Learning Minimal Separating DFA's for Compositional Verification[⋆]

Yu-Fang Chen[1], Azadeh Farzan[2], Edmund M. Clarke[3], Yih-Kuen Tsay[1], and Bow-Yaw Wang[4]

[1]National Taiwan University [2]University of Toronto [3]Carnegie Mellon University [4]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 describe 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 [8] 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:

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

⋆ This research was sponsored by the iCAST project of the National Science Council, Taiwan, under the grants no. NSC96-3114-P-001-002-Y and no. NSC97-2745-P-001-001, GSRC (University of California) under contract no. SA423679952, National Science Foundation under contracts no. CCF0429120, no. CNS0411152, and no. CCF0541245, Semiconductor Research Corporation under contract no. 2005TJ1366 and no. 2005TJ1860, and Air Force (University of Vanderbilt) under contract no. 1872753

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 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 $\mathcal{A}$ and (2) component M_2 satisfies the contextual assumption $\mathcal{A}$.

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 [4, 10] based on the machine learning algorithm L^* [2, 17]. Following this line of research, there have been results for symbolic implementations [1, 18], various optimization techniques [12, 6], an extension to liveness properties [11], performance evaluation [9], and applications to problems such as component substitutability analysis [5]. However, all of the above suffer from the same problem: they do not guarantee finding a small assumption even if one exists. Though minimality of the assumption does not ensure better performance, we will show in this paper that it helps most of the time.

The problem of finding a minimal assumption for compositional verification can be reduced to the problem of finding a minimal separating DFA (deterministic finite automaton) of two disjoint regular languages [14]. A DFA $\mathcal{A}$ *separates* two disjoint languages L_1 and L_2 if its language $\mathcal{L}(\mathcal{A})$ contains L_1 and is disjoint from L_2 ($L_1 \subseteq \mathcal{L}(\mathcal{A})$ and $\mathcal{L}(\mathcal{A}) \cap L_2 = \emptyset$). The DFA $\mathcal{A}$ is *minimal* if it has the least number of states among all separating DFA's. Several approaches [14, 16, 13] 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.* [14] 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}, that finds the aforementioned minimal separating DFA. The query complexity of our algorithm is quadratic in the product of the sizes of the two minimal DFA's for the two input languages. 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 [9]. 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. For example, Grinchtein *et al.* [13] used such an algorithm as the basis for *learning network invariants of parameterized systems*. Although we only discuss the application of L^{Sep} to automated compositional verification in this paper, the algorithm can certainly be adapted for other applications as well.

2 Preliminaries

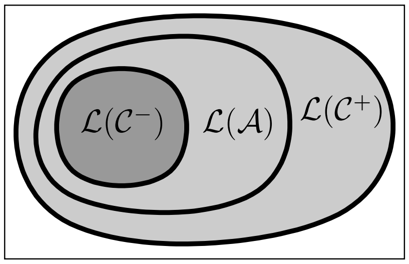
An *alphabet* Σ is a finite set. A finite *string* over Σ is a finite sequence of elements from Σ. The empty string is represented by λ. The set of all finite strings over Σ is denoted by Σ^*, and Σ^+ is the set of all nonempty finite strings over Σ (so, $\Sigma^+ = \Sigma^* \backslash \{\lambda\}$). The length of string u is denoted by $|u|$ and $|\lambda| = 0$. 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 $u \in S$.

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 function, 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) $\mathcal{C}$ *is a tuple* $(\Sigma, S, s_0, \delta, Acc, Rej, Dont)$*, where* Σ*,* S*,* s_0*, and* δ *are as defined in a DFA.* S *is partitioned into three disjoint sets* Acc*,* Rej*, and* $Dont$*.* Acc *is the set of accepting states,* Rej *is the set of rejecting states, and* $Dont$ *is the set of don't care states.*

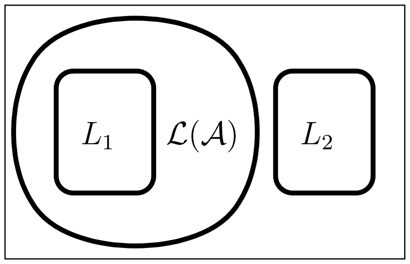
For a 3DFA $\mathcal{C} = (\Sigma, S, s_0, \delta, Acc, Rej, Dont)$, a string u is *accepted* if $\delta(s_0, u) \in Acc$, is *rejected* if $\delta(s_0, u) \in Rej$, and is a *don't care* string if $\delta(s_0, u) \in Dont$. Let $\mathcal{C}^+$ denote the DFA $(\Sigma, S, s_0, \delta, Acc \cup Dont)$, where all don't care states become accepting states, and $\mathcal{C}^-$ denote 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 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 $\mathcal{C}$ rejects. It follows that $\mathcal{A}$ accepts strings in $\mathcal{L}(\mathcal{C}^-)$ and rejects those in $\overline{\mathcal{L}(\mathcal{C}^+)}$, or equivalently, $\mathcal{L}(\mathcal{C}^-) \subseteq \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{C}^+)$. A *minimal consistent* DFA of $\mathcal{C}$ is a DFA $\mathcal{A}$ which is consistent with $\mathcal{C}$ and has the



$\mathcal{L}(\mathcal{C}^-)$
$\mathcal{L}(\mathcal{A})$
$\mathcal{L}(\mathcal{C}^+)$

(a) A DFA $\mathcal{A}$ consistent with a 3DFA $\mathcal{C}$



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

Fig. 1. Consistent and Separating DFA's

least number of states among all DFA's consistent with $\mathcal{C}$. Figure 1(a) illustrates a DFA $\mathcal{A}$ consistent with a 3DFA $\mathcal{C}$. In the figure, the bounding box is the set of all finite strings Σ^*. The dark shaded area represents $\mathcal{L}(\mathcal{C}^-)$. The union of the dark shaded area and the light shaded area represents $\mathcal{L}(\mathcal{C}^+)$. The DFA $\mathcal{A}$ is consistent with $\mathcal{C}$ as it accepts all strings in $\mathcal{L}(\mathcal{C}^-)$ and rejects those not in $\mathcal{L}(\mathcal{C}^+)$.

Given two disjoint regular languages L_1 and L_2, a *separating DFA* $\mathcal{A}$ for L_1 and L_2 satisfies $L_1 \subseteq \mathcal{L}(\mathcal{A})$ and $\mathcal{L}(\mathcal{A}) \cap L_2 = \emptyset$. It follows that $\mathcal{A}$ accepts all strings in L_1 and rejects those in L_2, or equivalently, $L_1 \subseteq \mathcal{L}(\mathcal{A}) \subseteq \overline{L_2}$. We say a DFA $\mathcal{A}$ *separates* L_1 and L_2 if and only if $\mathcal{A}$ is a separating DFA for L_1 and L_2. A separating DFA is *minimal* if it has the least number of states among all separating DFA's for L_1 and L_2. Figure 1(b) shows a separating DFA $\mathcal{A}$ for 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. When the context is clear, we abbreviate "sound with respect to L_1 and L_2" simply as "sound". Figure 2(a) illustrates the condition when $\mathcal{C}$ is sound with respect to L_1 and L_2. Both $L_1 \subseteq \mathcal{L}(\mathcal{C}^-)$ and $\mathcal{L}(\mathcal{C}^+) \subseteq \overline{L_2}$ are true in this figure. Any DFA consistent with $\mathcal{C}$ accepts strings in $\mathcal{L}(\mathcal{C}^-)$ (the dark area) and possibly some strings in the light shaded area. Hence it accepts all strings in L_1 but none in L_2, i.e., it separates L_1 and L_2. Therefore, $\mathcal{C}$ is sound. Figure 2(c) illustrates the case that $\mathcal{C}$ is unsound. We can show that either $L_1 \nsubseteq \mathcal{L}(\mathcal{C}^-)$ or $\mathcal{L}(\mathcal{C}^+) \nsubseteq \overline{L_2}$ implies $\mathcal{C}$ is unsound. Assuming that we have $L_1 \nsubseteq \mathcal{L}(\mathcal{C}^-)$. It follows that there exists some string $u \in L_1$ that satisfies $u \notin \mathcal{L}(\mathcal{C}^-)$. The DFA $\mathcal{A}$ that recognizes $\mathcal{L}(\mathcal{C}^-)$ (the dark area) is consistent with $\mathcal{C}$. However, $\mathcal{A}$ is not a separating DFA for L_1 and L_2 because it rejects u, a string in L_1. We can then conclude that $\mathcal{C}$ is unsound. The case that $\mathcal{L}(\mathcal{C}^+) \nsubseteq \overline{L_2}$ can be shown to be unsound by a similar argument.

A 3DFA $\mathcal{C}$ is *complete with respect to* L_1 *and* L_2 if any separating DFA for L_1 and L_2 is consistent with $\mathcal{C}$. Again, when the context is clear, we abbreviate "complete with respect to L_1 and L_2" as "complete". Figure 2(b) shows the situation when $\mathcal{C}$ is complete for L_1 and L_2. Any separating DFA for L_1 and L_2 accepts all strings in L_1 but none in L_2. Hence it accepts strings in $\mathcal{L}(\mathcal{C}^-)$ (the

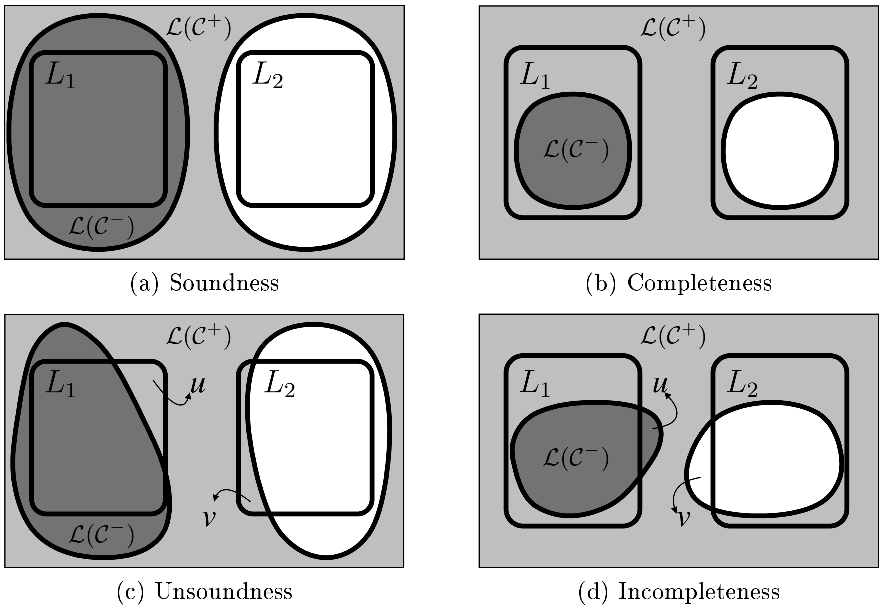


Fig. 2. Soundness and Completeness of a 3DFA $\mathcal{C}$

dark area) and possibly those in the light shaded area, i.e., it is consistent with $\mathcal{C}$. Therefore, $\mathcal{C}$ is complete. Figure 2(d) illustrates the case that $\mathcal{C}$ is incomplete. We can show that either $\mathcal{L}(\mathcal{C}^-) \not\subseteq L_1$ or $\overline{L_2} \not\subseteq \mathcal{L}(\mathcal{C}^+)$ implies $\mathcal{C}$ is incomplete. Assuming that we have $\mathcal{L}(\mathcal{C}^-) \not\subseteq L_1$. It follows that there exists some string $u \in \mathcal{L}(\mathcal{C}^-)$ that satisfies $u \notin L_1$. The DFA $\mathcal{A}$ that recognizes L_1 is a separating DFA for L_1 and L_2. However, $\mathcal{A}$ is not consistent with $\mathcal{C}$ because $\mathcal{A}$ rejects u, a string in $\mathcal{L}(\mathcal{C}^-)$. We can then conclude that $\mathcal{C}$ is incomplete. The case that $\overline{L_2} \not\subseteq \mathcal{L}(\mathcal{C}^+)$ can be shown to be incomplete by a similar argument.

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

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

3 Overview of Learning a Minimal Separating DFA

Given two disjoint regular languages L_1 and L_2, our task is to find a minimal DFA $\mathcal{A}$ that separates L_1 and L_2, namely $L_1 \subseteq \mathcal{L}(\mathcal{A}) \subseteq \overline{L_2}$. Our key idea is to use a 3DFA as a succinct representation for the samples collected from 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 don't care states. Observe that for any DFA $\mathcal{A}$, the following two conditions are equivalent: (1) $\mathcal{A}$ is consistent with $\mathcal{C}$, which means $\mathcal{A}$ accepts all accepted strings in $\mathcal{C}$ and rejects all rejected strings in $\mathcal{C}$. (2) $\mathcal{A}$ separates L_1 and L_2, which means $\mathcal{A}$ accepts all strings in L_1 and rejects all strings in L_2.

It follows that DFA's consistent with $\mathcal{C}$ and those separating L_1 and L_2 in fact coincide. We therefore reduce the problem of finding the minimal separating DFA for L_1 and L_2 to the problem of finding the minimal DFA consistent with the 3DFA $\mathcal{C}$.

By Proposition 1, $\mathcal{C}$ is both *sound* and *complete* with respect to L_1 and L_2 because $L_1 = \mathcal{L}(\mathcal{C}^-)$, the accepted strings in $\mathcal{C}$, and $L_2 = \overline{\mathcal{L}(\mathcal{C}^+)}$, the rejected strings in $\mathcal{C}$.

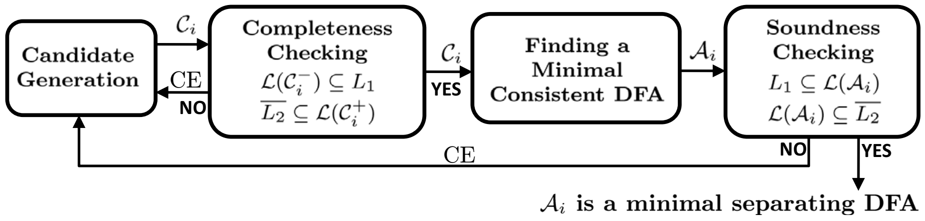

Candidate Generation
$\mathcal{C}_i$
CE
NO
Completeness Checking
$\mathcal{L}(\mathcal{C}_i^-) \subseteq L_1$
$\overline{L_2} \subseteq \mathcal{L}(\mathcal{C}_i^+)$
$\mathcal{C}_i$
YES
Finding a Minimal Consistent DFA
$\mathcal{A}_i$
Soundness Checking
$L_1 \subseteq \mathcal{L}(\mathcal{A}_i)$
$\mathcal{L}(\mathcal{A}_i) \subseteq \overline{L_2}$
CE
NO
YES
$\mathcal{A}_i$ is a minimal separating DFA

Fig. 3. Learning a Minimal Separating DFA – Overview

Figure 3 depicts the flow of our algorithm. The candidate generation step is performed by the *candidate generator*, which produces a series of candidate 3DFA's $\mathcal{C}_i$ targeting the 3DFA $\mathcal{C}$ using an extension of L^*. The *completeness checking* step examines whether $\mathcal{C}_i$ is complete with respect to L_1 and L_2. If $\mathcal{C}_i$ is incomplete, a counterexample is returned to the *candidate generator* to refine the next conjecture. Otherwise, $\mathcal{C}_i$ is complete, and the next step is to compute a minimal DFA $\mathcal{A}_i$ consistent with $\mathcal{C}_i$.

The following lemma characterizing the *sizes* of the minimal consistent DFA $\mathcal{A}_i$ and minimal separating DFA's for L_1 and L_2:

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 a DFA consistent with $\mathcal{C}_i$. Because $\mathcal{A}_i$ is the *minimal* DFA consistent with $\mathcal{C}_i$, we have $|\widehat{\mathcal{A}}| \geq |\mathcal{A}_i|$. □

Finally, we check if $\mathcal{A}_i$ separates L_1 and L_2, i.e., $L_1 \subseteq \mathcal{L}(\mathcal{A}_i)$ and $\mathcal{L}(\mathcal{A}_i) \subseteq \overline{L_2}$. If $\mathcal{A}_i$ is a separating DFA for L_1 and L_2, together with Lemma 1, we can conclude that $\mathcal{A}_i$ is a *minimal* separating DFA for L_1 and L_2. Note that even if $\mathcal{C}_i$ is unsound, it is still possible that a minimal consistent DFA of $\mathcal{C}_i$ separates L_1 and L_2. It follows that L^{Sep} may find a minimal separating DFA before the *candidate generator* produces the *sound* and *complete* 3DFA.

If $\mathcal{A}_i$ is not a separating DFA for L_1 and L_2, we get a counterexample to the *soundness* of $\mathcal{C}_i$ (will be described in the next section) and then send it to the *candidate generator* to refine the next conjecture. *Candidate generator* is guaranteed to converge to the *sound* and *complete* 3DFA, hence, our algorithm is guaranteed to find the minimal separating DFA and terminate.

4 The L^{Sep} Algorithm

L^{Sep} is an active[1] 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** where the teacher returns *true* if the given string w is in L_1, *false* if w is in L_2, and *don't care* otherwise, and
- **containment queries** where the teacher solves language containment problems of the following four types: (i) $L_1 \subseteq \mathcal{L}(A_i)$, (ii) $\mathcal{L}(A_i) \subseteq L_1$, (iii) $\overline{L_2} \subseteq \mathcal{L}(A_i)$, and (iv) $\mathcal{L}(A_i) \subseteq \overline{L_2}$. The teacher returns "YES" if the containment holds, and "NO" with a counterexample otherwise, where A_i is a conjecture DFA.

As sketched 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 generation step is performed by the *candidate generator*, which extends the observation table in L^* [17] 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 distinguishing strings, and a function T from $(S \cup S\Sigma) \times E$ to $\{+, -, ?\}$; see Figure 4 for an example. Let $\alpha \in S \cup S\Sigma$ and $\beta \in E$. The function T maps $\pi = (\alpha, \beta)$ to $+$ if $\alpha\beta \in L_1$; it maps π to $-$ if $\alpha\beta \in L_2$; otherwise T maps π to ?. In the observation table of Figure 4, the entry for (ba, b) is $+$ because the string $bab \in L_1$[2].

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* expands the observation table by extracting distinguishing strings

	λ	b
λ	$-$	?
b	?	?
ba	$-$	$+$
bab	$+$	$+$
a	$-$	?
bb	?	?
baa	$-$	$+$
$baba$	$-$	?
$babb$	$+$	$+$

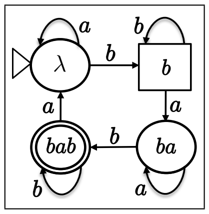


Fig. 4. An Observation Table and Its Corresponding 3DFA. The square node denotes a don't care state.

[1] A learning algorithm is *active* if it can actively query the teacher to label samples; otherwise, it is *passive*.

[2] Here $L_1{=}(a^*b^+a^+b^+)(a^+b^+a^+b^+)^*$ and $L_2{=}a^*(b^*a^+)^*$.

from counterexamples and then generates another conjecture 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 the reader to [7] for details.

Completeness Checking

The L^{Sep} algorithm 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's for L_1 and L_2 are considered, the L^{Sep} algorithm checks whether $\mathcal{C}_i$ is *complete*.

By Proposition 1, checking completeness reduces to checking whether $\mathcal{L}(\mathcal{C}_i^-) \subseteq L_1$ and $\overline{L_2} \subseteq \mathcal{L}(\mathcal{C}_i^+)$, which can be done by containment queries. L^{Sep} first builds the DFA's $\mathcal{C}_i^+$ and $\mathcal{C}_i^-$. It then submits the containment queries $\mathcal{L}(\mathcal{C}_i^-) \subseteq L_1$ 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 generation* and *completeness checking* may be needed to find a complete 3DFA.

Finding a Minimal Consistent DFA

After the *completeness checking*, the next step is to compute a minimal DFA consistent with $\mathcal{C}_i$. We reduce the problem to the minimization problem of incompletely specified finite state machines [15]. 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 [15] 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.
- $L_1 \nsubseteq \mathcal{L}(\mathcal{A}_i)$. There is a string $u \in L_1 \setminus \mathcal{L}(\mathcal{A}_i)$. Moreover, we have $\mathcal{L}(\mathcal{A}_i) \supseteq \mathcal{L}(\mathcal{C}_i^-)$ because $\mathcal{A}_i$ is consistent with $\mathcal{C}_i$. Therefore, $u \in L_1 \setminus \mathcal{L}(\mathcal{C}_i^-)$. By Proposition 1, u is a counterexample to the soundness of $\mathcal{C}_i$. It is sent to the *candidate generator* to refine the 3DFA in the next iteration.
- $\mathcal{L}(\mathcal{A}_i) \nsubseteq \overline{L_2}$. There is a string $v \in \mathcal{L}(\mathcal{A}_i) \setminus \overline{L_2}$. The string v is in fact a counterexample to the soundness of $\mathcal{C}_i$ by an analogous argument. It is sent to the *candidate generator* as well.

4.1 Correctness

The following theorem states the correctness of the L^{Sep} algorithm.

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 (submitted to soundness checking) separates L_1 and L_2, L^{Sep} terminates and returns a minimal separating DFA.
3. If the minimal consistent DFA does not separate L_1 and L_2, a counterexample to the soundness of $\mathcal{C}_i$ is sent to the *candidate generator*.
4. Because of 3, 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.

□

4.2 Complexity Analysis

We now estimate the number of queries used in the L^{Sep} algorithm. Lemma 2 states an upper bound on the size of the minimal sound and complete 3DFA (a proof can be found in [7]). By Lemma 2, the *query complexity* of L^{Sep} is established in Theorem 2.

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 $\mathcal{C}$ that accepts all strings in L_1 and rejects all strings in L_2 is smaller than $|\mathcal{B}_1| \times |\mathcal{B}_2|$.*

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 that accepts all strings in L_1 and rejects all strings in L_2. The *candidate generator* takes at most $O(|\mathcal{C}|^2 + |\mathcal{C}| \log m)$ membership queries and proposes at most $|\mathcal{C}| - 1$ incorrect 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

We discuss how to adapt L^{Sep} to the context of automated compositional verification. The adapted version is referred to as "adapted L^{Sep}". We first explain how to reduce the problem of finding a minimal assumption in assume-guarantee reasoning to the problem of finding a minimal separating automaton. We then show how adapted L^{Sep} handles the case in which the system violates the property and introduce heuristics to improve the efficiency of the adapted algorithm.

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

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

The second premise, $\mathcal{L}(M_1) \cap \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(P)$, in the rule can be rewritten as $\mathcal{L}(\mathcal{A}) \subseteq \overline{(\mathcal{L}(M_1) \cap \overline{\mathcal{L}(P)})}$[3]. Therefore, the two premises can be summarized as

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

This immediately translates the problem of finding a minimal assumption in assume-guarantee reasoning to the problem of finding a minimal separating automaton of the two languages $\mathcal{L}(M_2)$ and $\mathcal{L}(M_1) \cap \overline{\mathcal{L}(P)}$. Therefore, if the system composed of M_1 and M_2 satisfies the property P, L^{Sep} can be used to find a contextual assumption $\mathcal{A}$ that is needed by the assume-guarantee rule[4].

The case when the system violates the property: The adapted L^{Sep} algorithm handles the case that the system violates the property as follows:

1. A membership query on a string v returns *true*, *false*, or *don't care* in the same way as the original L^{Sep} algorithm.
2. In addition, it returns *fail* if v is in both input languages. If *fail* is returned by a query, the adapted L^{Sep} algorithm terminates and reports v as a witness that the two languages are not disjoint, i.e., the property is violated.[5]
3. When a conjecture query returns a counterexample w, the adapted L^{Sep} algorithm submits a membership query on w. If *fail* is not returned by the query, the algorithm proceeds as usual.

The following lemma states the correctness of the adapted L^{Sep} algorithm (a proof can be found in [7]):

Lemma 3. *If $\mathcal{L}(M_1) \cap \mathcal{L}(M_2) \nsubseteq \mathcal{L}(P)$, eventually the* fail *result will be returned by a membership query.*

[3] It can be done using the following steps: $\mathcal{L}(M_1) \cap \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(P) \Leftrightarrow (\mathcal{L}(M_1) \cap \mathcal{L}(\mathcal{A})) \cap \overline{\mathcal{L}(P)} = \emptyset \Leftrightarrow \mathcal{L}(\mathcal{A}) \cap (\mathcal{L}(M_1) \cap \overline{\mathcal{L}(P)}) = \emptyset \Leftrightarrow \mathcal{L}(\mathcal{A}) \subseteq \overline{(\mathcal{L}(M_1) \cap \overline{\mathcal{L}(P)})}$.

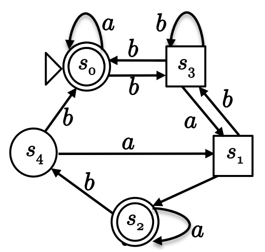
[4] The reduction was first observed by Gupta *et al.* [14].

[5] The facts that *the system violates the property* and *the two input languages are not disjoint* are equivalent to each other, which can be proved as follows: $\mathcal{L}(M_1) \cap \mathcal{L}(M_2) \nsubseteq \mathcal{L}(P) \Leftrightarrow \mathcal{L}(M_1) \cap \mathcal{L}(M_2) \cap \overline{\mathcal{L}(P)} \neq \emptyset \Leftrightarrow \mathcal{L}(M_2) \cap (\mathcal{L}(M_1) \cap \overline{\mathcal{L}(P)}) \neq \emptyset$.

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.

We first use Paull and Unger's algorithm [15] to find the sets of "maximal" compatible states[6], which are the candidates for the states in the reduced DFA. Consider an example shown in Figure 5. We have $Q_1 = \{s_0, s_1\}$, $Q_2 = \{s_0, s_2\}$, $Q_3 = \{s_0, s_3, s_4\}$.

We then choose the largest set from $\{Q_1, Q_2, Q_3\}$ that contains s_0 as the initial state of the reduced DFA. Here we take Q_3. The next state of Q_3 after reading symbol a is the largest set $Q' \in \{Q_1, Q_2, Q_3\}$ that satisfies $Q' \supseteq \{s' \mid s' = \delta(s, a),\ \textit{for all}\ s \in Q_3\} = \{s_0, s_1\}$. Here we get Q_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 $Q \in \{Q_1, Q_2, Q_3\}$ can be found using the same procedure. The procedure terminates after the transition function of the reduced DFA is completely specified. The state Q is an accepting state in the reduced DFA if there exists a state $s \in Q$ such that s is an accepting state in the 3DFA, otherwise it is a rejecting state in the reduced DFA. Formally, we define the reduced DFA $(\Sigma, \hat{S}, \hat{s_0}, \hat{\delta}, \hat{F})$ as follows, let $\mathcal{Q}$ be the sets of maximal compatible states:



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

Fig. 5. The 3DFA to be reduced

- $\hat{S} \subseteq \mathcal{Q}$; $\hat{s_0} = Q \in \mathcal{Q}$, where Q is the largest set that contains s_0;
- $\hat{\delta}(\hat{s}, a) = \hat{s}'$, where $\hat{s}'$ is the largest set $Q \in \mathcal{Q}$ such that $Q \supseteq \{s' \mid s' = \delta(s, a),\ \textit{for all}\ s \in \hat{s}\}$;
- $\hat{s} \in \hat{F}$ if there exists a state $s \in \hat{s}$ such that $s \in A$, where A is the set of accepting states in the 3DFA.

According to our experimental results, 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 *completeness checking* in the adapted version. *Completeness checking* takes a lot of time because the two DFA's C_i^+ and C_i^- can be large and several iteration between *candidate generation* and *completeness checking* may be needed to find a complete 3DFA.

[6] Two states are *incompatible* if there exists some string that leads one of them to an accepting state and leads the other to a rejecting state. Otherwise, the two states are *compatible*. The states in a *set of compatible states* are pairwise compatible. A set of compatible states Q is *maximal* if there exists no other set of compatible states Q' such that $Q' \supset Q$.

6 Experiments

We evaluated L^{Sep} and its adapted version by two sets of experiments. First, we compared the L^{Sep} algorithm with the algorithm of Gupta *et al.* [14] and that of Grinchtein *et al.* [13] on a large set of randomly-generated sample problems. Second, we evaluated the *adapted* L^{Sep} algorithm and compared it with other automated compositional verification algorithms on the LTSA benchmarks [9]. A more detailed description of the settings of our experiments can be found in [7].

6.1 Experiment 1

Avg. DFA Size	13	21	32	42	54	70	86	102	124
(i,j)	(4,4)	(5,4)	(6,4)	(7,4)	(8,4)	(9,4)	(10,4)	(11,4)	(12,4)
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 [14]	6.6	58.7	266.7	431.5	1308.8	>4000	>4000	>4000	>4000
Grinchtein [13]	51.8	139	255.6	514.7	>4000	>4000	>4000	>4000	>4000

Avg. DFA Size	16	24	36	48	63	80	99	119	142
(i,j)	(4,8)	(5,8)	(6,8)	(7,8)	(8,8)	(9,8)	(10,8)	(11,8)	(12,8)
Algorithms	Average execution time								
L^{Sep}	0.15	0.44	0.96	2.1	3.7	6.4	11	17.8	26.9
Gupta [14]	96.2	625.9	972.3	>4000	>4000	>4000	>4000	>4000	>4000
Grinchtein [13]	813.4	>4000	>4000	>4000	>4000	>4000	>4000	>4000	>4000

Unit: Second

Table 1. Comparison of the Three Algorithms. The row "Avg. DFA Size" is the average size of the two input DFA's $\mathcal{B}_1$ and $\mathcal{B}_2$ in a sample problem. Each column is the average result of 100 sample problems. The row "(i,j)" is the parameters of the sample generator.

We first describe the *sample generator*. Each sample problem has two DFA's $\mathcal{B}_1$ and $\mathcal{B}_2$ such that $\mathcal{L}(\mathcal{B}_1) \subseteq \mathcal{L}(\mathcal{B}_2)$. The *sample generator* has two input parameters i and j. It first randomly generates[7] two DFA's $\mathcal{A}_1$ and $\mathcal{A}_2$ such that $|\mathcal{A}_1| = |\mathcal{A}_2| = i$. Both use the same alphabet, which is of size j. Then the *sample generator* builds the DFA $\mathcal{B}_1$ by constructing the minimal DFA that recognizes $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$ and $\mathcal{B}_2$ by constructing the minimal DFA that recognizes $\mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$. The *sample generator* has two important properties: (1) the difference between $|\mathcal{B}_1|$ and $|\mathcal{B}_2|$ is small; (2) there exists a (relatively) small separating DFA for $\mathcal{B}_1$ and $\mathcal{B}_2$.

We used eighteen different input parameters ($i = 4 \sim 12, j = 4, 8$). For each pair (i, j), we randomly generated a set of 100 different sample problems (we eliminated duplications). The *average sizes* of input DFA's ranging from 13 to 142. We also dropped trivial cases ($|\mathcal{B}_1| = 1$ or $|\mathcal{B}_2| = 1$). Table 1 shows the results. We set a timeout of 4000 seconds (for each set of 100 sample problems).

[7] For each state s in $\mathcal{A}_1$ (respectively $\mathcal{A}_2$) and for each symbol a, a destination state s' in $\mathcal{A}_1$ (respectively $\mathcal{A}_2$) 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.

If the algorithm did not solve any problem in a set of 100 problems within the timeout period, we mark it as >4000. The time spent on failed tasks is included in the total processing time.

6.2 Experiment 2

	L^{Sep}		Cobleigh		Gupta		Problem	MO
	Time	|A|	Time	|A|	Time	|A|	Size	
1-2	**0.1**	3	170	74	32	3	45, 80	0.08
1-3	**0.4**	3	-	-	109	3	82, 848	0.7
1-4	**1.6**	3	-	-	219	3	138, 4046	4.2
2-2	508	7	**89**	52	-	-	39, 89	0.08
2-3	-	-	**1010**	93	-	-	423, 142	0.7
2-4	-	-	**7063**	152	-	-	2022, 210	4
3-2	**1.9**	3	51	57	140	3	39, 100	0.09
3-3	**13**	3	601	110	551	3	423, 164	0.8
3-4	**55**	3	4916	189	1639	3	2022, 69	4.2
4-2	**5.8**	3	21	35	90	3	39, 87	0.09
4-3	**20.8**	3	1109	103	433	3	423, 140	0.75
4-4	**44.9**	3	6390	156	793	3	2022, 208	4.1
5-2	**940**	64	998	127	-	-	45, 133	0.08
7-2	362	39	**48**	46	-	-	39, 104	0.09
7-3	-	-	**405**	76	-	-	423, 168	0.9
7-4	-	-	**3236**	123	-	-	2022, 256	4.1
9-2	**1345**	52	4448	240	-	-	45, 251	0.09
10-2	6442	18	-	-	**196**	3	151, 309	0.8
10-3	5347	22	-	-	**601**	3	327, 3369	6.1
10-4	-	-	-	-	**1214**	3	658, 16680	33
11-2	**6533**	82	-	-	-	-	151, 515	0.8
12-2	**36**	4	1654	162	-	-	151, 273	0.8
12-3	**133**	4	-	-	-	-	327, 2808	6.6
12-4	**450**	4	-	-	-	-	658, 13348	33
15-2	**1477**	88	-	-	5992	3	151, 309	0.8
15-3	5840	5	-	-	**4006**	3	327, 3369	5.9
15-4	-	-	-	-	**6880**	3	658, 16680	33
19-2	**5.8**	3	-	-	266	3	234, 544	0.3
19-3	**13**	3	-	-	1392	3	962, 5467	2.9
19-4	**69**	3	-	-	7636	3	2746, 52852	35
21-3	**45**	3	-	-	4558	3	962, 5394	2.9
21-4	**718**	3	-	-	3839	3	2746, 51225	34.8
22-2	**0.6**	3	8	25	12	3	900, 30	0.3
22-3	**2.3**	3	1242	193	54	3	7083, 264	4.6
22-4	**11**	3	-	-	170	3	30936, 2190	33
23-2	92	9	**8.9**	37	-	-	50, 40	0.1
24-2	1.2	6	**0.2**	12	1.2	3	13, 14	0.01
24-3	5.1	6	**0.33**	12	-	-	48, 14	0.02
24-4	18	6	**0.63**	12	-	-	157, 14	0.1
25-2	**1156**	5	3050	257	-	-	41, 260	0.1
26-2	512	38	**239**	121	-	-	65, 123	0.1
27-2	848	46	**830**	193	-	-	41, 204	0.1
28-2	**755**	46	757	185	-	-	41, 188	0.1
29-2	926	21	**891**	193	-	-	41, 195	0.1
30-2	1083	24	**986**	193	-	-	41, 195	0.1
31-2	**204**	5	274	121	4975	3	65, 165	0.1
32-2	**9.9**	3	646	193	121	3	41, 261	0.1
32-3	**44**	3	-	-	-	-	1178, 4806	2.6
32-4	**886**	3	-	-	-	-	289, 117511	382

Table 2. Experimental Results on the LTSA Benchmarks. The "L^{Sep}" column is the result of the adapted L^{Sep} algorithm. "Time" is the execution time in seconds and $|A|$ is the size of the contextual assumption found by the algorithm. "Cobleigh" and "Gupta" give results from [10] and [14], respectively. We highlight in bold font the best results. The column "Problem Size" is the pair $(|M_2|, |M_1| \times |\overline{P}|)$, where $|M_2|$ is the size of the DFA M_2 and $|M_1| \times |\overline{P}|$ is the size of the product of the two DFA's $\overline{M_1}$ and P. The column "MO" is the execution time for monolithic verification. The symbol "-" indicates that the algorithm did not finish within the timeout period. For each row, we use n-m to denote benchmark problem n with m components.

We evaluated the adapted L^{Sep} algorithm on the LTSA benchmarks [9]. We compared the adapted L^{Sep} algorithm with the algorithms of Gupta *et al.*, Grinchtein *et al.*, and Cobleigh *et al.* [10]. We implemented all of those algorithms, including the heuristic algorithm for minimizing a 3DFA. We did not consider optimization techniques such as alphabet refinement [6, 12]. This is fair because such techniques can also be easily adapted to L^{Sep}. The experimental results are shown in Table 2. The sizes of components 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. We used the decomposition suggested by the benchmarks to build components M_1 and M_2. Furthermore, we swapped M_1 and M_2; in [9], they check $\mathcal{L}(M_1) \subseteq \mathcal{L}(\mathcal{A})$ and $\mathcal{L}(M_2) \cap \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(P)$ in the experiments. We swapped them because in the original arrangement, a large portion of the cases have an assumption of size 1. We set a timeout of 10000 seconds. Actually we checked all the 89 LTSA benchmark problems (of 2 ,3, and 4 components). 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 the timeout period (30 cases). In addition, we do not list the result of Grinchtein *et al.* because of the space limitation. In this set of experiments, it cannot solve most of the problems within the timeout period (84 cases). Even if it solved the problem (5 cases), it is slower than others.

The adapted L^{Sep} algorithm performs better than all the other algorithms in 30 among the 49 problems. The algorithm of Cobleigh *et al.* wins 14 problems. However, in 8 of the 14 cases (23-2, 24-2, 24-3, 24-4, 26-2, 27-2, 29-2, 30-2), their algorithm finds an assumption with size almost the same as $|\overline{M_1} \times P|$. In those cases, there is no hope of defeating monolithic verification. In contrast, our algorithm scales better than monolithic verification in several problem sets. For example, in 1-m, 19-m, 22-m, and 32-m, the execution time of the adapted L^{Sep} algorithm grows much slower than monolithic verification. In 1-m and 22-m, we can see that the adapted L^{Sep} algorithm takes more execution time than monolithic verification when the number of components is 2, but its performance surpasses monolithic verification when the number of components becomes 4.

7 Discussion and Further Work

The algorithm of Gupta *et al.* is *passive*, using only containment queries (which is slightly more general than equivalence queries). From a lower bound result by Angluin [3] on learning with equivalence queries, the query complexity of the algorithm of Gupta *et al.* can be shown to be exponential in the sizes of the minimal DFA's of the two input languages. Moreover, the data structures that they use to represent the samples are essentially trees, which may grow exponentially. These explain why their algorithm does not perform well in the experiments.

The algorithm of Grinchtein *et al.* [13] is an improved version of an earlier algorithm of Pena and Oliveira [16], which is *active*. However, according to our experiments, this improved active algorithm is outperformed by the purely passive learning algorithm of Gupta *et al.* in most cases. The main reason for the inefficiency of this particular active learning algorithm seems to be that the membership queries introduce a lot of redundant samples, even though they reduce the number of iterations required. The redundant samples substantially increase the running time of the exponential procedure of computing the minimal DFA. In contrast, our active algorithm L^{Sep} indeed performs better than the passive algorithm of Gupta *et al.*

The better performance of L^{Sep} can be attributed to the facts that the algorithm utilizes membership queries to accelerate learning and has a more compact representation of the samples (a 3DFA) collected from the queries. For further work, it will be interesting to adapt L^{Sep} for other applications, such as inferring network invariants of parameterized systems and to evaluate the performance of the resulting solutions. Given that L^{Sep} is a better learning algorithm, we hope that other applications will also benefit from it.

References

1. R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *CAV 2005, LNCS 3576*, pages 548–562. Springer, 2005.
2. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
3. D. Angluin. Negative results for equivalence queries. *Machine Learning*, 5(2):121–150, 1990.
4. H. Barringer, D. Giannakopoulou, and C.S. Păsăreanu. Proof rules for automated compositional verification through learning. In *SAVCBS 2003*, pages 14–21, 2003.
5. S. Chaki, E.M. Clarke, N. Sinha, and P. Thati. Dynamic component substitutability analysis. In *FME 2005, LNCS 3582*, pages 512–528. Springer, 2005.
6. S. Chaki and O. Strichman. Optimized L*-based assume-guarantee reasoning. In *TACAS 2007, LNCS 4424*, pages 276–291. Springer, 2007.
7. Y.-F. Chen, A. Farzan, E.M. Clarke, Y.-K. Tsay, and B.-Y. Wang. Learning minimal separating DFA's for compositional verification. Technical Report CMU-CS-09-101, Carnegie Mellon Univeristy, 2009.
8. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.
9. J.M. Cobleigh, G.S. Avrunin, and L.A. Clarke. Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. *ACM Transactions on Software Engineering and Methodology*, 7(2):1–52, 2008.
10. J.M. Cobleigh, D. Giannakopoulou, and C.S. Păsăreanu. Learning assumptions for compositional verification. In *TACAS 2003, LNCS 2619*, pages 331–346. Springer.
11. A. Farzan, Y.-F. Chen, E.M. Clarke, Y.-K. Tsay, and B.-Y. Wang. Extending automated compositional verification to the full class of omega-regular languages. In *TACAS 2008, LNCS 4693*, pages 2–17. Springer, 2008.
12. M. Gheorghiu, D. Giannakopoulou, and C.S. Păsăreanu. Refining interface alphabets for compositional verification. In *TACAS 2007, LNCS 4424*, pages 292–307. Springer, 2007.
13. O. Grinchtein, M. Leucker, and N. Piterman. Inferring network invariants automatically. In *IJCAR 2006, LNAI 4130*, pages 483–497. Springer, 2006.
14. A. Gupta, K.L. McMillan, and Z. Fu. Automated assumption generation for compositional verification. In *CAV 2007, LNCS 4590*, pages 420–432. Springer, 2007.
15. M.C. Paull and S.H. Unger. Minimizing the number of states in incompletely specified sequential switching functions. *IRE Transitions on Electronic Computers*, EC-8:356–366, 1959.
16. J.M. Pena and A.L. Oliveira. A new algorithm for the reduction of incompletely specified finite state machines. In *ICCAD 1998*, pages 482–489. ACM Press, 1998.
17. R.L. Rivest and R.E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.
18. N. Sinha and E.M. Clarke. SAT-based compositional verification using lazy learning. In *CAV 2007, LNCS 4590*, pages 39–54. Springer, 2007.