T is minimal if for all states s and t in the same leaf we have $\lambda(s, x) = \lambda(t, x)$ implies $|x| \geq |\sigma(lca(s, t))|$ for all $x \in I^{\leq k}$

The recipe for establishing a minimal splitting tree is to create a splitting tree splitting blocks only with respect to output and next assuring that for $k = 1 \ldots |Q|$ the tree is k-stable and minimal.

Example 1.13. The example of Smetsers yield Figure 10 as a result of this process. Basically it starts splitting blocks with respect to output as shown in Figure 9b until the node labeled $\{s_0, s_2\}$ appears. This node cannot be split based on output because in DFA figure 9a it shows that all outgoing transitions of both states yield the same output. At this point $k = 1$ and one can observe that the sequence of $lca(\{s_0, s_2\}, a))$ has length 2, which is too long for the value of k. One thus has to move on to the next input. It is then possible to split this block w.r.t. the state after b, thus the associated sequence is ba. If this is continued for all levels and all blocks, the splitting tree and partition are identical to the former splitting tree, except that the labels are shorter.

In conclusion, following this recipe for the establishment of k-stable and minimal splitting trees, result in the shortest seperating sequence.

The complete and minimal splitting trees can henceforth be used to extract relevant information for the characterization set for the W-method.

Lemma 1. *Let be a complete splitting tree, then* $\{\sigma(u)|u \in T\}$ *is a characterization set.*

Proof Let $W = \{\sigma(u)|u \in T\}$and let $s, t \in Q$ be inequivalent states. A set $W \subset \Sigma^*$ is called a characterization set if for every pair of inequivalent states s,t there is a sequence $w \in W$ such that $\lambda(s, w) \neq \lambda(t, w)$ ([9], definition 17).

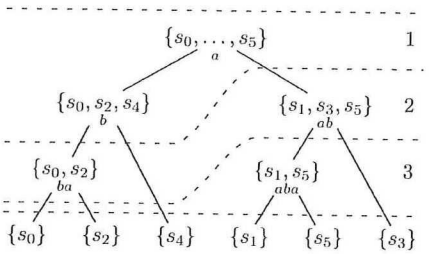


Figure 10: Minimal splitting tree for Smetsers mealy machine

Because s, t are inequivalent and T is complete, s and t are contained in different leaves of T. Hence $u = lca(s, t)$ exists and furthermore $\sigma(u) \in W$. This shows that W is a characterisation set. □

References

[1] Michael Sipser. *Introduction to the Theory of Computation.* Cengage Learning, 2012.

[2] Michael J Kearns and Umesh Virkumar Vazirani. *An introduction to computational learning theory.* MIT press, 1994.

[3] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[4] E Mark Gold. Complexity of automaton identification from given data. *Information and control*, 37(3):302–320, 1978.

[5] Malte Isberner, Falk Howar, and Bernhard Steffen. The ttt algorithm: A redundancy-free approach to active automata learning. In *RV*, pages 307–322, 2014.

[6] Therese Berg and Harald Raffelt. Model checking. *Model-Based Testing of Reactive Systems*, pages 77–84, 2005.

[7] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.

[8] Arthur Gill et al. *Introduction to the theory of finite-state machines.* McGraw-Hill, 1962.

[9] Rick Smetsers, Joshua Moerman, and David N Jansen. Minimal separating sequences for all pairs of states. In *International Conference on Language and Automata Theory and Applications*, volume 9618, pages 181–193. Springer, 2016.

[10] J.E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computation*, pages 189–196, 1971.

[11] David Lee and Mihalis Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Transactions on computers*, 43(3):306–320, 1994.