Automated Assumption Generation for Compositional Verification

Anubhav Gupta[1] K. L. McMillan[1] Zhaohui Fu[2]

[1]Cadence Berkeley Labs
[2]Department of Electrical Engineering, Princeton University

Abstract. We describe a method of computing an exact minimal automaton to act as an intermediate assertion in assume-guarantee reason, 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.

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 this inference rule:

$$\frac{\langle true \rangle\ M_1\ \langle A \rangle \qquad \langle A \rangle\ M_2\ \langle P \rangle}{\langle true \rangle\ M_1 \parallel M_2\ \langle P \rangle} \tag{1}$$

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. 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 its intersection with $\mathcal{L}(\neg P)$ is empty. The above inference rule can thus be written as:

$$\frac{\mathcal{L}(M_1) \subseteq \mathcal{L}(A) \qquad \mathcal{L}(A) \cap \mathcal{L}(M_2) \cap \mathcal{L}(\neg P) = \emptyset}{\mathcal{L}(M_1) \cap \mathcal{L}(M_2) \cap \mathcal{L}(\neg P) = \emptyset} \tag{2}$$

Let us designate the intersection of $\mathcal{L}(M_2)$ and $\mathcal{L}(\neg P)$ as M_2'. 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 reduceable to the problem of finding a minimal-state implementation of an incompletely specified finite state machine (ISFSM), shown to be NP-complete by Pfleer [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; in the worst case, the algorithm will return the trivial solution $\mathcal{L}(M_1)$ as the separating language, and thus provide no benefit in terms of state space reduction that could not be obtained by simply minimizing M_1. In fact, in our experiments with hardware verification problems, the 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 minimal 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. Moreover, the goal of assume-guarantee reasoning is a verification procedure with complexity proportional to $|M_1| + |M_2'|$ rather than $|M_1| \times |M_2'|$. 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 minimal separating automaton problem, suited to assume-guarantee reasoning for hardware verification. We apply the sampling-based algorithm used by Oliveira, *et al.*, for the ISFSM minimization problem [PO98]. This algorithm iteratively generates sample strings in $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2')$, computing at each step a minimal automaton consistent with the sample set. Finding a minimal automaton consistent with a set of labelled strings is itself an NP-complete problem [Gol78], and we solve it using a Boolean SAT solver. We use the sampling approach here because the standard techniques for solving the ISFSM 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.

Using a collection of synthetic hardware benchmarks, we show that this approach is effective in producing exact minimal intermediate assertions in cases where the approximate L* approach yields no reduction. In some cases, this 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) Σ is a finite alphabet, (3) $\delta : S \times \Sigma \to S$ 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) Σ is a finite alphabet, (3) $\delta : S \times \Sigma \to (S \cup \{\perp\})$ 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 $\delta(s, a) = \perp$. For both DFA's and IDFA's we extend the transition function δ 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 it 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 M if s 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 *minimal separating automaton* (MSA) for L_1 and L_2 is an automaton with minimal 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 mininimal DFA A for an unknown regular language L by posing *queries* to a *teacher*. In a membership query, the learner provides a string π, 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 slightly to learn a separating automaton for two languages L_1 and L_2. This differs from the L* algorithm only 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 π, the teacher responds yes if $\pi \in L_1$ and no if $\pi \in L_2$. If π is in neither L_1 nor L_2, the choice is arbitrary. Since the teacher does not know the minimal separating automaton, it cannot provide the correct answer, so it simply answers no. 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. Using Angluin's algorithm for the learner, we can show that the learned separating automaton A has no more states that the minimal automaton for L_1. This can, however, be arbitrarily larger than the mimimal 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 nondeterministic finite automata (NFA's) represented symbolically (the nondeterminism 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). Thus, in practice the execution time of the algorithm is singly exponential.

4 Solving the minimal separating automaton problem extactly

To find an exact MSA for two languages L_1 and L_2, we will follow the general approach of Pena and Oliveira [PO98] for minimizing ISFSM's. This is a learning approach that uses only equivalence queries. It relies on a subroutine that can compute a minimal DFA separating two *finite* sets of strings. Although Pena and Oliveira's work is limited to finite state machines, that 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 the procedure is shown in Algorithm 2. 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 minimal DFA A that separates S_1 and S_2 (using a procedure described below). The learner then performs an equivalence query on A. If A separates L_1 and L_2, the procedure terminates. Otherwise, we obtain a counterexample string π from the teacher. If $\pi \in L_1$ (and consequenctly, $\pi \notin \mathcal{L}(A)$) we add π to S_1, else we add π to S_2. 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. This order is arbitrary, and in practice we choose the order randomly for each query to avoid biasing the result toward a larger or smaller language.

The teacher in this procedure can be implemented using a model checker. That is, the checks $L_1 \subseteq \mathcal{L}(A)$ and $\mathcal{L}(A) \cap L_2 = \emptyset$ 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).

Theorem 1. *Let $L_1, L_2 \subseteq \Sigma^*$, for finite Σ. If L_1 and L_2 have a regular separator, then Algorithm* AsGr *terminates and outputs a minimal separating automaton for L_1 and L_2.*

Proof. Let A' be a minimal-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 minimal separating automaton for S_1 and S_2). Thus, if the procedure terminates, A is a minimal separating automaton for L_1 and L_2. Moreover, there are finitely many DFA's over finite Σ 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 minimal separating automaton for the finite languages S_1 and S_2. This problem has been studied extensively, and is known to be NP-complete [Gol78]. To solve it, we will borrow from the approach of [PO98].

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 DFA $(S, \Sigma, s_0, \delta, F, R)$ where S is the set of prefixes of $S_1 \cup S_2$, s_0 is the empty string, $F = S_1$ and $R = S_2$, and $\delta(\pi, a) = \pi a$ if $\pi a \in (S_1 \cup S_2)$ else $\perp$.*

Oliveira and Silva [OS98] showed that every IDFA A that separates S_1 and S_2 is homomorphic to $\textsc{TreeSep}(S_1, S_2)$ in a sense we will define. 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. We will call this process *folding*.

Definition 5. *Let M and M' be two IDFA's over alphabet Σ. 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 $\delta(s, a) \neq \perp$ then $\delta(\phi(s), a) = \phi(\delta(s, a))$*
- *for all $s \in F$, $\phi(s) \in F'$ and*
- *for all $s \in R$, $\phi(s) \in R'$.*

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

Lemma 1 (Oliveira and Silva). *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 IDFA A over Σ is a separating automaton for S_1 and S_2 if and only if there exists a folding ϕ from T to A.*

Now we will show how to construct a folding of the tree T by paritioning its states. If Γ is a partition of a set S, we will denote by $[s]_\Gamma$ the element of Γ containing element s of S.

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

- *for all $s, t \in S$, $a \in \Sigma$, if $\delta(s, a) \neq \perp$ and $\delta(t, a) \neq \perp$ and $[s]_\Gamma = [t]_\Gamma$ then $[\delta(s, a)]_\Gamma = [\delta(t, a)]_\Gamma$, and*
- *for all states $s \in F$ 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 Γ be a consistent partition of S. The* quotient *M/Γ is the IDFA $(\Gamma, \Sigma, s'_0, \delta', A', R')$ such that*

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

In the above, $\sqcup$ represents the least upper bound in the lattice containing $\perp$, $\top$ and the elements of S. Consistency guarantees that the least upper bound is never $\top$.

Theorem 2. *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 parition Γ of cardinality k. Moreover, T/Γ separates S_1 and S_2.*

Proof. Suppose Γ is a consistent parition of $S(T)$. It follows that the function ϕ mapping s to $[s]_\Gamma$ is a folding of T onto T/Γ. Thus, by the lemma, T/Γ 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 the lemma, there is a folding ϕ from T to A. By the definition of folding, the partition induced by ϕ is consistent and has (at most) k states. □.

According to this theorem, to find a minimal 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 minimal consistent parition Γ of $S(T)$. The minimal automaton A is then T/Γ.

We use a SAT solver to find the minimal parition, using the following encoding of the problem of existence of a consistent parition of k states. Let $n = \lceil log_2 k \rceil$. This is the number of bits needed to enumerate the partitions. For each state $s \in S(T)$, 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 $\bar{v}_s < n$ (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 and an accepting state cannot be put in the same parition). Call this set of constraints SatEnc(T). A truth assignment ψ satisfies SatEnc(T) exactly when the partition $\Gamma = \{\Gamma_0, \ldots, \Gamma_{n-1}\}$ 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 consisitent partition.

A minimal separating automaton for finite sets S_1 and S_2 can be found by Algorithm 2. Note that the quotient automaton T/Γ is an IDFA. We can convert this to a DFA by completing the partial transition function δ in any way we choose (for example, by making all the missing transitions go to a rejecting state), yielding an DFA that separates S_1 and S_2. This completes our procedure for computing an MSA of two languages L_1 and L_2.

Algorithm 1 Computing MSA for finite languages, using SAT encoding

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

To find an intermediate assertion for assume-guarantee reasoning, we have only to compute an MSA for $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2)$, using algorithm 2.

Let us now consider the overall complexity of of this procedure. 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 machines is then $O(2^{|M_1|})$ and $O(2^{|M_2'|})$ respectively. Let $|A|$ be the textual size of the MSA (note this is proportional to both the number of states and the size of Σ). Each iteration of the main loop involves solving the SAT problem 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

Algorithm 2 Our overall procedure

```
AsGr (L1, L2)
 1: S1 = {}; S2 = {};
 2: while (1) do
 3:     Let A be an MSA for S1 and S2;
 4:     if L1 ⊆ L(A) then
 5:         if L(A) ∩ L2 = ∅ then
 6:             return true; (A separates L1 and L2, property holds)
 7:         else
 8:             Let π ∈ L2 and π ∈ L(A); (negative counterexample)
 9:             if π ∈ L1 then
10:                 return false; (L1 and L2 not disjoint, property fails)
11:             else
12:                 S1 = S1 ∪ {π};
13:     else
14:         Let π ∈ L1 and π ∉ A; (positive counterexample)
15:         if π ∈ L2 then
16:             return false; (L1 and L2 not disjoint, property fails)
17:         else
18:             S2 = S2 ∪ {π};
```

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.

4.1 Optimizations

We use two optimizations to the above approach that effectively the size of the search space when finding a consistent parition of T. First, we exploit the fact that $\mathcal{L}(M_1)$ is prefix closed, since it it is the language of a finite state machine (on the other hand $\mathcal{L}(M_2')$ may not be prefix closed, since it includes the negation of a the property P). This menas thath if string π 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 partitions, 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 parition we restrict all the pre-existings states of T to be in the same partition as in the previous iteration. Only the paritions of the new states of T can be chosen. This forces us, if possible, to keep the 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 parition, and also tends to reduce the number of samples, perhaps because the structure of the automaton remains more stable.

5 Generalization with Decision Tree Learning

As mentioned earlier, in hardware verification, the size of alphabet Σ 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 miniscule fraction of the alphabet symbols. Thus, the IDFA A that we learn will also contain transitions for just a small fraction of Σ. 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 partion 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 paritial 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 transiton function can be thought of as providing "samples" of this classifaction 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 betweeen M_1 and M_2, thus we would like to bias the procedure towards 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, in line 8 of Algorithm 1 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 sample 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 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 as our incremental BMC engine to check the counterexamples. 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 out 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. In our benchmarks, all the circuit elements are essential for proving the property, and therefore localization-based verification techniques will not be effective. These benchmark sets are representative of the following typical scenario. A componenet of the system is providing a service to the rest of the system. The system if feeding data into this componenet and is reading data from the component, and the data is tagged. 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. For example, consider a processor and memory communicating over a bus. In order to prove the correctness of some instruction sequence, the only information that is needed is that the bus tranfers the data correctly. Any buffering or arbitration that happens on the bus is irrelevant.

Each circuit in the first benchmark set consists of a sequence of 3 shift registers, $R1$, $R2$ and $R3$, such that $R1$ feeds into $R2$ and $R2$ feeds into $R3$. The property that we want to prove is that we see some (fixed) symbol a at the output of $R3$ only if it was observed at the input of $R1$. We varied the lengths and widths of these registers. Our results are shown in Table 1. For the circuit S_$n1$_$n2$_$n3$, $n1$ is the width of the shift registers, $n2$ is the length of $R2$ and, $n3$ is the length of $R1$ and $R3$. In our decomposition, M_1 consists of $R1$ and $R3$, and $M2$ consists of $R2$. We compare our approach againt LSTAR. These benchmarks were trivial (almost 0s runtime) for BDD-based and interpolation-based model checking. For LSTAR, we report the total running time (Time), the number of states in the assumption DFA (States) and the number of membership queries (Queries). For our approach, we report the number of model checking calls (Iter), time spent in model checking (MC), maximum time spent in a model checking run (Max), time spent in counterexample checks (Chk), number of states in the assumption DFA (States), and the total running time (Time). A '_' symbol indicates a timeout. 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_2 only if a has been inputted into M_2. LSTAR only terminates on S_1 _6 _3, where it learns the assumption of size 65, which is the same as M_2.

Circuit	LSTAR			Minimal Separating Automaton					
	Time(s)	States	Queries	Iter	MC(s)	Max(s)	Chk(s)	States	Time(s)
S_1_6_3	338.81	65	16703	9	0.28	0.04	0.00	3	0.35
S_1_8_4	_	80	25679	9	0.37	0.04	0.00	3	0.44
S_1_10_4	_	78	24413	9	0.28	0.04	0.00	3	0.37
S_2_6_3	_	45	32444	27	1.31	0.04	0.01	3	1.29
S_2_8_4	_	43	29626	27	1.56	0.08	0.01	3	1.77
S_2_10_4	_	41	26936	27	1.83	0.09	0.01	3	2.11
S_3_6_3	_	24	35350	91	5.46	0.09	0.03	3	7.48
S_3_8_4	_	22	30997	90	10.68	0.28	0.03	3	14.36
S_3_10_4	_	21	26899	90	21.39	0.69	0.04	3	27.23

Table 1. Comparison of our approach against L* on simple shift register based benchmarks.

For the second benchmark set, we replaced the shift registers with circular buffers. We also allowed multiple (parallel) circular buffers in $R2$. Our results are shown in Table 2. For the circuit C_$n1$_$n2$_$n3$_$n4$, $n1$ is the width of the circular buffers, $n2$ is the number of (parallel) circular buffers in $R2$, $n3$ is the length of the buffers in $R2$ and, $n4$ is the length of $R1$ and $R3$. In the table, we have all the total running time (Time) of BDD-based model checking. LSTAR and interpolation-based model checking did not finish for all these benchmarks. On this benchmark set, our approach learns the smallest separating assumption and can scale to much larger designs comared to LSTAR and the state-of-the-art model checking techniques.

7 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 minimal. For our benchmark sets that were ideal for assume-guarantee reasoning, our approach performed better than state-of-the-art non-compositional methods as well.

There are many directions for future research: (1) Our framework only uses equivalence queries. Can membership queries be used for enhancing our technique? (2) Can the performance of our algorithm be improved by imposing additional restrictions on the assumption? For example, if we assume that the assumption language is stuttering closed, it can prune out long repeating sequences

Circuit	BDD	Lstar		Minimal Separating Automaton					
	Time(s)	States	Queries	Iter	MC(s)	Max(s)	Chk(s)	States	Time(s)
C_1_1_6_3	23.61	78	22481	29	2.09	0.17	0.05	3	2.42
C_1_1_8_4	198.36	78	22481	27	2.84	0.21	0.05	3	3.09
C_1_1_10_5	_	78	22481	33	3.99	0.42	0.89	3	4.41
C_1_2_6_3	_	57	16433	33	8.68	3.43	0.76	3	8.96
C_1_2_8_4	_	57	16433	26	531.92	521.89	0.05	3	532.14
C_2_1_6_3	_	30	26893	128	21.27	0.52	0.10	3	23.55
C_2_1_8_4	_	30	26893	102	25.62	3.21	0.06	3	26.48
C_2_1_10_5	_	30	26893	152	63.39	5.75	0.17	3	65.79
C_3_1_6_3	_	12	33802	427	569.50	19.90	0.23	3	622.15

Table 2. Comparison of our approach against L* and BDD-based model checking on circular buffer based benchmarks.

from the counterexamples. (3) Which generalization techniques (besides decision tree learning) would be effective in out framework? (4) Can we learn a parallel composition of DFAs?

References

[Ang87] D. Angluin. Learning regular sets from queries and counter-examples. *Inf. and Cont.*, 75:87–106, 1987.

[CGP03] J. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning assumptions for compositional verification, 2003.

[Gol78] E. Mark Gold. Complexity of automaton identification from given data. *Inf. and Cont.*, 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] Ken 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 *String Processing and Information Retrieval*, 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.

[PO98] Jorge M. Pena and Arlindo L. Oliveira. A new algorithm for the reduction of incompletely specified finite state machines. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 482–489, New York, NY, USA, 1998. ACM Press.

[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 *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 411–420, New York, NY, USA, 1989. ACM Press.