Date: October 3, 2008

Learning Minimal Separating DFAs for Compositional Verification *

Yu-Fang Chen[2], Azadeh Farzan[1],
Edmund M. Clarke[1], Yih-Kuen Tsay[2], and Bow-Yaw Wang[3]

[1]Carnegie Mellon University [2]National Taiwan University [3]Academia Sinica

Abstract. Algorithms for learning a minimal separating DFA of two disjoint regular languages have been proposed and adapted for different applications. One of the most important applications is learning minimal contextual assumptions in automated compositional verification. We propose in this paper an efficient learning algorithm, called L^{Sep}, that learns and generates a minimal separating DFA. Our algorithm has a quadratic query complexity in the product of sizes of the minimal DFA's for the two input languages. In contrast, the most recent algorithm of Gupta *et al.* has an exponential query complexity in the sizes of the two DFA's. Moreover, experimental results show that our learning algorithm significantly outperforms all existing algorithms on randomly-generated example problems. We detail how our algorithm can be adapted for automated compositional verification. The adapted version is evaluated on the LTSA benchmarks and compared with other automated compositional verification approaches. The result shows that our algorithm surpasses others in 30 of 49 benchmark problems.

1 Introduction

Compositional verification is seen by many as a promising approach for scaling up Model Checking [**?**] to larger designs. In the approach, one applies a compositional inference rule to break the task of verifying a system down to the subtasks of verifying its components. The compositional inference rule is usually in the so-called *assume-guarantee* style. One widely used assume-guarantee rule, formulated from a language-theoretic view, is the following:

$$\frac{\mathcal{L}(M_1) \cap \mathcal{L}(A) \subseteq \mathcal{L}(P) \quad \mathcal{L}(M_2) \subseteq \mathcal{L}(A)}{\mathcal{L}(M_1) \cap \mathcal{L}(M_2) \subseteq \mathcal{L}(P)}$$

We assume that the behaviors of a system or component are characterized by a language and any desired property is also described as a language. The parallel composition of two components is represented by the intersection of the languages of the two components. A system (or component) satisfies a property if the language of the system (or component) is a subset of the language

* This research was sponsored by the iCAST project of the National Science Council, Taiwan, under the grant no. NSC96-3114-P-001-002-Y

of the property. The above assume-guarantee rule then says that, to verify that the system composed of components M_1 and M_2 satisfies property P, one may instead verify the following two conditions: (1) component M_1 satisfies (guarantees) P under some contextual assumption A and (2) component M_2 satisfies the contextual assumption A.

The main difficulty in applying assume-guarantee rules to compositional verification is the need of human intervention to find contextual assumptions. For the case where components and properties are given as regular languages, several automatic approaches have been proposed to find contextual assumptions [**?**,**?**] based on the machine learning algorithm L^* [**?**,**?**]. Following this line of research, there have been results for symbolic implementations [**?**,**?**], various optimization techniques [**?**,**?**], an extension to liveness properties [**?**], performance evaluation [**?**], and applications to problems such as component substitutability analysis [**?**]. However, all of the above suffer from the same problem: they do not guarantee finding a small assumption even if one exists.

The problem of finding a minimal assumption for compositional verification can be reduced to the problem of finding a minimal separating DFA of two disjoint regular languages [**?**]. A DFA separates the two input languages if its language contains one of the input languages and is disjoint from the other input language. Several approaches [**?**,**?**,**?**] have been proposed to find a minimal separating DFA automatically. However, all of those approaches are computationally expensive. In particular, the most recent algorithm of Gupta *et al.* [**?**] has an exponential query complexity in the sizes of the minimal DFA's of the two input languages.

In this paper we propose a more efficient learning algorithm, called L^{Sep}, for finding minimal separating DFA's. The query complexity of our algorithm is quadratic in the product of the sizes of the two minimal DFA's. Moreover, our algorithm utilizes membership queries to accelerate learning and has a more compact representation of the samples collected from the queries. Experiments show that L^{Sep} significantly outperforms other algorithms on a large set of randomly-generated example problems.

We then give an adaptation of the L^{Sep} algorithm for automated compositional verification and evaluate its performance on the LTSA benchmarks [**?**]. The result shows that the adapted version of L^{Sep} surpasses other compositional verification algorithms on 30 of 49 benchmark problems. Besides automated compositional verification, algorithms for learning a minimal separating DFA have found other applications. Grinchtein *et al.* [**?**] used such an algorithm as the basis for *learning network invariants of parameterized systems*. Pena *et al.* [**?**] argued that such an algorithm can be used for the *incompletely specified finite state machine (ISFSM) minimization problem* because it can solve some very difficult instances. Although we only discuss the application of L^{Sep} to automated compositional verification in this paper, the algorithm can certainly be adapted for these other applications as well.

The remainder of this paper is organized as follows. Section 2 contains the preliminaries. Section 3 provides an overview of our idea for learning a minimal

separating DFA. In Section 4 we present the L^{Sep} algorithm. We then give an adaptation of L^{Sep} for automated compositional verification in Section 5. The experimental results are detailed in Section 6. Finally, in Section 7, we summarize our contributions and suggest directions for further work.

2 Preliminaries

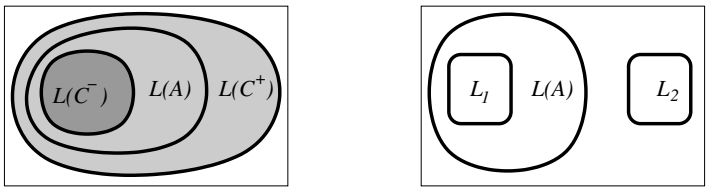
An *alphabet* Σ is a finite ordered set. A finite *string* over Σ is a finite sequence of elements of Σ. The empty string is represented by λ. For two strings $u = u_1 \dots u_n$ and $v = v_1 \dots v_m$ where $u_i, v_j \in \Sigma$, define the concatenation of the two strings as $uv = u_1 \dots u_n v_1 \dots v_m$. For a string u, u^n is recursively defined as uu^{n-1} with $u^0 = \lambda$. String concatenation is naturally extended to sets of strings where $S_1 S_2 = \{s_1 s_2 |\ s_1 \in S_1, s_2 \in S_2\}$.

A string u is a *prefix* (respectively *suffix*) of another string v if and only if there exists a string $w \in \Sigma^*$ such that $v = uw$ (respectively $v = wu$). A set of strings S is called *prefix-closed* (respectively *suffix-closed*) if and only if for all $v \in S$, if u is a prefix (respectively suffix) of v, then $u \in S$. The set of all finite strings over Σ is denoted by Σ^*, and Σ^+ is the set of all nonempty finite strings over Σ (naturally, $\Sigma^+ = \Sigma^* \backslash \{\lambda\}$). The length of string u is denoted by $|u|$ and $|\lambda| = 0$. A *deterministic finite automaton (DFA)* $\mathcal{A}$ is a tuple $(\Sigma, S, s_0, \delta, F)$, where Σ is an alphabet, S is a finite set of states, s_0 is the initial state, $\delta : S \times \Sigma \to S$ is the transition relation, and $F \subseteq S$ is a set of *accepting* states. The transition function δ is extended to strings of any length in the natural way. A string u is accepted by $\mathcal{A}$ if and only if $\delta(s_0, u) \in F$. Define $\mathcal{L}(\mathcal{A}) = \{u \mid u \text{ is accepted by } \mathcal{A}\}$. A language $L \subseteq \Sigma^*$ is *regular* if and only if there exists a finite automaton $\mathcal{A}$ such that $L = \mathcal{L}(\mathcal{A})$. The notation $\overline{L}$ denotes the complement with respect to Σ^* of the regular language L. Let $|L|$ denote the number of states of the minimal DFA that recognizes L and $|\mathcal{A}|$ denote the number of states in the DFA $\mathcal{A}$.

Definition 1. *(Three-Valued Deterministic Finite Automata) A 3-valued deterministic finite automaton (3DFA) is a tuple $\mathcal{C} = (\Sigma, S, s_0, \delta, Acc, Rej, Dont)$, where Σ is an alphabet, S is a finite set of states, s_0 is the initial state, $\delta : S \times \Sigma \to S$ is the transition relation, Acc is the set of accepting states, Rej is the set of rejecting states, and Dont is the set of don't care states.*

The set of states S is partitioned into the three sets Acc, Rej, and $Dont$. In $\mathcal{C}$, a string u is *accepted* if $\delta(s_0, u) \in Acc$, is *rejected* if $\delta(s_0, u) \in Rej$, and is a *don't care* string if $\delta(s_0, u) \in Dont$. Let $\mathcal{C}^+$ represent the DFA $(\Sigma, S, s_0, \delta, Acc \cup Dont)$, where all don't care states become accepting states, and $\mathcal{C}^-$ represent the DFA $(\Sigma, S, s_0, \delta, Acc)$, where all don't care states become rejecting states. By definition, we have that $\mathcal{L}(\mathcal{C}^-)$ is the set of accepted strings in $\mathcal{C}$ and $\overline{\mathcal{L}(\mathcal{C}^+)}$ is the set of rejected strings in $\mathcal{C}$.

A 3DFA $\mathcal{C}$ is *equivalent* to a 3DFA $\mathcal{C}'$ if and only if they coincide on the accepted and rejected strings. A DFA $\mathcal{A}$ is *consistent with* a 3DFA $\mathcal{C}$ if and only if $\mathcal{A}$ accepts all strings that $\mathcal{C}$ accepts, and rejects all strings that C rejects. Consequently, $\mathcal{A}$ accepts all strings in $\mathcal{L}(\mathcal{C}^-)$ (set of strings accepted by $\mathcal{C}$),



(a) DFA $\mathcal{A}$ consistent with $\mathcal{C}$

(b) DFA $\mathcal{A}$ separating L_1 and L_2

Fig. 1. Consistent and Separating DFAs.

and rejects all strings in $\overline{\mathcal{L}(\mathcal{C}^+)}$ (set of strings rejected by $\mathcal{C}$); in other words $\mathcal{L}(\mathcal{C}^-) \subseteq \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{C}^+)$. Figure 1(a) illustrates a DFA $\mathcal{A}$ consistent with the 3DFA $\mathcal{C}$: the bounding box is the set of strings Σ^*. The shaded area in the figure represents $\mathcal{L}(\mathcal{C}^+)$, and the dark shaded area represents $\mathcal{L}(\mathcal{C}^-)$.

A DFA $\mathcal{A}$ is the minimal consistent DFA of a 3DFA $\mathcal{C}$ if it is consistent with $\mathcal{C}$ and has the smallest number of states among all DFA's consistent with C.

A DFA $\mathcal{A}$ *separates* two disjoint regular languages L_1 and L_2 if and only if $L_1 \subseteq \mathcal{L}(\mathcal{A}) \subseteq \overline{L_2}$. For example, in Figure 1(b) $\mathcal{A}$ is a separating DFA for L_1 and L_2. Note that there may be many DFAs that separate languages L_1 and L_2. A minimal separating DFA for L_1 and L_2 is a smallest (in terms of number of states) separating DFA among all separating DFAs for L_1 and L_2.

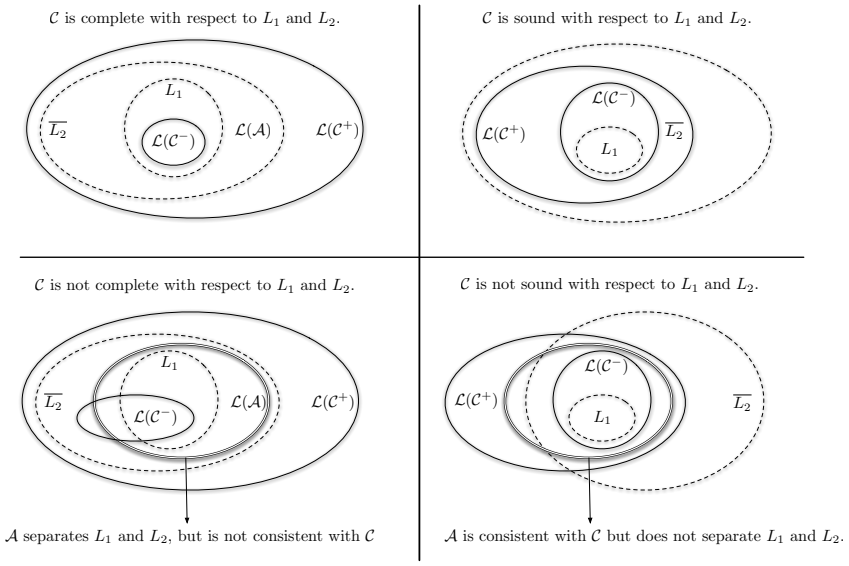


Fig. 2. Soundness/Completeness of a 3DFA $\mathcal{C}$ with respect to languages L_1 and L_2.

A 3DFA $\mathcal{C}$ is *sound with respect to* L_1 *and* L_2 if any DFA consistent with $\mathcal{C}$ separates L_1 and L_2. A 3DFA $\mathcal{C}$ is *complete with respect to* L_1 *and* L_2 if any separating DFA with respect to L_1 and L_2 is consistent with $\mathcal{C}$. Let us give look at this more closely using an example. Figure 2 illustrates examples of soundness and completeness scenarios. In the top part, we have show cases when $\mathcal{C}$ is sound (top-right) or complete (top-left) with respect to L_1 and L_2. In the bottom part, the soundness and completeness conditions are not satisfied. When $\mathcal{C}$ is not complete with respect to L_1 and L_2 (bottom-right), there exists a DFA $\mathcal{A}$ that separates L_1 and L_2, but is not consistent with $\mathcal{C}$. Reversely, when $\mathcal{C}$ is not sound with respect to L_1 and L_2 (bottom-left), there exists a DFA $\mathcal{A}$ that is consistent with $\mathcal{C}$, but does not separate L_1 and L_2.

Proposition 1. *Let L_1 and L_2 be disjoint regular languages and $\mathcal{C}$ be a 3DFA. Then*

1. *$\mathcal{C}$ is sound for L_1 and L_2 if and only if $L_1 \subseteq \mathcal{L}(\mathcal{C}^-)$ and $\mathcal{L}(\mathcal{C}^+) \subseteq \overline{L_2}$;*
2. *$\mathcal{C}$ is complete for L_1 and L_2 if and only if $\mathcal{L}(\mathcal{C}^-) \subseteq L_1$ and $\overline{L_2} \subseteq \mathcal{L}(\mathcal{C}^+)$.*

3 Overview of Learning Minimal Separating DFA

Our key idea is to use a 3DFA as a succinct representation for the samples collected from the input languages L_1 and L_2. Exploiting the three possible acceptance outcomes of a 3DFA (accept, reject, and don't care), we encode strings from L_1 and L_2 in a 3DFA $\mathcal{C}$ as follows. All strings of L_1 are accepted by $\mathcal{C}$ and all strings in L_2 are rejected by $\mathcal{C}$. The remaining strings take $\mathcal{C}$ into a don't care states. Therefore, $L_1 = \mathcal{L}(\mathcal{C}^-)$ and $\overline{L_2} = \mathcal{L}(\mathcal{C}^+)$. Recall that for a DFA $\mathcal{A}$ consistent with 3DFA $\mathcal{C}$, we have $\mathcal{L}(\mathcal{C}^-) \subseteq \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{C}^+)$. Moreover, $L_1 \subseteq \mathcal{L}(\mathcal{A}) \subseteq \overline{L_2}$ for any DFA $\mathcal{A}$ separating L_1 and L_2. Therefore, given $L_1 = \mathcal{L}(\mathcal{C}^-)$ and $\overline{L_2} = \mathcal{L}(\mathcal{C}^+)$, any DFA separating L_1 and L_2 is consistent with $\mathcal{C}$, and vice versa. Thus, the problem of finding the minimal separating DFA for L_1 and L_2 reduces to the problem of finding the minimal DFA consistent with the 3DFA $\mathcal{C}$.

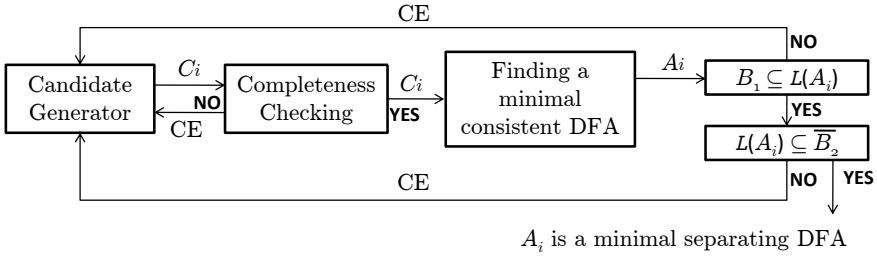


Fig. 3. Finding Minimal Separating Automata – Overview

Figure 3 depicts the flow of our algorithm. The *candidate generator* produces a series of candidate 3DFAs $\mathcal{C}_i$ targeting the 3DFA $\mathcal{C}$ using an extension of L^*. The *completeness checker* examines whether $\mathcal{C}_i$ is complete with respect to L_1 and L_2, i.e. all DFAs separating L_1 and L_2 are consistent with $\mathcal{C}_i$.

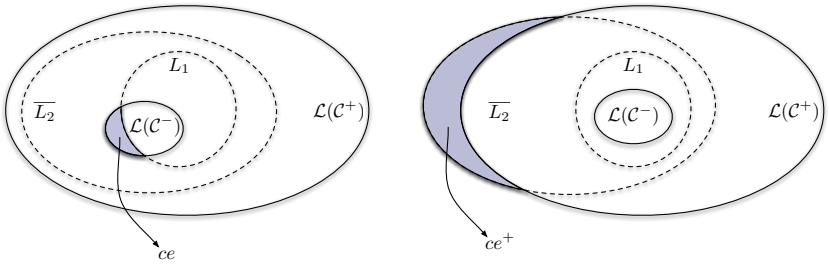


Fig. 4. Checking completeness: $\mathcal{L}(\mathcal{C}^-) \subseteq L_1$ (left), and $\overline{L_2} \subseteq \mathcal{L}(\mathcal{C}^+)$ (right).

By Proposition 1, checking completeness reduces to checking whether $\mathcal{L}(\mathcal{C}^-) \subseteq L_1$ and $\overline{L_2} \subseteq \mathcal{L}(\mathcal{C}^+)$. Figure 4 illustrates the two cases where each of these conditions fail where a "no" answer, together with a counterexample, is returned to the candidate generator. If both conditions are satisfied, i.e. $\mathcal{C}_i$ is complete with respect to L_1 and L_2, the next step is to compute a minimal DFA $\mathcal{A}_i$ consistent with $\mathcal{C}_i$. Finally, we check whether $\mathcal{A}_i$ separates L_1 and L_2, i.e. $L_1 \subseteq \mathcal{L}(\mathcal{A}_i)$ and $\mathcal{L}(\mathcal{A}_i) \subseteq \overline{L_2}$. Figure 5 demonstrates the two scenarios where these checks may fail. Note that $\mathcal{C}_i$ is complete with respect to L_1 and L_2, $\mathcal{A}_i$ is consistent with $\mathcal{C}_i$, but $\mathcal{A}_i$ does not separate L_1 and L_2. Therefore, counterexamples ce_1 and ce_2 are returned to the candidate generator which uses them to refine $\mathcal{C}_i$ in the next iteration. Eventually, the candidate $\mathcal{C}_i$ converges to the 3DFA $\mathcal{C}$ which is sound and complete with respect to L_1 and L_2. At this point, the minimal DFA consistent with $\mathcal{C}$ is also the minimal DFA separating L_1 and L_2.

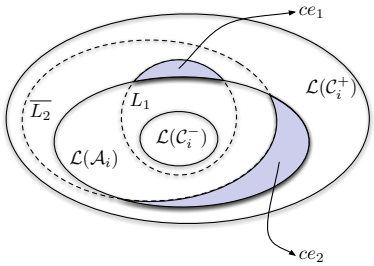


Fig. 5. Checking whether $\mathcal{A}_i$ separates L_1 and L_2.

4 The L^{Sep} Algorithm

L^{Sep} is an active learning algorithm which computes a minimal separating DFA for two disjoint regular languages L_1 and L_2. It assumes a teacher that answers the following two types of queries:

- **membership queries** on a string w, where the teacher returns *true* if w is in L_1, *false* if w is in L_2, and *don't care* otherwise.
- **containment queries** where the teacher solves language containment problems of the following four types: (i) $L_1 \subseteq \mathcal{L}(A_i)$, (ii) $L_1 \supseteq \mathcal{L}(A_i)$, (iii) $L_2 \subseteq \mathcal{L}(A_i)$, and (iv) $L_2 \supseteq \mathcal{L}(A_i)$. The teacher returns "YES" if the property holds, and "NO" with a counterexample otherwise.

As described in Section 3, the L^{Sep} algorithm performs the following steps to find a minimal separating DFA $\mathcal{A}$ for the languages L_1 and L_2 iteratively.

Candidate Generation

The Candidate Generator extends the observation table in L^* [**?**] to allow entries with don't cares. An *observation table* $\langle S, E, T\rangle$ is a triple of a prefix-closed set S of strings, a set E of experimental strings, and a function T from $(S \cup S\Sigma) \times E$ to $\{T, F, D\}$ (Figure 6). Let $\alpha \in S \cup S\Sigma$ and $\beta \in E$. The function T maps $\pi = (\alpha, \beta)$ to T if $\alpha\beta \in L_1$; it maps π to F if $\alpha\beta \in L_2$; otherwise T maps π to D. In the observation table of Figure 6, the entry for (ba, b) is T because the string $bab \in L_1$.

	λ	b
λ	F	D
b	D	D
ba	F	T
bab	T	T
a	F	D
bb	D	D
baa	F	T
$baba$	F	D
$babb$	T	T

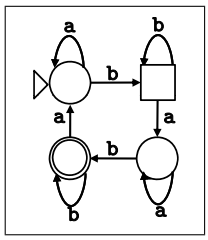


Fig. 6. An observation Table and Its Corresponding 3DFA. The square node denotes a don't care state.

The Candidate Generator constructs the observation table by posing membership queries. It generates a 3DFA $\mathcal{C}_i$ based on the observation table. If the 3DFA $\mathcal{C}_i$ is unsound or incomplete, the Candidate Generator refines the observation table by counterexamples returned by containment queries and generates another 3DFA. Let n be the size of the minimal sound and complete 3DFA and m be the length of the longest counterexample returned by containment queries. The Candidate Generator is guaranteed to find a sound and complete 3DFA with $O(n^2 + n \log m)$ membership queries. Moreover, it generates at most $n - 1$ incorrect 3DFA's. We refer readers to the appendix for details.

Completeness Checking

As elicited in Section 3, L^{Sep} finds the minimal DFA separating L_1 and L_2 by computing the minimal DFA consistent with $\mathcal{C}_i$. To make sure all separating DFA for L_1 and L_2 are considered, the L^{Sep} algorithm checks whether $\mathcal{C}_i$ is complete.

The completeness check is done by containment queries. L^{Sep} first builds the DFA's $\mathcal{C}_i^+$ and $\mathcal{C}_i^-$. It then submits the containment queries $L_1 \supseteq \mathcal{L}(\mathcal{C}_i^-)$

and $\overline{L_2} \subseteq \mathcal{L}(\mathcal{C}_i^+)$. If either of these queries fails, a counterexample is sent to the Candidate Generator to refine $\mathcal{C}_i$. Note that several iterations between Candidate Generator and Completeness Check may be needed to find a complete 3DFA.

Finding the Minimal Consistent DFA

After the Completeness Check, we are sure that any DFA separating L_1 and L_2 is also consistent with $\mathcal{C}_i$. Moreover, we have the following lemma characterizing the sizes of minimal separating DFA's and minimal consistent DFA's.

Lemma 1. *Let $\widehat{\mathcal{A}}$ be a minimal separating DFA of L_1 and L_2, and $\mathcal{A}_i$ be a minimal DFA consistent with $\mathcal{C}_i$. If $\mathcal{C}_i$ is complete, then $|\widehat{\mathcal{A}}| \geq |\mathcal{A}_i|$.*

Proof. By completeness, any separating DFA of L_1 and L_2 is consistent with $\mathcal{C}_i$. Hence the minimal separating DFA $\widehat{\mathcal{A}}$ is consistent with $\mathcal{C}_i$. We have $|\widehat{\mathcal{A}}| \geq |\mathcal{A}_i|$ by the minimality of $\mathcal{A}_i$. □

Hence, if any minimal DFA consistent with $\mathcal{C}_i$ also separates L_1 and L_2, it is a minimal separating DFA for L_1 and L_2. Our next step is to compute the minimal DFA consistent with $\mathcal{C}_i$. We reduce the problem to the minimization problem of incompletely specified finite state machines [**?**]. The L^{Sep} algorithm translates the 3DFA $\mathcal{C}_i$ into an incompletely specified finite state machine $\mathcal{M}$. It then invokes the algorithm in [**?**] to obtain a minimal finite state machine $\mathcal{M}_i$ consistent with $\mathcal{M}$. Finally, $\mathcal{M}_i$ is converted to a DFA $\mathcal{A}_i$.

Soundness Checking

After the minimal DFA $\mathcal{A}_i$ consistent with $\mathcal{C}_i$ is computed, L^{Sep} verifies whether $\mathcal{A}_i$ separates L_1 and L_2 by the containment queries $L_1 \subseteq \mathcal{L}(\mathcal{A}_i)$ and $\mathcal{L}(\mathcal{A}_i) \subseteq \overline{L_2}$. There are three possible outcomes:

- $L_1 \subseteq \mathcal{L}(\mathcal{A}_i) \subseteq \overline{L_2}$. Hence, $\mathcal{A}_i$ is in fact a separating DFA for L_1 and L_2. By Lemma 1, $\mathcal{A}_i$ is a minimal separating DFA for L_1 and L_2.
- $L_1 \nsubseteq \mathcal{L}(\mathcal{A}_i)$. Hence, there is a string $u \in L_1 \setminus \mathcal{L}(\mathcal{A}_i)$. Moreover, we have $\mathcal{L}(\mathcal{C}_i^-) \subseteq \mathcal{L}(\mathcal{A}_i)$ for $\mathcal{A}_i$ is consistent with $\mathcal{L}(\mathcal{C}_i)$. Hence $u \in L_1 \setminus \mathcal{L}(\mathcal{C}_i^-)$. u is a counterexample for the soundness of $\mathcal{C}_i$. It is sent to the Candidate Generator to refine the 3DFA in the next iteration.
- $\mathcal{L}(\mathcal{A}_i) \nsubseteq \overline{L_2}$. Hence, there exists a string $v \in \mathcal{L}(\mathcal{A}_i) \setminus \overline{L_2}$. v is in fact a counterexample for the soundness of $\mathcal{C}_i$ by a symmetric argument. It is sent to the Candidate Generator as well.

4.1 Correctness and Complexity Analysis

The following Theorem proves that the L^{Sep} algorithm is correct.

Theorem 1. *The L^{Sep} algorithm terminates and outputs a minimal separating DFA for L_1 and L_2.*

Proof. The statement follows from the following observations:

1. Each iteration of the L^{Sep} algorithm terminates;
2. If the minimal consistent DFA does not separate L_1 and L_2, a counterexample witnessing the incompleteness or unsoundness of $\mathcal{C}_i$ is sent to the Candidate Generator;
3. Because of 2, the Candidate Generator will eventually converge to the sound and complete 3DFA $\mathcal{C}$ defined in Section 3. In this case, the minimal consistent DFA is a minimal separating DFA for L_1 and L_2. Hence L^{Sep} terminates when $\mathcal{C}$ is found.

□

We now estimate the number of queries used in the L^{Sep} algorithm. The following lemma states an upper bound on the size of the minimal sound and complete 3DFA.

Lemma 2. *Let $\mathcal{B}_i$ be the minimal DFA accepting the regular language L_i for $i = 1, 2$. The size of the minimal 3DFA $\mathcal{C}$ such that $\mathcal{L}(\mathcal{C}^-) = L_1$ and $\mathcal{L}(\mathcal{C}^+) = \overline{L_2}$ is smaller than $|\mathcal{B}_1| \times |\mathcal{B}_2|$.*

Proof. Let $\mathcal{B}_1 = (\Sigma, S, s_0, \delta, Acc)$ and $\mathcal{B}_2 = (\Sigma, S', s'_0, \delta', Acc')$. We can create a 3DFA $\hat{\mathcal{C}} = (\Sigma, \hat{S}, \hat{s_0}, \hat{\delta}, \widehat{Acc}, \widehat{Rej}, \widehat{Dont})$ that satisfies $\mathcal{L}(\hat{\mathcal{C}}^-) = B_1$ and $\mathcal{L}(\hat{\mathcal{C}}^+) = \overline{B_2}$ as follows.

- $\hat{S} = S \times S'$;
- $\hat{s_0} = (s_0, s'_0)$;
- $\hat{\delta}((s_1, s'_1), a) = (s_2, s'_2)$ if $\delta(s_1, a) = s_2$, and $\delta(s'_1, a) = s'_2$.
- A state $(s, s') \in \widehat{Acc}$ if $s \in Acc$ and $s' \in Acc'$.
- A state $(s, s') \in \widehat{Rej}$ if $s \notin Acc$ and $s' \notin Acc'$.
- A state $(s, s') \in \widehat{Dont}$ if $s \notin Acc$ and $s' \in Acc'$.

Note that the state (s, s') with $s \in Acc$ and $s' \notin Acc'$ is not reachable in $\hat{\mathcal{C}}$ for $\overline{L_2} \supseteq L_1$.

By definition, the size of $\hat{\mathcal{C}}$ is less than $|\mathcal{B}_1| \times |\mathcal{B}_2|$. The size of $\mathcal{C}$ is no more than the size of $\hat{\mathcal{C}}$ because it is the minimal 3DFA that satisfies $\mathcal{L}(\hat{\mathcal{C}}^-) = L_1$ and $\mathcal{L}(\hat{\mathcal{C}}^+) = \overline{L_2}$. It follows that the size of $\mathcal{C}$ is less than $|\mathcal{B}_1| \times |\mathcal{B}_2|$. □

By Lemma 2, the query complexity of L^{Sep} is established in the following theorem.

Theorem 2. *Let $\mathcal{B}_i$ be the minimal DFA accepting the regular language L_i for $i = 1, 2$. The L^{Sep} algorithm uses at most $O((|\mathcal{B}_1| \times |\mathcal{B}_2|)^2 + (|\mathcal{B}_1| \times |\mathcal{B}_2|) \log m)$ membership queries and $4(|\mathcal{B}_1| \times |\mathcal{B}_2|) - 1$ containment queries to learn a minimal separating DFA for L_1 and L_2, where m is the length of the longest counterexample returned by the teacher.*

Proof. Let $\mathcal{C}$ be a minimal 3DFA such that $\mathcal{L}(\mathcal{C}^-) = L_1$ and $\mathcal{L}(\mathcal{C}^+) = \overline{L_2}$. The Candidate Generator takes at most $O(|\mathcal{C}|^2 + |\mathcal{C}| \log m)$ membership queries and proposes at most $|\mathcal{C}| - 1$ conjecture 3DFA's to L^{Sep}. By Lemma 2, the size of $\mathcal{C}$ is smaller than $|\mathcal{B}_1| \times |\mathcal{B}_2|$. It follows that the L^{Sep} algorithm takes $O((|\mathcal{B}_1| \times |\mathcal{B}_2|)^2 + (|\mathcal{B}_1| \times |\mathcal{B}_2|) \log m)$ membership queries and $4(|\mathcal{B}_1| \times |\mathcal{B}_2|) - 1$ containment queries (for each conjecture 3DFA, L^{Sep} uses at most 2 containment queries to check completeness and 2 containment queries to check soundness) to learn a minimal separating DFA in the worst case. □

5 Automated Compositional Verification

Here we discuss how to adapt L^{Sep} to the context of automated compositional verification. The adapted version is called "adapted L^{Sep}". We will first explain how to reduce the problem of finding the minimal assumption in assume-guarantee reasoning to the problem of finding the minimal separating automaton. Then we will show how adapted L^{Sep} handles the case in which the machine does not satisfy the property and introduce heuristics to improve the efficiency of the algorithm.

Finding the minimal assumption in assume-guarantee reasoning: We use the following assume-guarantee rule to verify if the machine composed of two components M_1 and M_2 satisfies a property P:

$$\frac{\mathcal{L}(M_2) \subseteq \mathcal{L}(A) \quad \mathcal{L}(M_1) \cap \mathcal{L}(A) \subseteq \mathcal{L}(P)}{\mathcal{L}(M_1) \cap \mathcal{L}(M_2) \subseteq \mathcal{L}(P)}$$

The rule has two premises. The second premise can be rewritten using the following steps:

$$\mathcal{L}(M_1) \cap \mathcal{L}(A) \subseteq \mathcal{L}(P) \Leftrightarrow (\mathcal{L}(M_1) \cap \mathcal{L}(A)) \cap \overline{\mathcal{L}(P)} = \emptyset \Leftrightarrow$$
$$\mathcal{L}(A) \cap (\mathcal{L}(M_1) \cap \overline{\mathcal{L}(P)}) = \emptyset \Leftrightarrow \mathcal{L}(A) \subseteq \overline{(\mathcal{L}(M_1) \cap \overline{\mathcal{L}(P)})} = \overline{\mathcal{L}(M_1)} \cup \mathcal{L}(P)$$

Therefore, one can summarize the two premises as $\mathcal{L}(M_2) \subseteq \mathcal{L}(A) \subseteq \overline{\mathcal{L}(M_1)} \cup \mathcal{L}(P)$. This immediately translates the problem of finding the minimal assumption in assume-guarantee reasoning to the problem of finding the minimal separating automaton of the two languages $\mathcal{L}(M_2)$ and $\mathcal{L}(M_1) \cap \overline{\mathcal{L}(P)}$. Therefore, if the machine satisfies the property, L^{Sep} can be used to find the minimal contextual assumption A that discharges the assume-guarantee rule[1].

The case that the machine violates the property: This can be handled by adding a check to the L^{Sep} algorithm. When a conjecture query returns a counterexample w, adapted L^{Sep} checks $w \in \mathcal{L}(M_2)$ and $w \in \mathcal{L}(M_1) \cap \overline{\mathcal{L}(P)}$ by submitting membership queries. If w passes the check, it is a string in the both $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2)$, but is not a string in $\mathcal{L}(P)$. It follows that w is a witness

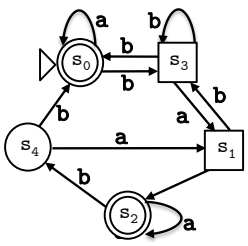
[1] The reduction was first observed by Gupta *et al.* [**?**].

that the machine violates the property. Otherwise, the L^{Sep} algorithm proceeds as usual. The correctness of the above procedure can be ensured by the following lemma:

Lemma 3. *If $\mathcal{L}(M_1) \cap \mathcal{L}(M_2) \nsubseteq \mathcal{L}(P)$, eventually the adapted L^{Sep} algorithm will find a counterexample w from a containment query such that w is in both $\mathcal{L}(M_2)$ and $\mathcal{L}(M_1) \cap \overline{\mathcal{L}(P)}$.*

Proof. See Appendix for the proof. □

Heuristics for efficiency: Minimizing a 3DFA is computationally expensive. In the context of automated compositional verification, we do not need to insist on finding a minimal solution. A heuristic algorithm that finds a small assumption with lower cost may be preferred. The adapted L^{Sep} algorithm uses the following heuristic to build a reduced DFA consistent with a 3DFA.



$\mathcal{C} = (\Sigma, S, s_0, \delta, A, R, D)$

Fig. 7. The 3DFA to be reduced

First, we use Paull and Unger's algorithm to find the sets of "maximal" compatible states[2], which are the candidates of the states in the reduced DFA. In this example (Figure 7), we have $S_1 = \{s_0, s_1\}$, $S_2 = \{s_0, s_2\}$, $S_3 = \{s_0, s_3, s_4\}$. We choose the largest set from $\{S_1, S_2, S_3\}$ that contains s_0 as the initial state of the reduced DFA. Here we take S_3. The next state of S_3 after reading the symbol a is the largest set $S' \in \{S_1, S_2, S_3\}$ that satisfies $S' \supseteq \{s' | s' = \delta(s, a), s \in S_3\} = \{s_0, s_1\}$. Here we get S_1. Note that we can always find a next state in the reduced DFA. This is because the next states (in the 3DFA) of a set of compatible states are also compatible states. Therefore, the set of the next states (in the 3DFA) is either a set of maximal compatible states or a subset of a set of maximal compatible states. The next states of any $S \in \{S_1, S_2, S_3\}$ can be found using the same procedure. The procedure terminates after the transition function of the reduced DFA is completely specified. The output of a state S is set to $+$ if there exists a state $s \in S$ such that $\theta(s) = +$, otherwise its output is set to $-$. According to our experimental result, although the adapted algorithm is not guaranteed to provide an optimal solution, it usually produces a satisfactory one and is much faster than the original version. Besides, since we do not insist on minimality, we also skip the completeness checking in the adapted version. This check takes a lot of time because the two DFA's C_i^+ and C_i^- can be large.

[2] Two states are *compatible* if applying any string to them produces no conflict output, i.e., one is accepted while the other is rejected. The states in a *set of compatible states* are pairwise compatible. A set of compatible states S is maximal if there exists no other set of compatible states S' such that $S' \supset S$.

6 Experiments

In this section, we evaluate L^{Sep} by two sets of experiments. First, we compare it with the algorithm of Gupta *et al.* [**?**] and algorithm of Grinchtein *et al.* [**?**] on a large set of randomly-generated sample problems. Secondly, we evaluate adapted L^{Sep} and compare it with other automated compositional verification algorithms on the LTSA benchmarks [**?**].

6.1 Experiment 1

All the algorithms are implemented in C++. We ran the first experiment on a Xeon 3.2GHz machine with a 3GB memory. The zChaff [**?**] tool was used to solve the SAT problems in Gupta's and Grinchtein's algorithms. For exact minimization of a 3DFA, we modified the Stamina tool [**?**] to generate the sets of compatible states and used BSOLO [**?**] to solve the binate covering problem.

We first describe the procedure for generating sample problems. Each sample problem has two DFA's B_1 and B_2 such that $\mathcal{L}(B_1) \subseteq \mathcal{L}(B_2)$. The procedure first randomly generates[3] two DFA's A_1 and A_2 such that $|A_1| = |A_2| = n$. Both use the same alphabet, which is of size m. Then the procedure builds the DFA B_1 by constructing the minimal DFA that recognizes $\mathcal{L}(A_1) \cap \mathcal{L}(A_2)$ and B_2 by constructing the minimal DFA that recognizes $\mathcal{L}(A_1) \cup \mathcal{L}(A_2)$. This procedure has two important properties: (1) the difference between $|B_1|$ and $|B_2|$ is small; (2) there exists a (relatively) small separating DFA for B_1 and B_2. We use two different alphabet sizes ($m = 4$ and $m = 8$) and sixteen different source DFA sizes (from $n = 4$ to $n = 12$). For each pair (n, m), we randomly generate a set of 100 different sample problems (we eliminate duplications). We also drop trivial cases ($|B_1| = 1$ or $|B_2| = 1$). Table 1 shows the results. We set a timeout of 4000 seconds (for each set of 100 sample problems). If the algorithm does not solve any problem in a set of 100 problems within the timeout period, we mark it as >4000. The time spent on unsuccessful tasks is included in the total processing time.

6.2 Experiment 2

We evaluate adapted L^{Sep} on the LTSA benchmarks [**?**] for automated compositional verification. The LTSA tool [**?**] is used to translate the benchmark problems to labeled transition systems in Aldebaran format [**?**], which is one of the supported input formats of our tool.

We compare the adapted L^{Sep} algorithm with the algorithms of Gupta *et al.*, Grinchtein *et al.*, and Cobleigh *et al.* [**?**]. We implemented all of those algorithms, including the heuristic algorithm for minimizing a 3DFA. We do not consider optimization techniques such as alphabet refinement [**?**,**?**]. This is fair because

[3] For each state s in A_1 (respectively A_2) and for each symbol a, a destination state s' in A_1 (respectively A_2) is picked at random and a transition $\delta(s, a) = s'$ is established. Each state has a 50% chance of being selected as a final state.

(n,m)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)	(4,9)	(4,10)	(4,11)	(4,12)
Algorithms	Average execution time								
L^{Sep}	0.04	0.16	0.4	0.84	1.54	2.5	4.3	6.8	10.9
Gupta [**?**]	6.6	58.7	266.7	431.5	1308.8	>4000	>4000	>4000	>4000
Grinchtein [**?**]	51.8	139	255.6	514.7	>4000	>4000	>4000	>4000	>4000
(n,m)	(8,4)	(8,5)	(8,6)	(8,7)	(8,8)	(8,9)	(8,10)	(8,11)	(8,12)
Algorithms	Average execution time								
L^{Sep}	0.15	0.44	0.96	2.1	3.7	6.4	11	17.8	26.9
Gupta [**?**]	96.2	625.9	972.3	>4000	>4000	>4000	>4000	>4000	>4000
Grinchtein [**?**]	813.4	>4000	>4000	>4000	>4000	>4000	>4000	>4000	>4000

Unit: Second

Table 1. Comparison of the Three Algorithms

such techniques can also be easily adapted to L^{Sep}. The experimental results are shown in Table 2. We use the decomposition suggested by the benchmarks to build components M_1 and M_2. We set a timeout of 10000 seconds. Actually we checked all the 89 LTSA benchmark problems (of size 2 ,3 and 4)[4]. In the table we do not list results with minimal contextual assumption of size 1 (10 cases) and those in which no algorithms finished within timeout period (30 cases).

The adapted L^{Sep} algorithm performs better than the all other algorithms in 30 among the 49 problems. The algorithm of Cobleigh *et al.* wins 14 problems. However, in 8 of the 14 cases (S23-2, S24-2, S24-3, S24-4, S26-2, S27-2, S29-2, S30-2), their algorithm finds an assumption with size almost the same to $|\overline{M_1} \times P|$. In those cases, there is no hope to defeat monolithic verification. In contrast, our algorithm scales better than monolithic verification in many problem sets. For example, in S1, S19, S22, and S32, the execution time of adapted L^{Sep} grows much slower than monolithic verification. In S1 and S22, we can see that the adapted L^{Sep} takes more execution time than monolithic verification when the size is 2, but its performance surpasses monolithic verification when the size becomes 4.

7 Conclusion

We proposed an efficient algorithm called L^{Sep} that learns and generates a minimal separating DFA of two given disjoint regular languages. The performance of L^{Sep} was shown analytically and experimentally to be superior to those of existing algorithms. We attributed the better performance to the facts that our algorithm utilizes membership queries to accelerate learning and has a more compact representation of the samples collected from the queries. We subsequently detailed how L^{Sep} can be adapted for automated compositional verification and showed by experiments that the resulting solution surpasses other approaches in a majority of the LTSA benchmark problems.

For further work, it will be interesting to see adaptation of L^{Sep} for other applications, such as inferring network invariants of parameterized systems, and

[4] For most of the problems of larger size, the LTSA tool fails to translate it to Aldebaran format and reports an out of memory message.

	L^{Sep}		Cobleigh		Gupta		Grinchtein		$\lvert M_2\rvert$	$\lvert\overline{M_1}\times P\rvert$	MO
	Time	|A|	Time	|A|	Time	|A|	Time	|A|			Time
S1-2	**0.1**	3	170	74	32	3	2039	3	45	80	0.08
S1-3	**0.4**	3	-	-	109	3	-	-	82	848	0.7
S1-4	**1.6**	3	-	-	219	3	-	-	138	4046	4.2
S2-2	508	7	**89**	52	-	-	-	-	39	89	0.08
S2-3	-	-	**1010**	93	-	-	-	-	423	142	0.7
S2-4	-	-	**7063**	152	-	-	-	-	2022	210	4
S3-2	**1.9**	3	51	57	140	3	-	-	39	100	0.09
S3-3	**13**	3	601	110	551	3	-	-	423	164	0.8
S3-4	**55**	3	4916	189	1639	3	-	-	2022	69	4.2
S4-2	**5.8**	3	21	35	90	3	203	3	39	87	0.09
S4-3	**20.8**	3	1109	103	433	3	2983	3	423	140	0.75
S4-4	**44.9**	3	6390	156	793	3	5157	3	2022	208	4.1
S5-2	**940**	64	998	127	-	-	-	-	45	133	0.08
S7-2	362	39	**48**	46	-	-	-	-	39	104	0.09
S7-3	-	-	**405**	76	-	-	-	-	423	168	0.9
S7-4	-	-	**3236**	123	-	-	-	-	2022	256	4.1
S9-2	**1345**	52	4448	240	-	-	-	-	45	251	0.09
S10-2	6442	18	-	-	**196**	3	-	-	151	309	0.8
S10-3	5347	22	-	-	**601**	3	-	-	327	3369	6.1
S10-4	-	-	-	-	**1214**	3	-	-	658	16680	33
S11-2	**6533**	82	-	-	-	-	-	-	151	515	0.8
S12-2	**36**	4	1654	162	-	-	-	-	151	273	0.81
S12-3	**133**	4	-	-	-	-	-	-	327	2808	6.6
S12-4	**450**	4	-	-	-	-	-	-	658	13348	33
S15-2	**1477**	88	-	-	5992	3	-	-	151	309	0.77
S15-3	5840	5	-	-	**4006**	3	-	-	327	3369	5.9
S15-4	-	-	-	-	**6880**	3	-	-	658	16680	33
S19-2	**5.8**	3	-	-	266	3	-	-	234	544	0.26
S19-3	**13**	3	-	-	1392	3	-	-	962	5467	2.9
S19-4	**69**	3	-	-	7636	3	-	-	2746	52852	35
S21-3	**45**	3	-	-	4558	3	-	-	962	5394	2.9
S21-4	**718**	3	-	-	3839	3	-	-	2746	51225	34.8
S22-2	**0.6**	3	8	25	12	3	-	-	900	30	0.33
S22-3	**2.3**	3	1242	193	54	3	-	-	7083	264	4.6
S22-4	**11**	3	-	-	170	3	-	-	30936	2190	33
S23-2	92	9	**8.9**	37	-	-	-	-	50	40	0.12
S24-2	1.2	6	**0.2**	12	1.2	3	226	3	13	14	0.003
S24-3	5.1	6	**0.33**	12	-	-	-	-	48	14	0.02
S24-4	18	6	**0.63**	12	-	-	-	-	157	14	0.12
S25-2	**1156**	5	3050	257	-	-	-	-	41	260	0.11
S26-2	512	38	**239**	121	-	-	-	-	65	123	0.1
S27-2	848	46	**830**	193	-	-	-	-	41	204	0.1
S28-2	**755**	46	757	185	-	-	-	-	41	188	0.1
S29-2	926	21	**891**	193	-	-	-	-	41	195	0.1
S30-2	1083	24	**986**	193	-	-	-	-	41	195	0.1
S31-2	**204**	5	274	121	4975	3	-	-	65	165	0.1
S32-2	**9.9**	3	646	193	121	3	-	-	41	261	0.1
S32-3	**44**	3	-	-	-	-	-	-	1178	4806	2.6
S32-4	**886**	3	-	-	-	-	-	-	289	117511	382

Table 2. Experimental results of LTSA Benchmarks. The "L^{Sep}" column is the result of adapted L^{Sep}. "Time" is the execution time in seconds and $|A|$ is the size of contextual assumption found by the algorithm. "Cobleigh" and "Gupta" give results from **[?]** and **[?]**, respectively. The two columns $|M_2|$ and $|\overline{M_1}\times P|$ are the sizes of DFA M_2 and the product of DFA's $\overline{M_1}$ and P. The sizes are slightly different from the original version because we determinized them. We think the size after determinization can better reflect the difficultly of a benchmark problem. Furthermore, we swapped M_1 and M_2; they check $\mathcal{L}(M_1) \subseteq \mathcal{L}(A)$ and $\mathcal{L}(M_2) \cap \mathcal{L}(A) \subseteq \mathcal{L}(P)$ in their experiments. We swap them because in their original arrangement, a large portion of the cases have an assumption of size 1. MO is the result for monolithic verification. The symbol "-" indicates that the algorithm does not finish within timeout period and did not find a contextual assumption. For each row, we use Sn-m to denote experiment subject n with m instances.

to evaluate the performance of resulting solutions. Given that L^{Sep} is a better learning algorithm, we hope that other applications will also benefit from it.

A Appendix

A.1 Candidate Generator

The Candidate Generator is essentially an extension of L^* to allow don't care values. The most important theorem used by L^* is the Myhill-Nerode theorem. In the following paragraphs, we will first extend Myhill-Nerode theorem to allow don't care values. It then follows by a detail description of how Candidate Generator works and its correctness proofs.

Given a 3DFA $C = (\Sigma, S, s_0, \delta, Acc, Rej, Dont)$, we define the equivalence relation $\equiv_C$ as follows:

$$u \equiv_C v \text{ iff } \forall w \in \Sigma^* \begin{cases} \delta(s_0, uw) \in Acc & \rightarrow \delta(s_0, vw) \in Acc \\ \delta(s_0, uw) \in Rej & \rightarrow \delta(s_0, vw) \in Rej \\ \delta(s_0, uw) \in Dont & \rightarrow \delta(s_0, vw) \in Dont \end{cases}$$

This equivalence relation partitions Σ^* into equivalence classes. We use the notation $[u]_C$ to denote the equivalence class that contains u.

Lemma 4. *The equivalence relation $\equiv_C$ has a finite index.*

Proof. Let u and v be two finite strings that lead C to the same state, *i.e.*, $\delta(s_0, u) = \delta(s_0, v)$. It follows that, for all strings w, uw and vw lead C to the same state. Hence we have $\forall w \in \Sigma^*.(\delta(s_0, uw) \in Acc \rightarrow \delta(s_0, vw) \in Acc) \wedge (\delta(s_0, uw) \in Rej \rightarrow \delta(s_0, vw) \in Rej) \wedge (\delta(s_0, uw) \in Dont \rightarrow \delta(s_0, vw) \in Dont)$. Then, $u \equiv_C v$ can be inferred. Because C has a finite number of states and the number of classes of $\equiv_C$ is not larger than the number of states in C, the equivalence relation $\equiv_C$ has a finite index can be inferred. □

Because the number of equivalence classes is finite (Lemma 4), we can build a 3DFA $C' = (\Sigma, S', s'_0, \delta', Acc', Rej', Dont')$ such that each state in C' is one-to-one mapped to an equivalence class of the relation $\equiv_C$ as follows:

- Each state is mapped to an equivalence class.
- The initial state s'_0 is mapped to the class $[\lambda]_{\equiv_C}$.
- $\delta'(s_1, a) = s_2$ if s_1 is mapped to the class $[u]_{\equiv_C}$, s_2 is mapped to the class $[v]_{\equiv_C}$, and $ua \in [v]_{\equiv_C}$.
- A state $s \in Acc'$ iff s is mapped to the class $[u]_{\equiv_C}$ and $\delta(s_0, u) \in Acc$.
- A state $s \in Rej'$ iff s is mapped to the class $[u]_{\equiv_C}$ and $\delta(s_0, u) \in Rej$.
- A state $s \in Dont'$ iff s is mapped to the class $[u]_{\equiv_C}$ and $\delta(s_0, u) \in Dont$.

Lemma 5. *The 3DFA C' is the minimal 3DFA that is equivalent to C.*

Proof. Assume that there exists a 3DFA $\widehat{C}$ smaller than C' and equivalent to C. Because $\widehat{C}$ has fewer states than C', there exist two strings u and v that lead C' to different states, but lead $\widehat{C}$ to the same state. Because u and v lead C' to different states, they are in difference equivalence classes. It follows that there exists a string w such that uw and vw are different types (accepting, rejecting, or don't care) of strings in C. Assume uw is an accepting string in C; then vw

cannot be an accepting string in C. If uw leads $\widehat{C}$ to an accepting state, then vw is a counterexample for the assumption that $\widehat{C}$ is equivalent to C. Otherwise, uw is a counterexample for the assumption that $\widehat{C}$ is equivalent to C. The case where uw is a rejecting string in C and the case where uw is a don't care string in C lead to the same result. The existence of $\widehat{C}$ is rejected by the results of the case analysis. □

The Candidate Generator In the following paragraphs, we describe the Candidate Generator which is an extension of the learning algorithm L^* [**?**,**?**]. Let L_1 and L_2 be two disjoint regular languages, C is a 3DFA that accepts every string in L_1 and rejects every string in L_2. The goal of this algorithm is to learn the equivalence classes of $\equiv_C$ and then convert them to a 3DFA. By Lemma 5, the 3DFA converted from $\equiv_C$ is the minimal 3DFA equivalent to C. Similar to L^*, the Candidate Generator assumes the existence of a *teacher* that knows C and answers queries about it. If such a teacher is provided, this Candidate Generator is guaranteed to generate a *minimal* 3DFA equivalent to C using a *polynomial* number of queries (polynomial in the size of the minimal 3DFA and the length of the longest counter example returned by the teacher).

This extension is based on the revised version of L^* proposed by Rivest and Schapire [**?**]. Assume the target 3DFA $C = (\Sigma, S, s_0, \delta, Acc, Rej, Dont)$. The Candidate Generator assumes there exists a *teacher* that answers the following two types of queries:

1. For a **membership query** on a string w, the teacher answers which type of string w (Accepting, Rejecting, or Don't care) is in the target 3DFA C.
2. For an **equivalence query** on a 3DFA C', the teacher checks whether or not C' equals to the target 3DFA C. If they are equivalent, a YES answer is returned to the Candidate Generator. Otherwise, a NO answer accompanied by a counterexample that witnesses the inequality of C and C' is returned. Note that in the main text, we interpret equivalence queries as proposing conjectures to L^{Sep} If C' is is not sound and complete with respect to L_1 and L_2, i.e., C' does not equal C, a counterexample will be returned from L^{Sep} unless a minimal separating DFA is found.

The Candidate Generator collects information from the teacher by proposing queries and uses an *observation table* to record the query results:

Definition 2. *An observation table is a tuple $\langle S, E, T\rangle$ where S is a set of prefix-closed strings in Σ^* such that each string in S represents an equivalence class of $\equiv_C$; E is a set of strings in Σ^* such that each string in E is a distinguishing string; and $T : (S \cup S\Sigma) \times E \longrightarrow \{T, F, D\}$ is defined as*

$$T(\alpha, \sigma) = \begin{cases} T & \text{if } \delta(s_0, \alpha\sigma) \in Acc, \text{ i.e., } \alpha\sigma \text{ is accepted by } C \\ F & \text{if } \delta(s_0, \alpha\sigma) \in Rej, \text{ i.e., } \alpha\sigma \text{ is rejected by } C \\ D & \text{if } \delta(s_0, \alpha\sigma) \in Dont, \text{ i.e., } \alpha\sigma \text{ is a don't care string in } C. \end{cases}$$

An observation table is *closed* if for every string $s' \in S\Sigma$, there exists a string $s \in S$ such that $T(s, \bullet) = T(s', \bullet)$. The notation $T(s, \bullet)$ indicates the row of the table that starts with s. We have $T(s, \bullet) = T(s', \bullet)$ if and only if for all string w in E, $T(s, w) = T(s', w)$.

The procedure of the Candidate Generator for learning a 3DFA is similar to the Rivest's version of L^*. The Candidate Generator completes the observation table by asking membership queries for each pair of strings $(\alpha, \sigma) \in (S \cup S\Sigma) \times E$; an "Accepted" response sets $T(\alpha, \sigma)$ to "T", a "Rejected" response sets $T(\alpha, \sigma)$ to "F", and a "Don't care" response sets $T(\alpha, \sigma)$ to "D".

Algorithm 1: *Make Close*

```
repeat
    isClosed ← true;
    if there exists s ∈ S and a ∈ Σ such that T(sa, •) ≠ T(s', •) for all s' ∈ S
    then
        add sa to S;
        isClosed ← false;
        extend T to (S ∪ SΣ)E using membership queries;
    end
until isClosed;
```

The observation table is expanded to a *closed* table by Algorithm 1. Once the table is closed, a conjecture 3DFA $C' = (\Sigma, S', s'_\lambda, \delta', Acc', Rej', Dont')$ is constructed as follows:

- $S' = \{s'_w | w \in S\}$.
- The initial state is s'_λ.
- For all $a \in \Sigma$, $\delta'(s'_w, a) = s'_{w'}$ iff $T(wa, \bullet) = T(w', \bullet)$.
- A state $s'_w \in Acc'$ iff $T(w, \lambda) = T$.
- A state $s'_w \in Rej'$ iff $T(w, \lambda) = F$.
- A state $s'_w \in Dont'$ iff $T(w, \lambda) = D$.

Define w as the *representative string* of the state s_w.

The Candidate Generator then submits an equivalence query on C'. If a YES answer is returned, C' is a correct and the algorithm terminates. Otherwise, a distinguishing string is extracted from the counterexample ce returned from the teacher.

Let $ce = x_i y_i$, where $|x_i| = i$. Define string z_i as the representative string of the state $\delta'(s'_0, x_i)$. The function $MQ_C(i)$ denotes the membership query result for the string $z_i y_i$ in the 3DFA C. Algorithm 2 extracts a distinguishing string from ce. It uses a binary search to find an integer ptr, where $0 \leq \mathsf{ptr} \leq |ce|$, such that $MQ_C(\mathsf{ptr}) \neq MQ_C(\mathsf{ptr} + 1)$. Let b be the first character of the string y_{ptr}. By definition, $y_{\mathsf{ptr}} = b y_{\mathsf{ptr}+1}$.

We have that (1) the two strings $z_{\mathsf{ptr}} b$ and $z_{\mathsf{ptr}+1}$ lead C' to the state $s_{z_{\mathsf{ptr}+1}}$, i.e., they are in the same equivalence class of $\equiv'_C$. However, (2) they are not in

the same equivalence class of $\equiv_C$, and because the membership query results of $z_{\mathsf{ptr}}by_{\mathsf{ptr}+1}$ and $z_{\mathsf{ptr}+1}y_{\mathsf{ptr}+1}$ are different in C, i.e., the string $y_{\mathsf{ptr}+1}$ can distinguish them. Adding the string $y_{\mathsf{ptr}+1}$ to the E set of the observation separates the string $z_{\mathsf{ptr}}b$ from the equivalence class $[z_{\mathsf{ptr}+1}]_{C'}$ and introduces at least one new state to the next conjecture. The string $y_{\mathsf{ptr}+1}$ is thus a distinguishing string for the Candidate Generator.

Each time when E is expanded, Candidate Generator repeats the procedure to complete the table propose a new conjecture. The algorithm only terminates when it finds a 3DFA that equals C.

Algorithm 2: *Find an Experiment*

```
Input: ce : a counterexample
begin ← 0;
end ← |ce|;
ptr ← (begin + end)/2;
repeat
    if MQ_C(ptr) ≠ MQ_C(end) then
        begin ← ptr;
        ptr ← (begin + end)/2;
    else
        end ← ptr;
        ptr ← (begin + end)/2;
    end
until MQ_C(ptr) ≠ MQ_C(ptr + 1);
return y_{ptr+1};
```

Complexity: If two strings are in the same equivalence class of $\equiv_C$, they cannot be distinguished by any suffix. It follows that, at most, only one of them can belong to the S set of the table. Thus, the size of S is always smaller or equal to the size of $\equiv_C$. Let n be the size of $\equiv_C$, *i.e.*, the size of the minimal 3DFA that equals C. Since each incorrect conjecture introduces a new distinguishing experiment and thus increases the size of S, the algorithm is guaranteed to guess at most $n-1$ incorrect conjectures. That is, the algorithm is guaranteed to find a minimal 3DFA that equals C using at most $n-1$ equivalence queries. Let m be the length of the longest counterexample returned from an equivalence query. For each equivalence query on an incorrect conjecture, we need at most $\log_2 m$ membership queries to find a distinguishing experiment. Together with the fact that the Candidate Generator needs at most $n-1$ equivalence queries to find the target 3DFA, it submits at most $(n-1)log_2 m$ membership queries to find distinguishing experiments and the size of the E set is at most n. Additionally, since the sizes of S and E are always smaller than or equal to n, at most $(n + |\Sigma|)n$ membership queries are required to fill up the observation table. Thus, the Candidate Generator takes $O(n^2 + n \log m)$ membership queries and $n-1$ equivalence queries to learn a minimal 3DFA that is equivalent C.

A.2 The Query Complexity of the Algorithm of Gupta *et al.*

The worst query complexity of the Algorithm of Gupta *et al.* can be established as follows: Let L_1 and L_2 be two languages such that $L_1 = \overline{L_2}$, and the Algorithm of Gupta *et al.* is applied to find a minimal separating DFA for this two languages.

Assume that strings are ordered according to their lengths and strings of same length are ordered lexicographically. Here we show a case that the Algorithm of Gupta *et al.* found a correct conjecture when all strings of length $\leq n$ are returned from the teacher.

Initially, assume that λ is the sample set and is labeled $+$. The Algorithm of Gupta *et al.* will generate a one-state automaton that accepts strings in Σ^* as the first conjecture C_1 and submit a containment query on C_1. Let s_1 be the smallest string such that s_1 is not in the current sample set of the algorithm of Gupta *et al.*. We define that $s_1 \notin L_1$ and assume the teacher always return the smallest counterexample. Thus, the teacher will return s_1 as a negative counterexample. The algorithm of Gupta *et al.* then builds that next conjecture C_2 that is consistent with the current sample set and submit a conjecture query on C_2. Let s_2 be the smallest string such that s_2 is not in the current sample set. If $s_2 \in \mathcal{L}(C_2)$, then we define $s_2 \notin {}_1$, otherwise we define $s_2 \in {}_1$. Under such a definition, s_2 will be returned by the teacher as the counterexample for the 2nd iteration. The procedure will repeat until all strings of length n are in the sample set. The algorithm will propose a correct conjecture at the iteration that all strings of length n are in the sample set. Formally, we define the language L_1 as follows: the string $s_i \in \mathcal{L}(C_i)$ if and only if $s_i \notin L_1$, where $1 \leq i \leq$ (the number of strings with length $\leq n$). In this example, the algorithm of Gupta *et al.* uses an exponential (in n) number of containment queries to find a conjecture separating DFA.

A.3 Proofs of Lemmas

Lemma 2. *The DFA A is a minimal consistent DFA of the 3DFA C.*

Proof. **(A is a DFA consistent with C)** For each string u accepted by C, the last symbol in the sequence produced by inputting u to M is "+". Because M' is consistent with M, the last symbol in the sequence produced by inputting u to M is also "+".. It follows that u is accepted by A. If u is rejected by C, last symbol in the sequence produced by inputting u to M or M' is "$-$". It follows that u is rejected by A. Hence, we proved that A is a DFA consistent with C. **(A is a minimal DFA consistent with C)** Assume that there exists a DFA $\hat{A} = (\Sigma, \hat{S}, \hat{s_0}, \hat{\delta}, \widehat{Acc})$ such that $\hat{A}$ is consistent with C and $\hat{A}$ is smaller than A. Define an FSM $\hat{M} = (\Sigma, \{+, -\}, \hat{S}, \hat{s_0}, \hat{\delta}, \hat{\theta})$, where $\hat{s} \in \widehat{Acc} \Rightarrow \hat{\theta}(\hat{s}) = +$ and $\hat{s} \notin \widehat{Acc} \Rightarrow \hat{\theta}(\hat{s}) = -$. For every string u, if "+" (respectively, "$-$") is the last symbol in the sequence produced by inputting u to M, u is accepted by C and hence u is also accepted by $\hat{A}$. It follows that the last symbol in the sequence produced by inputting u to $\hat{M}$ is "+" (respectively, "$-$"). Therefore, we can conclude that $\hat{M}$ *is consistent with* M. From the fact that $\hat{A}$ is isomorphic to

$\hat{M}$ and $\hat{A}$ is smaller than A, which is isomorphic to M', we can infer that $\hat{M}$ is smaller than M', which is a minimal FSM consistent with M, which contradicts our assumption and thus proved that A is a minimal DFA consistent with C. □

Lemma 4. *If $\mathcal{L}(M_1) \cap \mathcal{L}(M_2) \nsubseteq \mathcal{L}(P)$, eventually adapted L^{Sep} will get a counterexample w from a containment query such that w is in both $\mathcal{L}(M_2)$ and $\overline{\mathcal{L}(P)} \cap \mathcal{L}(M_1)$.*

Proof. Let $L_1 = \mathcal{L}(M_2)$ and $L_2 = \overline{\mathcal{L}(P)} \cap \mathcal{L}(M_1)$. The string w is in both $\mathcal{L}(M_2)$ and $\overline{\mathcal{L}(P)} \cap \mathcal{L}(M_1)$ if and only if w is in $\mathcal{L}(M_2) \cap (\overline{\mathcal{L}(P)} \cap \mathcal{L}(M_1)) = L_1 \cap L_2$.

The formula $\mathcal{L}(M_1) \cap \mathcal{L}(M_2) \nsubseteq \mathcal{L}(P)$ can be written as follows:

$$\mathcal{L}(M_1) \cap \mathcal{L}(M_2) \nsubseteq \mathcal{L}(P) \Leftrightarrow (\mathcal{L}(M_1) \cap \mathcal{L}(M_2)) \cap \overline{\mathcal{L}(P)} \neq \emptyset \Leftrightarrow$$
$$\mathcal{L}(M_2) \cap (\overline{\mathcal{L}(P)} \cap \mathcal{L}(M_1)) \neq \emptyset \Leftrightarrow L_1 \cap L_2 \neq \emptyset$$

It follows that L^{Sep} can never find a separating DFA for L_1 and L_2 and then terminate. Let L_1^- be the language $L_1 \cap L_2$. Assume that L^{Sep} never propose a membership query on a string w such that w is in $L_1 \cap \overline{L_2}$. Under such an assumption, L^{Sep} will behave the same as the two languages to separate are L_1^- and L_2. Because L_1^- and L_2 are disjoint, L^{Sep} will eventually find a DFA A such that $L_1^- \subseteq \mathcal{L}(A) \subseteq \overline{L_2}$.

The algorithm will then check $L_1 \subseteq \mathcal{L}(A)$ by submitting a containment query. Because (1) $\mathcal{L}(A) \subseteq \overline{L_2}$ and (2) $L_1 \nsubseteq \overline{L_2}$, it follows that $L_1 \nsubseteq \mathcal{L}(A)$. The teacher will then return a counterexample $w \in L_1$, but $w \notin \mathcal{L}(A)$. Because $\mathcal{L}(A) \subseteq \overline{L_2}$, we can then infer $w \notin \overline{L_2} \Leftrightarrow w \in L_2$. Therefore, $w \in L_1 \cap L_2$ and the lemma is proved. □