# Automated Assumption Generation for Compositional Verification

Anubhav Gupta    K. L. McMillan    Zhaohui Fu

Cadence Design Systems, Inc.

**Abstract.** We describe a method for computing a minimum-state automaton to act as an intermediate assertion in assume-guarantee reasoning, using a sampling approach and a Boolean satisfiability solver. For a set of synthetic benchmarks intended to mimic common situations in hardware verification, this is shown to be significantly more effective than earlier approximate methods based on Angluin's L* algorithm. For many of these benchmarks, this method also outperforms BDD-based model checking and interpolation-based model checking. We also demonstrate how domain knowledge can be incorporated into our algorithm to improve its performance.

## 1 Introduction

Compositional verification is a promising approach for alleviating the state-explosion problem in model checking. This technique decomposes the verification task for the system into simpler verification problems for the individual components of the system. Consider a system $M$ composed of two components $M_1$ and $M_2$, and a property $P$ that needs to be verified on $M$. The *assume-guarantee* style for compositional verification uses the following inference rule:

$$\frac{\begin{array}{c}\langle true\rangle\ M_1\ \langle A\rangle \\ \langle A\rangle\ M_2\ \langle P\rangle\end{array}}{\langle true\rangle\ M_1 \parallel M_2\ \langle P\rangle} \tag{1}$$

This rule states that $P$ can be verified on $M$ by identifying an assumption $A$ such that: $A$ holds on $M_1$ in all environments and $M_2$ satisfies $P$ in any environment that satisfies $A$. For example, consider a system consisting of a processor and memory unit, communicating over a bus. We want to prove that an addition instruction on the system executes correctly. In order to prove this property, we decompose this system into two components, $M_1$ and $M_2$, where $M_1$ consists of the bus and $M_2$ consists of the processor and memory unit. We identify an assumption about $M_1$ which states that a data value $x$ is read from $M_1$ if and only if $x$ was written on $M_1$. Using this assumption, we prove the two antecedents of the assume-guarantee rule and this proves the property on $M$. Since our assumption is much simpler than the component $M_1$ itself, proving the two antecedents is much easier than proving the property directly on $M$. There are two key challenges to an assume-guarantee based verification strategy:

1. Identifying an appropriate decomposition of the system.
2. Identifying a simple assumption.

In this article, we presume that the system has been decomposed and we focus on assumption identification.

In a language-theoretic framework, we model a process as a regular language, specified by a finite automaton. Process composition is intersection of languages, and a process satisfies a property $P$ when its intersection with $\mathcal{L}(\neg P)$, the set of strings not satisfying $P$, is empty. The above inference rule can thus be written as:

$$\frac{\begin{array}{rcl}\mathcal{L}(M_1) & \subseteq & \mathcal{L}(A) \\ \mathcal{L}(A) \cap \mathcal{L}(M_2) \cap \mathcal{L}(\neg P) & = & \emptyset\end{array}}{\mathcal{L}(M_1) \cap \mathcal{L}(M_2) \cap \mathcal{L}(\neg P) \ = \ \emptyset} \tag{2}$$

To simplify matters, we can use a satndard product construction to produce an automaton $M_2'$ such that $\mathcal{L}(M_2') = \mathcal{L}(M_2) \cap \mathcal{L}(\neg P)$. That is, $M_2'$ accepts all the strings of $M_2$ that do not satisfy property $P$. The problem of constructing an assume-guarantee argument then amounts to finding an automaton $A$ that separates $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2')$, in the sense that $\mathcal{L}(A)$ accepts all the strings in $\mathcal{L}(M_1)$, but rejects all the strings in $\mathcal{L}(M_2')$. Clearly, we would like to find an automaton $A$ with as few states as possible, to minimize the state-explosion problem in checking the antecedents of the assume-guarantee rule.

For deterministic automata, the problem of finding a minimum-state separating automaton is NP-complete. It is reducible to the problem of finding a minimum-state implementation of an Incomplete Deterministic Finite Automaton (IDFA), shown to be NP-complete by Pfleeger [Pfl73]. To avoid this complexity, Cobleigh et al. proposed a polynomial-time approximation method [CGP03] based on a modification of Angluin's L* algorithm [Ang87,RS89] for active learning of a regular language. The primary drawback of this approach is that there is no approximation bound. That is, in the worst case, the algorithm will return one of the extremal solutions (either $\mathcal{L}(M_1)$ or $\mathcal{L}(M_2')^c$, depending on the implementation). In this case there is no benefit in terms of state space reduction that could not be obtained by determinizing and minimizing $M_1$ or $M_2'$. Alur et al. [AMN05] have presented a symbolic implementation of this approach, which suffers from the same drawback. In fact, in our experiments with hardware verification problems, the L*-based approach failed to produce a state reduction for any of our benchmark problems.

In this paper, we argue that it may be worthwhile to solve the minimum-state separating automaton problem exactly. Since the overall verification problem is PSPACE-complete when $M_1$ and $M_2'$ are expressed symbolically, there is no reason to require that the sub-problem of finding an intermediate assertion be solved in polynomial time. The goal of assume-guarantee reasoning is a verification procedure that is singly exponerntial in $|M_1|$ and in $|M_2'|$, but not in $|M_1| + |M_2'|$, where $|M|$ denotes the textual size of $M$. If this is achieved, it may not matter that the overall complexity is exponential in $|A|$, provided $A$ is small.

With this rationale in mind, we present an exact approach to the minimum-state separating automaton problem, suited to assume-guarantee reasoning for hardware verification. We apply the sampling-based algorithm used by Pena and Oliveira [PO99] for the IDFA minimization problem. This algorithm iteratively generates sample strings in $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2')$, computing at each step a minimum-state automaton consistent with the sample set. Finding a minimum-state automaton consistent with a set of labeled strings is itself an NP-complete problem [Gol78], and we solve it using a Boolean Satisfiability (SAT) solver. We use the sampling approach here because the standard techniques for solving the IDFA minimization problem [KVBSV97] require explicit state representation, which is not practical for hardware verification.

For hardware applications, we must also deal with the fact that the alphabet is exponential in the number of Boolean signals connecting $M_1$ and $M_2'$. This difficulty is also observed in L*-based approaches, where the number of queries is proportional to the size of the alphabet. We handle this problem by learning an automaton over a partial alphabet and generalizing to the full alphabet using Decision Tree Learning [Mit97] methods.

The performance of our iterative algorithm depends on the number of sample strings in $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2')$ that it requires to generalize to the minimum-state separating automaton. We demonstrate how domain-specific properties of the separating automaton can be incorporated into our algorithm to speed up its convergence.

Using a collection of synthetic hardware benchmarks, we show that our approach is effective in producing minimum-state intermediate assertions in cases where the approximate L* approach yields no reduction. In some cases, our method also provides a substantial reduction in overall verification time compared to direct model checking using state-of-the-art methods.

## 2 Preliminaries

### 2.1 Deterministic Finite Automaton

**Definition 1.** *A* Deterministic Finite Automaton (DFA) *$M$ is a tuple $(S, \Sigma, s_0, \delta, F)$ where: (1) $S$ is a finite set of states, (2) $\Sigma$ is a finite alphabet, (3) $\delta : S \times \Sigma \rightarrow S$ is a transition function, (4) $s_0 \in S$ is the initial state, and (5) $F \subseteq S$ is the set of accepting states.*

**Definition 2.** *An* Incomplete Deterministic Finite Automaton (IDFA) *$M$ is a tuple $(S, \Sigma, \delta, s_0, F, R)$ where: (1) $S$ is a finite set of states, (2) $\Sigma$ is a finite alphabet, (3) $\delta : S \times \Sigma \rightarrow (S \cup \{\bot\})$ is a partial transition function, (4) $s_0 \in S$ is the initial state, (5) $F \subseteq S$ is the set of accepting states, and (6) $R \subseteq S$ is the set of rejecting states.*

Intuitively, an IDFA is incomplete because some states may not have outgoing transitions for the complete alphabet, and some states are neither accepting nor rejecting. If there is no transition from state $s$ on symbol $a$ then $\delta(s, a) = \bot$. For both DFA's and IDFA's, we extend the transition function $\delta$ in the usual way to apply to strings. That is, if $\pi \in \Sigma^*$ and $a \in \Sigma$ then $\delta(s, \pi a) = \delta(\delta(s, \pi), a)$ when $\delta(s, \pi) \neq \perp$ and $\delta(s, \pi a) = \perp$ otherwise.

A string $s$ is *accepted* by a DFA $M$ if $\delta(s_0, s) \in F$, otherwise $s$ is *rejected* by $M$. A string $s$ is *accepted* by an IDFA if $\delta(q_0, s) \in F$. A string $s$ is *rejected* by an IDFA if $\delta(q_0, s) \in R$.

Given two languages $L_1, L_2 \subseteq \Sigma^*$, we will say that a DFA or IDFA *separates* $L_1$ and $L_2$ when it accepts all strings in $L_1$ and rejects all strings in $L_2$. A *minimum-state separating automaton* (MSA) for $L_1$ and $L_2$ is an automaton with minimum number of states separating $L_1$ and $L_2$ (we will apply this notion to either DFA's or IDFA's as the context warrants).

## 3 The L* approach

For comparison purposes, we first describe the L*-based approximation method for learning separating automata [CGP03]. In the L* algorithm, a *learner* infers the minimum-state DFA $A$ for an unknown regular language $L$ by posing *queries* to a *teacher*. In a *membership* query, the learner provides a string $\pi$, and the teacher replies yes if $\pi \in L$ and no otherwise. In an *equivalence* query, the learner proposes an automaton $A$, and the teacher replies yes if $\mathcal{L}(A) = L$ and otherwise provides a counterexample. The counterexample may be positive (*i.e.*, a string in $L \setminus \mathcal{L}(A)$) or negative (*i.e.*, a string in $\mathcal{L}(A) \setminus L$). Angluin [Ang87] gave an algorithm for the learner that guarantees to discover $A$ in a number of queries polynomial in the size of $A$.

Cobleigh et al. [CGP03] modified this procedure to learn a separating automaton for two languages $L_1$ and $L_2$. Their procedure differs from the L* algorithm in the responses provided by the teacher. In the case of an equivalence query, the teacher responds yes if $A$ is a separating automaton for $L_1$ and $L_2$. Otherwise, it provides either a positive counterexample as a string in $L_1 \setminus \mathcal{L}(A)$ or a negative counterexample as a string in $L_2 \cap \mathcal{L}(A)$. To a membership query on a string $\pi$, the teacher responds yes if $\pi \in L_1$ and no if $\pi \in L_2$. If $\pi$ is in neither $L_1$ nor $L_2$, the correct choice is unknown, since the teacher does not know the minimum-state separating automaton. One possible policy is to always answer no in the unknown case. Thus, in effect, the teacher is asking the learner to learn $L_1$, but is willing to accept any guess that separates $L_1$ and $L_2$. Alternatively, the teach can always answer yes in the unknown case. Under this policy, the teacher is asking the learner to learn $L_2^c$, but is willing to accept any guess that separates $L_1$ and $L_2$.

Using Angluin's algorithm for the learner, we can show that the learned separating automaton $A$ has no more states that the minimum-state deterministic automaton for $L_1$ (or alternatively $L_2^c$). This can, however, be arbitrarily larger than the minimum-state separating automaton.

As in Angluin's original algorithm, the number of queries is polynomial in the size of $A$, and in particular, the number of equivalence queries is at most the number of states in $A$. In the assume-guarantee application, $L_1 = \mathcal{L}(M_1)$ and

$L_2 = \mathcal{L}(M_2')$. For hardware verification, $M_1$ and $M_2'$ are Non-deterministic Finite Automata (NFA's) represented symbolically (the non-determinism arising from hidden inputs and from the construction of the automaton for $\neg P$). Answering a membership query is therefore NP-complete (essentially a bounded model checking problem) while answer an equivalence query is PSPACE-complete (a symbolic model checking problem).

We now consider the overall worst-case complexity of the approach using symbolic representations. We will assume the version of the algorithm in which the teach answers no for unknown queries (the other case is similar). The execution time of the algorithm includes both membership and equivalence queries. The number of both of these is bounded by $|A|$, the final textual size of automaton $A$ explicitly represented. The cost of testing membership of a string in $\mathcal{L}(M_1)$ is propertional to the length of the string and the numebr of states in $M_1$ (since $M_1$ is non-deterministic). This gives a worst-case run time of $O((2^{|M_1|} + 2^{|M_2'|}) \times (n + |A|) \times |A|)$ where $n$ represents the length of the longest counterexample produced by the teacher (and thus the length of the longest membership query). In the worst case, $|A|$ can be exponential in the number of states of $M_1$ (since it can be a determinization of $M_1$). Thus it can be doubly exponential in $|M_1|$, the symbolic textual size of $M_1$. Moreover, assuming the teacher produces minimum-length counterexamples, $n$ can be as large as $|A|$ times the number of states of $M_1$ (alternatively $M_2'$). Thus, the worst-case complexity is actually much worse than simply computing the product of $M_1$ and $M_2'$ (which is only singly exponential in $|M_1| + |M_2'|$). The method only provides an advantage when both $|A|$ and $n$ are small.

## 4 Computing the minimum-state separating automaton

To find an exact MSA for two languages $L_1$ and $L_2$, we will follow the general approach of Pena and Oliveira [PO99] for minimizing IDFA's. This is a learning approach that uses only equivalence queries. It relies on a subroutine that can compute a minimum-state DFA separating two *finite* sets of strings. Although Pena and Oliveira's work is limited to finite automata, the technique can be applied to *any* languages $L_1$ and $L_2$ that have a regular separator, even if $L_1$ and $L_2$ are themselves not regular.

The overall flow of our LangMSA algorithm for computing an MSA for two languages is shown in Algorithm 1. We maintain two sets of sample strings, $S_1 \subseteq L_1$ and $S_2 \subseteq L_2$. The main loop begins by computing a minimum-state DFA $A$ that separates $S_1$ and $S_2$ (line 3), using the SatMSA algorithm described below. The learner then performs an equivalence query on $A$ (lines 4,5). If $A$ separates $L_1$ and $L_2$, the procedure terminates (line 6). Otherwise, we obtain a counterexample string $\pi$ from the teacher (lines 8,14). If $\pi \in L_1$ (and consequently, $\pi \notin \mathcal{L}(A)$) we add $\pi$ to $S_1$ (line 12), else we add $\pi$ to $S_2$ (line 18). This procedure is repeated until an equivalence query succeeds. In the figure, we test first for a negative counterexample, and then for a positive counterexample.

**Algorithm 1** Computing an MSA for two languages

```
LangMSA ($L_1, L_2$)
1: $S_1 = \{\}$; $S_2 = \{\}$;
2: while (1) do
3:     Let $A$ be an MSA for $S_1$ and $S_2$;
4:     if $L_1 \subseteq \mathcal{L}(A)$ then
5:         if $\mathcal{L}(A) \cap L_2 = \emptyset$ then
6:             return $A$; (A separates $L_1$ and $L_2$)
7:         else
8:             Let $\pi \in L_2$ and $\pi \in \mathcal{L}(A)$; (negative counterexample)
9:             if $\pi \in L_1$ then
10:                return false; ($L_1$ and $L_2$ are not disjoint)
11:            else
12:                $S_1 = S_1 \cup \{\pi\}$;
13:    else
14:        Let $\pi \in L_1$ and $\pi \notin A$; (positive counterexample)
15:        if $\pi \in L_2$ then
16:            return false; ($L_1$ and $L_2$ are not disjoint)
17:        else
18:            $S_2 = S_2 \cup \{\pi\}$;
```

This order is arbitrary, and in practice we choose the order randomly for each query to avoid biasing the result towards $L_1$ or $L_2$.

The teacher in this procedure can be implemented using a model checker. That is, the checks $L_1 \subseteq \mathcal{L}(A)$ (line 4) and $\mathcal{L}(A) \cap L_2 = \emptyset$ (line 5) are model checking problems. In our application, $L_1$ and $L_2$ are the languages of symbolically represented NFA's, and we use symbolic model checking methods [McM93] to perform the checks (note that testing containment in $\mathcal{L}(A)$ requires complementing $A$, but this is straightforward since $A$ is deterministic). The checks $\pi \in L_1$ (line 9) and $\pi \in L_2$ (line 15) can be implemented using a bounded model checker [BCCZ99], because the length of the counterexample is known.

**Theorem 1.** *Let $L_1, L_2 \subseteq \Sigma^*$, for finite $\Sigma$. If $L_1$ and $L_2$ have a regular separator, then Algorithm* LangMSA *terminates and outputs a minimum-state separating automaton for $L_1$ and $L_2$.*

*Proof.* Let $A'$ be a minimum-state separating automaton for $L_1$ and $L_2$ with $k$ states. Since $S_1 \subseteq L_1$ and $S_2 \subseteq L_2$, it follows that $A'$ is also a separating automaton for $S_1$ and $S_2$. Thus, $A$ has no more than $k$ states (since it is a minimum-state separating automaton for $S_1$ and $S_2$). Thus, if the procedure terminates, $A$ is a minimum-state separating automaton for $L_1$ and $L_2$. Moreover, there are finitely many DFA's over finite $\Sigma$ with $k$ states. At each iteration, one such automaton is ruled out as a separator of $S_1$ and $S_2$. Thus, the algorithm must terminate. □

It now remains only to find an algorithm to compute a minimum-state separating automaton for the finite languages $S_1$ and $S_2$ (line 3). This problem has been studied extensively, and is known to be NP-complete [Gol78]. To solve it, we will borrow from the approach of Oliveira and Silva [OS98].

**Definition 3.** *An IDFA $M = (S, \Sigma, s_0, \delta, F, R)$ is* tree-like *when the relation $\{(s_1, s_2) \in S^2 \mid \exists a.\ \delta(s_1, a) = s_2\}$ is a directed tree rooted at $s_0$.*

Given any two disjoint finite sets of strings $S_1$ and $S_2$, we can construct a tree-like IDFA that accepts $S_1$ and rejects $S_2$, which we will call $\textsc{TreeSep}(S_1, S_2)$.

**Definition 4.** *Let $S_1, S_2 \subseteq \Sigma^*$ be disjoint, finite languages. The tree-like separator $\textsc{TreeSep}(S_1, S_2)$ for $S_1$ and $S_2$ is the tree-like IDFA $(S, \Sigma, s_0, \delta, F, R)$ where $S$ is the set of prefixes of $S_1 \cup S_2$, $s_0$ is the empty string, $F = S_1$, $R = S_2$, and $\delta(\pi, a) = \pi a$ if $\pi a \in S$ else $\delta(\pi, a) = \bot$.*

**Definition 5.** *Let $M = (S, \Sigma, s_0, \delta, F, R)$ and $M' = (S', \Sigma, s'_0, \delta', F', R')$ be two IDFA's over alphabet $\Sigma$. The map $\phi : S \to S'$ is a* folding *of $M$ onto $M'$ when:*

- *$\phi(s_0) = s'_0$,*
- *For all $s \in S$, $a \in \Sigma$, if $\delta(s, a) \neq \bot$ then $\phi(\delta(s, a)) = \delta'(\phi(s), a)$,*
- *For all $s \in F$, $\phi(s) \in F'$, and*
- *For all $s \in R$, $\phi(s) \in R'$.*

**Lemma 1.** *Let $M = (S, \Sigma, s_0, \delta, F, R)$ and $M' = (S', \Sigma, s'_0, \delta', F', R')$ be two IDFA's over alphabet $\Sigma$. Let $\phi : S \to S'$ be a folding of $M$ onto $M'$. Then, for all $t \in \Sigma^*$, if $\delta(s_0, t) \neq \bot$ then $\phi(\delta(s_0, t)) = \delta'(\phi(s_0), t)$.*

*Proof.* The proof follows directly by induction on the length of $t$.

The following theorem says that every separating IDFA for $S_1$ and $S_2$ can be obtained as a folding of the tree-like automaton $\textsc{TreeSep}(S_1, S_2)$.

**Theorem 2.** *Let $T = (S, \Sigma, s_0, \delta, F, R)$ be a tree-like IDFA, with accepting set $S_1$ and rejecting set $S_2$. Then an IDFA $A = (S', \Sigma, s'_0, \delta', F', R')$ over $\Sigma$ is a separating automaton for $S_1$ and $S_2$ if and only if there exists a folding $\phi$ from $T$ to $A$.*

*Proof.* Suppose there exists a folding $\phi$ from $T$ to $A$. Let $s$ be a string in $S_1$. Since $T$ accepts $S_1$, $\delta(s_0, s) \in F$. Since $\phi$ is a folding from $T$ onto $A$, $\phi(\delta(s_0, s) \in F'$. Using Lemma 1, we get $\phi(\delta(s_0, s)) = \delta'(s'_0, s)$, which implies $\delta'(s'_0, s) \in F'$. Thus, $A$ accepts $s$. Similarly, we can show that if $s$ is a string in $S_2$, then $A$ rejects $s$. Therefore, $A$ is a separating automaton for $S_1$ and $S_2$.

Conversely, suppose $A$ is a separating automaton for $S_1$ and $S_2$. We define a function $\phi : S \to S'$ as $\phi(s) = \delta'(s'_0, s)$. Note that by Definition 4, each state $s \in S$ is a string in $\Sigma^*$. We now prove that $\phi$ is a folding from $T$ onto $A$, by showing that it satisfies the conditions specified in Definition 5.

- $\phi(s_0) = \delta'(s'_0, s_0) = s'_0$ since $s_0$ is the empty string.
- Let $s \in S$, $a \in \Sigma$ and $\delta(s, a) \neq \bot$. Then, by Definition 4, $\phi(\delta(s, a)) = \phi(sa)$. Using the definition of $\phi$, we get $\phi(\delta(s, a)) = \delta'(s'_0, sa)$, which simplifies to $\phi(\delta(s, a)) = \delta'(\delta'(s'_0, s), a) = \delta'(\phi(s), a))$.
- Let $s \in F$. Then, $s \in S_1$. Since $A$ accepts $S_1$, we get $\phi(s) = \delta'(s'_0, s) \in F'$.
- Let $s \in R$. Then, $s \in S_2$. Since $A$ rejects $S_2$, we get $\phi(s) = \delta'(s'_0, s) \in R'$.

□

Thus, to find a separating automaton $A$ of $k$ states, we have only to guess a map from the states of $\textsc{TreeSep}(S_1, S_2)$ to the states of $A$ and construct $A$ accordingly. Now we will show how to construct a folding of the tree $T$ by partitioning its states. If $\Gamma$ is a partition of a set $S$, we will denote by $[s]_\Gamma$ the element of $\Gamma$ containing element $s$ of $S$.

**Definition 6.** *Let $M = (S, \Sigma, s_0, \delta, F, R)$ be an IDFA over $\Sigma$. A consistent partition of $M$ is a partition $\Gamma$ of $S$ such that*

– *for all $s, t \in S$, $a \in \Sigma$, if $\delta(s, a) \neq \bot$ and $\delta(t, a) \neq \bot$ and $[s]_\Gamma = [t]_\Gamma$ then $[\delta(s, a)]_\Gamma = [\delta(t, a)]_\Gamma$, and*
– *for all $s \in F$ and $t \in R$, $[s]_\Gamma \neq [t]_\Gamma$.*

**Definition 7.** *Let $M = (S, \Sigma, s_0, \delta, F, R)$ be an IDFA and let $\Gamma$ be a consistent partition of $M$. The* quotient *$M/\Gamma$ is the IDFA $(\Gamma, \Sigma, s'_0, \delta', A', R')$ such that*

– $s'_0 = [s_0]_\Gamma$,
– $\delta'(s', a) = \sqcup\{[\delta(s, a)]_\Gamma \mid [s]_\Gamma = s'\}$,
– $F' = \{[s]_\Gamma \mid s \in F\}$, *and*
– $R' = \{[s]_\Gamma \mid s \in R\}$.

In the above definition, $\sqcup$ represents the least upper bound in the lattice with partial order $\preceq$ ; containing the bottom element $\bot$, the top element $\top$ and the elements of $\Gamma$; such that for all $s, t \in \Gamma$ if $s \neq t$ then $s \not\preceq t$. Consistency guarantees that the least upper bound is never $\top$.

**Theorem 3.** *Let $T$ be a tree-like IDFA with accepting set $S_1$ and rejecting set $S_2$. There exists an IDFA of $k$ states separating $S_1$ and $S_2$ exactly when $T$ has a consistent partition $\Gamma$ of cardinality $k$. Moreover, $T/\Gamma$ separates $S_1$ and $S_2$.*

*Proof.* Suppose $\Gamma$ is a consistent partition of $T$. It follows that the function $\phi$ mapping $s$ to $[s]_\Gamma$ is a folding of $T$ onto $T/\Gamma$. Thus, by Theorem 2, $T/\Gamma$ is separates $S_1$ and $S_2$, and moreover it has $k$ states. Conversely, suppose $A$ is an IDFA of $k$ states separating $S_1$ and $S_2$. By Theorem 2, there is a folding $\phi$ from $T$ to $A$. Using Definition 5, it can be shown that the partition induced by $\phi$ is consistent and has (at most) $k$ states. □

According to Theorem 3, to find a minimum-state separating automaton for two disjoint finite sets $S_1$ and $S_2$, we have only to construct a corresponding tree-like automaton $T$, and then find the minimum-state consistent partition $\Gamma$ of $T$. The minimum-state separating automaton $A$ is then $T/\Gamma$.

We use a SAT solver to find the minimum-state partition, using the following encoding of the problem of existence of a consistent partition of $k$ states. Let $n = \lceil log_2 k \rceil$. For each state $s \in S$, we introduce a vector of Boolean variables $\bar{v}_s = (v_s^0 \dots v_s^{n-1})$. This represents the number of the partition to which $s$ is assigned (and also the corresponding state of the quotient automaton). We then construct a set of Boolean constraints that guarantee that the partition is consistent. First, for each $s$, we must have $\bar{v}_s < k$ (expressed over the bits of $\bar{v}_s$). Then, for every pair of states $s$ and $t$ that have outgoing transitions on symbol $a$, we have a constraint $\bar{v}_s = \bar{v}_t \Rightarrow \bar{v}_{\delta(s,a)} = \bar{v}_{\delta(t,a)}$ (that is, the partition must respect the transition relation). Finally, for every pair of states $s \in F$ and $t \in R$, we have the constraint $\bar{v}_s \neq \bar{v}_t$ (that is, a rejecting state and an accepting state cannot be put in the same partition). We call this set of constraints $\textsc{SatEnc}(T)$. A truth assignment $\psi$ satisfies $\textsc{SatEnc}(T)$ exactly when the partition $\Gamma = \{\Gamma_0, \ldots, \Gamma_{k-1}\}$ is a consistent partition of $T$ where $\Gamma_i = \{s \in S \mid \bar{v}_s = i\}$. Thus, from a satisfying assignment, we can extract a consistent partition.

Algorithm 2 (SatMSA) outlines our approach for computing a minimum-state separating automaton for two finite languages. Note that the quotient automaton $T/\Gamma$ is an IDFA. We can convert this to a DFA by completing the partial transition function $\delta$ in any way we choose (for example, by making all the missing transitions go to a rejecting state), yielding a DFA that separates $S_1$ and $S_2$.

**Algorithm 2** Computing an MSA for two finite languages, using SAT encoding

```
SatMSA $(S_1, S_2)$
 1: Let $T = \textsc{TreeSep}(S_1, S_2)$;
 2: Let $k = 1$;
 3: while (1) do
 4:     if $\textsc{SatEnc}(T)$ is satisfiable then
 5:         Let $\psi$ be a satisfying assignment of $\textsc{SatEnc}(T)$;
 6:         Let $\Gamma = \{\{s \in S \mid \bar{v}_s = i\} \mid i \in 0 \ldots k-1\}$;
 7:         Let $A = T/\Gamma$;
 8:         Extend $\delta(A)$ to a total function;
 9:         return DFA $A$;
10:     Let $k = k + 1$;
```

This completes the description of our LangMSA procedure for computing an MSA for two languages $L_1$ and $L_2$. To find an intermediate assertion for assume-guarantee reasoning, we have only to compute an MSA for $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2')$.

Let us now consider the overall complexity of assume-guarantee reasoning using the LangMSA algorithm (Algorithm 1). We will assume that $M_1$ and $M_2'$ are expressed symbolically as Boolean circuits with textual size $|M_1|$ and $|M_2'|$ respectively. The number of states of these DFA's is then $O(2^{|M_1|})$ and $O(2^{|M_2'|})$ respectively. Let $|A|$ be the textual size of the MSA (note that this is proportional to both the number of states and the size of $\Sigma$). Each iteration of the main loop involves solving the SAT problem $\textsc{SatEnc}(T)$ and solving two model checking problems. The SAT problem can, in the worst case, be solved by enumerating all the possible DFA's of the given size, and thus is $O(2^{|A|})$. The model checking problems are $O(|A| \times 2^{|M_1|})$ and $O(|A| \times 2^{|M_2'|})$. The number of iterations is at most $2^{|A|}$, the number of possible automata, since each iteration rules out one automaton. Thus the overall run time is $O(2^{|A|}(2^{|A|} + |A| \times (2^{|M_1|} + 2^{|M_2'|})))$.

This is singly exponential in $|A|$, $|M_1|$ and $|M_2'|$, but notably we do not incur the cost of computing the product of $M_1$ and $M_2$. Fixing the size of $A$, we have simply $O(2^{|M_1|} + 2^{|M_2'|})$.

Unfortunately, $|A|$ is worst-case exponential in $|M_1|$, since in the worst case we have $\mathcal{L}(A) = \mathcal{L}(M_1)$. This means that the overall complexity is doubly exponential in the input size. It may seem illogical to apply a doubly exponential algorithm to a PSPACE-complete problem. However, we will observe that in practice, if there is a small intermediate assertion, this approach can be more efficient than singly exponential approaches. In the case when the alphabet is large, however, we will need some way to compactly encode the transition function.

## 5 Generalization with Decision Tree Learning

As mentioned earlier, in hardware verification, the size of the alphabet $\Sigma$ is exponential in the number of Boolean signals passing between $M_1$ and $M_2$. This means that in practice the samples we obtain of $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2')$ can contain only a minuscule fraction of the alphabet symbols. Thus, the IDFA $A$ that we learn will also contain transitions for just a small fraction of $\Sigma$. We therefore need some way to generalize from this IDFA to a DFA over the full alphabet in a reasonable way. This is not a very well-defined problem. In some sense we would like to apply Occam's razor, inferring the *simplest* total transition function that is consistent with the partial transition function of the IDFA. There might be many ways to do this. For example, if the transition from a given state on symbol $a$ is undefined in the IDFA, we could map it to the next state for the nearest defined symbol, according to some distance measure.

The approach we take here is to use decision tree learning methods to try to find the simplest generalization of the partial transition function as a decision tree. Given an alphabet symbol, the decision tree branches on the values of the Boolean variables that define the alphabet, and at its leaves gives the next state of the automaton. We would like to find the simplest decision tree expressing a total transition function consistent with the partial transition function of the IDFA. Put another way, we can think of the transition function of any state as a classifier, classifying the alphabet symbols according to which state they transition to. The partial transition function can be thought of as providing *samples* of this classification and we would like to find the simplest decision tree that is consistent with these samples. Intuitively, we expect the intermediate assertion to depend on only a small set of the signals exchanged between $M_1$ and $M_2$, thus we would like to bias the procedure toward transition functions that depend on few signals. To achieve this, we use the ID3 method for learning decision trees from examples [Qui86].

This allows us (line 8 of Algorithm 2) to generalize the IDFA to a symbolically represented DFA that represents a guess as to what the full separating language should be, based on the samples of the alphabet seen thus far. If this guess is incorrect, the teacher will produce a counterexample that refutes it, and thus refines the next guess.

## 6 Example

We illustrate the working of the SatMSA algorithm on an example. In this example, we are trying to learn a separating automaton for two components which communicate over 3 Boolean signals, namely $x_1, x_2$ and $x_3$. The alphabet $\Sigma$, which consists of all possible assignments to the interface signals, is equal to $\{0,1\}^3$. Figure 1 shows the input sample sets $S_1$ and $S_2$ for the SatMSA algorithm. Each alphabet symbol corresponds to an assignment to $(x_1, x_2, x_3)$.

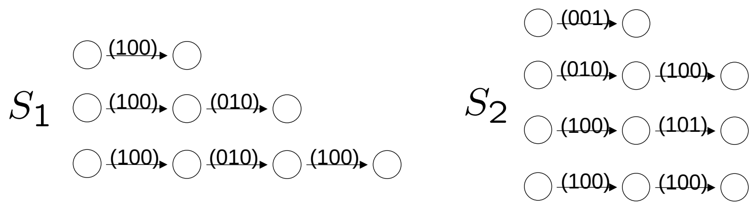

**Fig. 1.** Sample sets $S_1$ and $S_2$ for the SatMSA algorithm.

Figure 2 shows $T$ = TreeSep$(S_1, S_2)$, the tree-like separator for $S_1$ and $S_2$. SatEnc$(T)$ (line 4) is unsatisfiable for $k = 1, 2$ and the SatMSA algorithm finds a satisfying assignment to SatEnc$(T)$ for $k = 3$. The partition number for each state is shown in Figure 3.

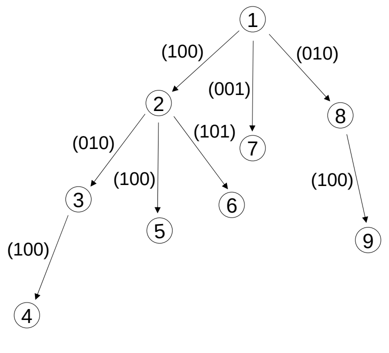

**Fig. 2.** TreeSep$(S_1, S_2)$, corresponding to the sample sets in Figure 1.

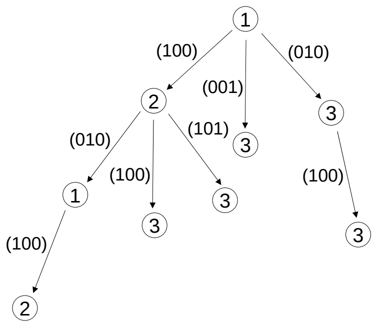

**Fig. 3.** The partitioned TreeSep$(S_1, S_2)$.

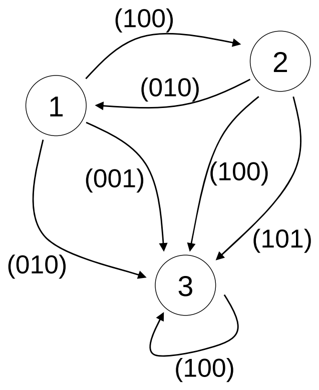

**Fig. 4.** The IDFA generated by SatMSA. State 1 is the initial state. States 1,2 are accepting states while State 3 is a rejecting state.

The partitioning produces the IDFA shown in Figure 4. The sample sets for the decision tree learner, corresponding to the transition functions of the 3 states, are shown in Figure 5. Figure 5 also shows the decision trees generated by ID3. Figure 6 shows the resulting complete DFA.

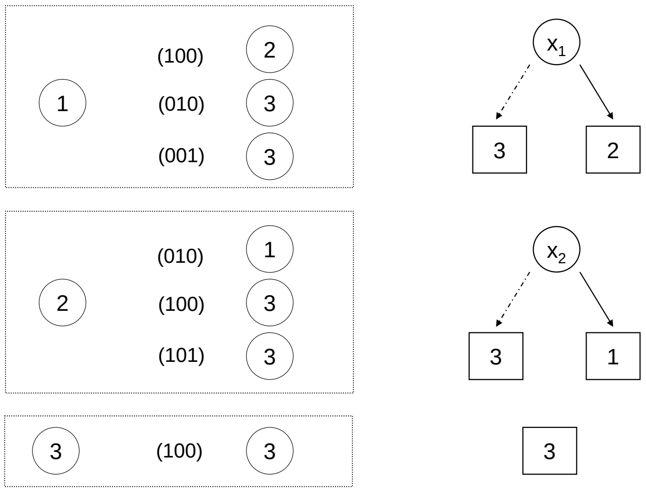

**Fig. 5.** The sample sets for the decision tree learner, and the corresponding decision trees.

## 7 Optimizations

We use two optimizations to the above approach that effectively reduce the size of the search space when finding a consistent partition of $T$. First, we exploit the fact that $\mathcal{L}(M_1)$ is prefix closed in the case of hardware verification (on the other hand $\mathcal{L}(M_2')$ may not be prefix closed, since it includes the negation of the property $P$). This means that if string $\pi$ is in the accepting set of $T$, we can assume that all its prefixes are accepted as well. This allows us to mark the ancestors of any accepting state of $T$ as accepting, thus reducing the space of consistent partitions. In addition, since $M_1$ is prefix closed, it follows that there is a prefix closed intermediate assertion and we can limit our search to prefix closed languages. These languages can always be accepted by an automaton with a single rejecting state. Thus, we can group all the rejecting states into a single partition, again reducing the space of possible partitions.

Our second optimization is to compute the consistent partition incrementally. We note that each new sample obtained as a counterexample from the teacher adds one new branch to the tree $T$. In our first attempt to obtain a partition

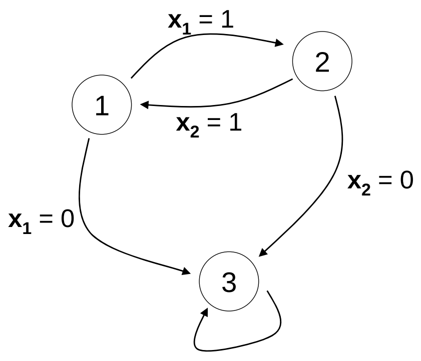

**Fig. 6.** The separating automaton generated by SATMSA. State 1 is the initial state. States 1,2 are accepting states while State 3 is a rejecting state.

we restrict all the pre-existing states of $T$ to be in the same partition as in the previous iteration. Only the partitions of the new states of $T$ can be chosen. This forces us, if possible, to maintain the old behavior of the automaton $A$ for all the pre-existing samples and to change only the behavior for the new sample. If this problem is infeasible, the restriction is removed and the algorithm proceeds as usual. Heuristically, this tends to reduce the SAT solver run time in finding a partition, and also tends to reduce the number of samples, perhaps because the structure of the automaton remains more stable.

## 8 Incorporating Domain Knowledge

If some domain-specific property of the separating automaton is known apriori, it can be incorporated into the LANGMSA algorithm to speed up the convergence. In this paper, we experimented with two such properties, which we describe in the following sections.

### 8.1 Stuttering Closure

**Definition 8.** *A language $L$ over $\Sigma$ is said to be* stuttering closed *if for all $s_1, s_2 \in \Sigma^*$, $a$ in $\Sigma$, if $s_1 a s_2 \in L$, then $s_1(a^+)s_2 \in L$.*

It can be shown that if $L$ is prefix closed and stuttering closed, it is accepted by an automaton that satisfies the following property: if a state $s$ has an incoming transition on an alphabet symbol $a$, then the outgoing transition from $s$ on $a$ is a self-loop. This property can be used to reduce the space of consistent partitions, by adding additional constrains to SATENC($T$) (line 4 of Algorithm 2). The constraints are formalized in the following Lemma.

**Lemma 2.** *Let $M = (S, \Sigma, s_0, \delta, F, R)$ be an IDFA over $\Sigma$. If $\Gamma$ is a consistent partition of $M$ and $M$ accepts a prefix closed and stuttering closed language, then for all $s, t \in S$, $a \in \Sigma$, if $\delta(s, a) = s_1$ and $\delta(t, a) = t_1$, then:*

- *If $[s_1]_\Gamma = [t]_\Gamma$, then $[t]_\Gamma = [t_1]_\Gamma$.*
- *If $[t_1]_\Gamma = [s]_\Gamma$, then $[s]_\Gamma = [s_1]_\Gamma$.*

In addition to reducing the space of consistent partitions, the stuttering closure property can be used to prune repeating sequences of alphabet symbols from the counterexample traces, before the traces are added to the sample sets. This leads to a reduction in size of SatEnc($T$).

### 8.2 Learning Multiple Decision Trees

If the separating automaton has $k$ states, the transition function of each state is a function that maps an assignment to the alphabet variables to one of the $k$ states. As described in Section 5, the motivation for generalization with a decision tree is that it biases the learner towards transition functions that depend on only a few alphabet variables. In our benchmarks, the transition function for a state $s$ depends on all of the alphabet variables. However, the transition from a state $s$ to a state $t$ is predicated with a subset of the variables. In order to exploit this property, we learn multiple decision trees to capture the transition function of a state $s$, one tree for each possible next state $t$. Each decision tree classifies the alphabet symbols into two classes, corresponding to whether or not they transition to the corresponding next state. Since each transition is predicated with only a subset of the variables, the decision tree for a transition is much smaller than the decision tree for the full transition function, and is therefore easier to learn. Moreover, the run-time complexity of learning multiple decision trees is the same as learning a single decision tree. Therefore, our LangMSA algorithm can potentially converge faster by learning multiple decision trees for a state transition function.

## 9 Results

We have implemented our techniques on top of Cadence SMV [McM]. The user specifies a decomposition of the system into two components. We use Cadence SMV as our BDD-based model checker to verify the assumptions, and also as our incremental BMC engine to check whether counterexamples are real. We use an internally developed SAT solver. We implemented a variant of the ID3 [Qui86] algorithm to generate decision trees. We also implemented the L*-based approach (Lstar) proposed by Cobleigh et al. [CGP03], using the optimized version of the L* algorithm suggested by Rivest and Schapire [RS89]. All our experiments were carried on a 3GHz Intel Xeon machine with 4GB memory, running Linux. We used a timeout of 1000s for our experiments. We compared our approach against Lstar, and the Cadence SMV implementation of standard BDD-based model checking and interpolation-based model checking.

We generated two sets of benchmarks for our experiments. For all our benchmarks, the property is true and all the circuit elements are essential for proving the property. Therefore localization-based verification techniques will not be effective. These benchmark sets are representative of the following typical scenario. A component of the system is providing a service to the rest of the system. The system is feeding data into the component and is reading data from the component. The verification task is to ensure that the data flowing through the system is not corrupted. This property can be verified by using a very simple assumption about the component, which essentially states that the component does not corrupt the data.

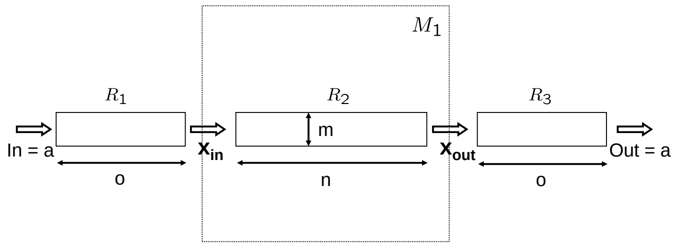

**Fig. 7.** The shift-register based benchmark set.

Each circuit in the first benchmark set consists of a sequence of 3 shift-registers: $R_1$, $R_2$ and $R_3$, such that $R_1$ feeds into $R_2$ and $R_2$ feeds into $R_3$ (Figure 7). The property that we want to prove is that we see some (fixed) symbol $a$ at the output of $R_3$ only if it was observed at the input of $R_1$. We varied the lengths and widths of the shift-registers. Our results are shown in Table 1. For the circuit $S_{m\_n\_o}$, $m$ is the width of the shift-registers, $n$ is the length of $R_2$, and $o$ is the length of $R_1$ and $R_3$. In our decomposition, $M_1$ consists of $R_2$, and $M_2$ consists of $R_1$ and $R_3$. We compare our approach against LSTAR. These benchmarks were trivial (almost 0 s runtime) for BDD-based and interpolation-based model checking. For LSTAR, we report the total running time (Time), the number of model checking calls (Iters), the number of states in the assumption DFA (States), and the number of membership queries (Queries). In case of a timeout, we report the number of states, and queries made, for the last generated DFA. For our approach, we report the total running time (Time), the number of model checking calls (Iters), time spent in model checking (MC), maximum time spent in a model checking run (Max), time spent in counterexample checks (Chk), and the number of states in the assumption DFA (States). On this benchmark set, our approach clearly outperforms LSTAR both in the total runtime and in the size of the assumption automaton. Our approach identifies the 3 state assumption, which says that $a$ can be seen at the output of $M_1$ only if $a$ has been inputted into $M_1$ (Figure

8). LSTAR only terminates on $S_{1\_6\_3}$, where it learns the assumption of size 65, which is the same as $M_1$.

| Circuit | LSTAR | | | | LANGMSA | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Time(s) | Iters | States | Queries | Time(s) | Iters | MC(s) | Max(s) | Chk(s) | States |
| $S_{1\_6\_3}$ | 362.61 | 90 | 65 | 16703 | 0.44 | 9 | 0.34 | 0.04 | 0.00 | 3 |
| $S_{1\_8\_4}$ | (> 1000) | 113 | 80 | 25358 | 0.45 | 9 | 0.31 | 0.04 | 0.00 | 3 |
| $S_{1\_10\_5}$ | (> 1000) | 107 | 76 | 23179 | 0.51 | 9 | 0.36 | 0.05 | 0.00 | 3 |
| $S_{2\_6\_3}$ | (> 1000) | 64 | 45 | 32444 | 1.20 | 27 | 0.99 | 0.04 | 0.01 | 3 |
| $S_{2\_8\_4}$ | (> 1000) | 62 | 43 | 29626 | 1.58 | 27 | 1.36 | 0.07 | 0.01 | 3 |
| $S_{2\_10\_5}$ | (> 1000) | 59 | 40 | 25639 | 1.87 | 27 | 1.57 | 0.08 | 0.01 | 3 |
| $S_{3\_6\_3}$ | (> 1000) | 35 | 24 | 35350 | 7.52 | 91 | 5.42 | 0.17 | 0.03 | 3 |
| $S_{3\_8\_4}$ | (> 1000) | 32 | 22 | 30997 | 14.36 | 90 | 10.46 | 0.27 | 0.03 | 3 |
| $S_{3\_10\_5}$ | (> 1000) | 29 | 21 | 26899 | 27.82 | 90 | 21.77 | 0.85 | 0.04 | 3 |

**Table 1.** Comparison of LANGMSA against LSTAR on shift-register based benchmarks.

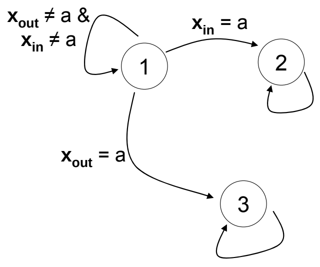

**Fig. 8.** The assumption DFA generated by LANGMSA for our benchmarks. State 1 is the initial state. States 1,2 are accepting states while State 3 is a rejecting state.

For the second benchmark set, we replaced the shift-registers with circular-buffers. We also allowed multiple parallel circular-buffers in $R_2$. Our results are shown in Table 2. For the circuit $C_{m\_n\_o\_p}$, $m$ is the width of the circular-buffers, $n$ is the number of parallel circular-buffers in $R_2$, $o$ is the length of the buffers in $R_2$, and $p$ is the length of $R_1$ and $R_3$. We also report the total running time (Time) of BDD-based model checking. LSTAR and interpolation-based model checking timed-out for all these benchmarks. On this benchmark set, our approach learns the smallest separating assumption and can scale to much larger designs compared to LSTAR, interpolation-based model checking and BDD-based model checking.

| Circuit | BDD | LSTAR | | | LANGMSA | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Time(s) | Iters | States | Queries | Time(s) | Iters | MC(s) | Max(s) | Chk(s) | States |
| $C_{1\_1\_6\_3}$ | 23.46 | 26 | 78 | 22481 | 4.90 | 29 | 4.66 | 0.32 | 0.06 | 3 |
| $C_{1\_1\_8\_4}$ | 160.08 | 26 | 78 | 22481 | 9.66 | 27 | 9.45 | 1.85 | 0.03 | 3 |
| $C_{1\_1\_10\_5}$ | $(> 1000)$ | 26 | 78 | 22481 | 11.42 | 33 | 10.97 | 1.32 | 0.09 | 3 |
| $C_{1\_2\_6\_3}$ | $(> 1000)$ | 26 | 57 | 16433 | 178.65 | 33 | 178.16 | 69.61 | 0.16 | 3 |
| $C_{2\_1\_6\_3}$ | $(> 1000)$ | 20 | 30 | 26893 | 47.20 | 128 | 43.88 | 3.98 | 0.10 | 3 |
| $C_{2\_1\_8\_4}$ | $(> 1000)$ | 20 | 30 | 26893 | 162.82 | 102 | 161.77 | 20.91 | 0.06 | 3 |
| $C_{2\_1\_10\_5}$ | $(> 1000)$ | 20 | 30 | 26893 | 829.46 | 152 | 826.55 | 126.76 | 0.20 | 3 |
| $C_{3\_1\_6\_3}$ | $(> 1000)$ | 16 | 12 | 33802 | 721.94 | 427 | 664.44 | 23.60 | 0.24 | 3 |

**Table 2.** Comparison of LANGMSA against BDD-based model checking and LSTAR on circular-buffer based benchmarks.

Table 3 and Table 4 show the results of incorporating domain knowledge into the LANGMSA algorithm for the two benchmark sets. In these tables, LANGMSAS corresponds to LANGMSA with the stuttering closure assumption, LANGMSAM learns multiple decision trees for a transition function, and LANGMSASM assumes stuttering closure and learns multiple decision trees. The addition of domain knowledge reduces the number of iterations on most of the benchmarks. There are some benchmarks on which this is not the case. This happens because the number of iterations is also sensitive to the specific counterexample generated by the model checker.

| Circuit | LANGMSA | | LANGMSAS | | LANGMSAM | | LANGMSASM | |
|---|---|---|---|---|---|---|---|---|
| | Time(s) | Iters | Time(s) | Iters | Time(s) | Iters | Time(s) | Iters |
| $S_{1\_6\_3}$ | 0.44 | 9 | 0.94 | 7 | 0.47 | 9 | 0.97 | 7 |
| $S_{1\_8\_4}$ | 0.45 | 9 | 1.06 | 7 | 0.46 | 9 | 0.92 | 7 |
| $S_{1\_10\_5}$ | 0.51 | 9 | 0.64 | 7 | 0.46 | 9 | 0.67 | 7 |
| $S_{2\_6\_3}$ | 1.20 | 27 | 2.83 | 21 | 0.69 | 15 | 1.34 | 11 |
| $S_{2\_8\_4}$ | 1.58 | 27 | 3.26 | 21 | 0.83 | 15 | 1.40 | 11 |
| $S_{2\_10\_5}$ | 1.87 | 27 | 3.05 | 21 | 0.90 | 15 | 1.48 | 11 |
| $S_{3\_6\_3}$ | 7.52 | 91 | 14.94 | 81 | 1.54 | 26 | 1.96 | 16 |
| $S_{3\_8\_4}$ | 14.36 | 90 | 29.18 | 83 | 2.32 | 26 | 2.01 | 16 |
| $S_{3\_10\_5}$ | 27.82 | 90 | 46.00 | 83 | 3.04 | 26 | 2.26 | 16 |

**Table 3.** Effect of adding domain knowledge to LANGMSA on shift-register based benchmarks.

Our approach generates the smallest assumption DFA for a verification proof based on assume-guarantee reasoning. However, if the property is false, no such

| Circuit | LangMSA | | LangMSAs | | LangMSAm | | LangMSAsm | |
|---|---|---|---|---|---|---|---|---|
| | Time(s) | Iters | Time(s) | Iters | Time(s) | Iters | Time(s) | Iters |
| $C_{1\_1\_6\_3}$ | 4.90 | 29 | 2.36 | 25 | 3.23 | 36 | 2.48 | 15 |
| $C_{1\_1\_8\_4}$ | 9.66 | 27 | 5.00 | 27 | 2.72 | 20 | 4.25 | 15 |
| $C_{1\_1\_10\_5}$ | 11.42 | 33 | 5.93 | 26 | 4.51 | 19 | 4.17 | 18 |
| $C_{1\_2\_6\_3}$ | 178.65 | 33 | 170.21 | 25 | 164.93 | 26 | 148.70 | 15 |
| $C_{2\_1\_6\_3}$ | 47.20 | 128 | 29.47 | 97 | 12.20 | 56 | 6.48 | 27 |
| $C_{2\_1\_8\_4}$ | 162.82 | 102 | 75.51 | 119 | 12.40 | 47 | 54.57 | 118 |
| $C_{2\_1\_10\_5}$ | 829.46 | 152 | 174.78 | 103 | 88.04 | 50 | 21.62 | 44 |
| $C_{3\_1\_6\_3}$ | 721.94 | 427 | 564.67 | 371 | 40.04 | 100 | 32.56 | 82 |

**Table 4.** Effect of adding domain knowledge to LangMSA on circular-buffer based benchmarks.

assumption DFA exists. In this case, our approach will eventually identify a counterexample to the property. However, a drawback of our algorithm is that it does not provide any bounds on the size of the DFA that it will generate before it identifies the counterexample.

## 10 Conclusion and Future Work

We have presented an automated approach for assume-guarantee reasoning that generates the smallest assumption DFA. Our experiments indicate that this technique can outperform existing L*-based approaches for computing an assumption automaton that is not guaranteed to be minimum-state. For many of our benchmarks, our approach performed better than state-of-the-art non-compositional methods as well. We also illustrated how domain knowledge can be incorporated into our algorithm.

There are many directions for future research. Some of the questions that we want to answer are: (1) Our framework only uses equivalence queries. Can membership queries be used for enhancing our technique? (2) Which generalization techniques (besides decision tree learning) would be effective in our framework? (3) Can we learn a parallel composition of DFAs?

## References

[AMN05] Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic compositional verification by learning assumptions. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 548–562, 2005.

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *In Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, LNCS, 1999.

[CGP03] J. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning assumptions for compositional verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2003.

[Gol78] E. Mark Gold. Complexity of automaton identification from given data. *Information and Computation*, 37:302–320, 1978.

[KVBSV97] T. Kam, T. Villa, R. Brayton, and A. L. Sangiovanni-Vincentelli. *Synthesis of FSMs: Functional Optimization*. Kluwer Academic Publishers, 1997.

[McM] K. L. McMillan. Cadence SMV. Cadence Berkeley Labs, CA.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

[Mit97] Tom M. Mitchell. *Machine Learning*. WCB/McGraw-Hill, 1997.

[OS98] Arlindo L. Oliveira and Joao P. Marques Silva. Efficient search techniques for the inference of minimum size finite automata. In *Proceedings of the Symposium on String Processing and Information Retrieval (SPIRE)*, pages 81–89, 1998.

[Pfl73] C. F. Pfleeger. State reduction in incompletely specified finite state machines. *IEEE Transactions on Computers*, C-22:1099–1102, 1973.

[PO99] Jorge M. Pena and Arlindo L. Oliveira. A new algorithm for exact reduction of incompletely specified finite state machines. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 18(11):1619–1632, 1999.

[Qui86] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1986.

[RS89] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 411–420, New York, NY, USA, 1989. ACM Press.