SAT-based Compositional Verification using Lazy Learning ⋆

Nishant Sinha, Edmund Clarke
Carnegie Mellon University, USA
{nishants,emc}@cs.cmu.edu

Abstract. A recent approach to automated assume-guarantee reasoning (AGR) for concurrent systems relies on computing environment assumptions for components using the L^* algorithm for learning regular languages. While this approach has been investigated extensively for message passing systems, it still remains a challenge to scale the technique to large shared memory systems, mainly because the assumptions have an exponential communication alphabet size. In this paper, we propose a SAT-based methodology that employs both induction and interpolation to implement automated AGR for shared memory systems. The method is based on a new *lazy* approach to assumption learning, which avoids an explicit enumeration of the exponential alphabet set during learning by using symbolic alphabet clustering and iterative counterexample-driven localized partitioning. Preliminary experimental results on benchmarks in Verilog and SMV are encouraging and show that the approach scales well in practice.

1 Introduction

Verification approaches based on compositional reasoning allow us to prove properties (or discover bugs) for large concurrent systems in a divide-and-conquer fashion. Assume-guarantee reasoning (AGR) [21, 18, 24] is a particular form of compositional verification, where we first generate environment assumptions for a component and discharge them on its environment (i.e., the other components). The primary bottleneck is that these approaches require us to manually provide appropriate environment assumptions. Recently, an approach [12] has been proposed to automatically generate these assumptions using learning algorithms for regular languages assisted by a model checker. Consider an AGR rule, called **NC**. This rule states that given finite state systems M_1, M_2 and P, the parallel composition $M_1 \parallel M_2$ satisfies P (written as $M_1 \parallel M_2 \vDash P$) iff there exists an *environment assumption* A for M_1 such that the composition of M_1 and A satisfies P ($M_1 \parallel A \vDash P$) and M_2 satisfies A ($M_2 \vDash A$). It is known that if M_1 and P are finite-state (their languages are regular), then a finite state assumption A exists. Therefore, the task of computing A is cast as a machine learning problem, where an algorithm for learning regular languages L^* [6, 26]

⋆ This research was sponsored by the National Science Foundation under grant nos. CNS-0411152, CCF-0429120, CCR-0121547, and CCR-0098072, the US Army Research Office under grant no. DAAD19-01-1-0485, the Office of Naval Research under grant no. N00014-01-1-0796, the Defense Advanced Research Projects Agency under subcontract no. SA423679952, the General Motors Corporation, and the Semiconductor Research Corporation grant no. TJ-1366.

is used to automatically compute A. The L^* *learner* computes a deterministic finite automaton (DFA) corresponding to an unknown regular language by asking queries to a *teacher* entity, which is capable of answering membership (whether a trace belongs to the desired assumption) and candidate (whether the current assumption hypothesis is correct) queries about the unknown language. Using these queries, the learner improves its hypothesis DFA using iterative state-partitioning (similar to the DFA minimization algorithms [17]) until the teacher replies that a given hypothesis is correct. In our context, a model checker plays the role of the teacher. It answers the queries by essentially checking the two premises of the rule **NC** with respect to the a given hypothesis A. While this approach is effective for small systems, there are a number of problems in making it scalable:

– *Efficient Teacher Implementation:* The teacher, i.e., the model checker, must be able to answer membership and candidate queries efficiently. More precisely, each query may itself involve exploration of a large state space making explicit-state model checking infeasible.
– *Alphabet explosion:* If M_1 and M_2 interact using a set X of global shared communication variables (referred to as a *shared memory system* subsequently), the alphabet of the assumption A consists of all the valuations of X and is exponential in size of X. The learning algorithm explicitly enumerates the alphabet set at each iteration and performs membership queries for enumeration step. Therefore, it is prohibitively expensive to apply L^* directly to shared memory systems with a large number of shared communication variables. Indeed, it is sometimes impossible to enumerate the full alphabet set, let alone learning an assumption hypothesis. We refer to this problem as the *alphabet explosion* problem.
– *System decomposition:* The natural decompositions of a system according to its modular syntactic description may not be suitable for compositional reasoning. Therefore, techniques for obtaining good decompositions automatically are required.

In this work we address the first two problems. More precisely, we propose (i) to efficiently implement the teacher using SAT-based model checking; and (ii) a *lazy* learning approach for mitigating the alphabet explosion problem. For an approach dealing with the third problem, see, for instance, the work in [22].

SAT-based Teacher. In order to allow the teacher to scale to larger models, we propose to implement it using a SAT-based symbolic model checker. In particular, we use SAT-based bounded model checking (BMC) [9] to process both membership and candidate queries. BMC is effective in processing membership queries, since they involve unrolling the system transition relation to a finite depth (corresponding to the given trace t) and require only a Boolean answer. The candidate queries, instead, require performing unbounded model checking to show that there is no counterexample for any depth. Therefore, we employ complete variants of BMC to answer the candidate queries. In particular, we have implemented two different variants based on k-induction [27] and interpolation [20] respectively. Moreover, we use a SMT solver as the main decision procedure [29, 3].

Lazy Learning. The main contribution of our work is a *lazy* learning algorithm l^* which tries to ameliorate the alphabet explosion problem. The lazy approach avoids an expensive eager alphabet enumeration by *clustering* alphabet symbols and exploring transitions on these clusters symbolically. In other words, while the states of the assumption are explicit, each transition corresponds to a set of alphabet symbols, and is explored symbolically. The procedure for learning from a counterexample ce obtained from the teacher is different: besides partitioning the states of the previous hypothesis as in the L^* algorithm, the lazy algorithm may also partition an alphabet cluster (termed as *cluster-partitioning*) based on the analysis of the counterexample. Note that since our teacher uses a SAT-based symbolic model checker, it is easily able to answer queries for traces where each transition corresponds to a set of alphabet symbols. Moreover, this approach is able to avoid the quantifier elimination step (expensive with SAT) that is used to compute the transitions in an earlier BDD-based approach to AGR [4]. We have developed several optimizations to l^*, including a SAT-based counterexample generalization technique that enables coarser cluster partitions.

Our hope, however, is that in real-life systems where compositional verification is useful, we will require only a few state and cluster partitions until we converge to an appropriate assumption hypothesis. Indeed if the final assumption has a small number of states and its alphabet set is large, then there must be a large number of transitions between each pair of states in the assumption which differ only on the alphabet label. Therefore, a small number of cluster partitions should be sufficient to distinguish the different outgoing clusters from each state. Experiments based on the earlier BDD-based approach to AGR [4, 22] as well as our approach have confirmed this expectation.

We have implemented our SAT-based compositional approach in a tool called Symoda (stands for SYmbolic MODular Analyzer). The tool implements SAT-based model checking algorithms based on k-induction and interpolation together with the lazy learning algorithms presented in this paper. Preliminary experiments on Verilog and SMV examples show that our approach is effective as an alternative to the BDD-based approach in combating alphabet explosion and is able to outperform the latter on some examples.

Related Work. Compositional verification based on learning was proposed by Cobleigh et al. [12] in the context of rendezvous-based message passing systems and safety properties using explicit-state model checking. It has been extended to to shared memory systems using symbolic algorithms in [4, 22]. The problem of whether it is possible to obtain good decompositions of systems for this approach has been studied in [11]. An overview of other related work can be found in [16, 22, 28]. SAT-based bounded model checking for LTL properties was proposed by Biere et al. [9] and several improvements, including techniques for making it complete have been proposed [25, 5]. All the previous approaches are non-compositional, i.e., they build the transition relation for the whole system. To the best of our knowledge, our work in the first to address automated compositional verification in the setting of SAT-based model checking.

The symbolic BDD-based AGR approach [4] for shared memory systems using automated system decomposition [22] is closely related to ours. The technique uses a BDD-based model checker and avoids alphabet explosion by using eager state-partitioning to introduce all possible new states in the next assumption, and by computing the transition relation (edges) using BDD-based quantifier elimination. In contrast, we use a SAT-based model checker and our lazy learning approach does not require a quantifier elimination step, which is expensive with SAT. Moreover, due to its eager state-partitioning, the BDD-based approach may introduce unnecessary states in the assumptions. Two other approaches to improve learning based on alphabet set underapproximation and iterative enlargement have been proposed [10, 16]. Our lazy approach is complementary and can learn assumptions effectively in cases where a small alphabet set is not sufficient. Further, it is possible to combine the previous approach with ours by removing variables from assumption alphabets and adding them back iteratively. Finally, a learning algorithm for parameterized systems (where the alphabet consists of a small set of basis symbols, each of which is parameterized by a set of boolean variables) was proposed in [8]. Our lazy algorithm, in contrast, performs queries over a set of traces using a SAT-based model checker and performs more efficient counterexample analysis.

2 Notation and Preliminaries

We define the notions of symbolic transition systems, automata, and composition which we will use in the rest of the paper. Our formalism borrows notation from [23, 19]. Let $X = \{x_1, \ldots, x_n\}$ be a finite set of typed variables defined over a non-empty finite domain of values $\mathcal{D}$. We define a *label* a as a total map from X to $\mathcal{D}$ which maps each variable x_i to value d_i. An X-*trace* ρ is a finite sequence of labels on X. The next-time label is $a' = a\langle X/X' \rangle$ is obtained from a by replacing each $x_i \in dom(a)$ by x'_i. Given variables X and the corresponding next-time variables X', let us denote the (finite) set of all predicates on $X \cup X'$ by Φ_X (TRUE and FALSE denote the boolean constants). Given labels a and b on X, we say that a label pair (a, b') satisfies a predicate $\phi \in \Phi_X$, denoted $\phi(a, b')$, if ϕ evaluates to TRUE under the variable assignment given by a and b'.

CFA. A *communicating finite automata* (CFA) C on a set of variables X (called the support set) is a tuple $\langle X, Q, q0, \delta, F \rangle$; Q denotes a finite set of states, $q0$ is the initial state, $\delta \subseteq Q \times \Phi_X \times Q$ is the transition relation and F is the set of final states. For states $q, q' \in Q$ and $\phi \in \Phi_X$, if $\delta(q, \phi, q')$ holds, then we say that ϕ is a transition predicate between q and q'. For each state q, we define its follow set *fol*(q) to be the set of outgoing transition predicates, i.e., $fol(q) = \{\phi | \exists q' \in Q.\ \delta(q, \phi, q')\}$. We say that *fol*$(q)$ is complete iff $\bigvee\{\phi \in fol(q)\}$ = TRUE and disjoint iff for all $\phi_i, \phi_j \in fol(q)$, $\phi_i \wedge \phi_j =$ FALSE. Also, we say that δ is complete (deterministic) iff for each $q \in Q$, *fol*(q) is complete (disjoint). The alphabet Σ of C is defined to be the set of label pairs (a, a') on variables X and X'. The above definition of transitions (on current and next-time variables) allows compact representation of CFAs and direct composition with STSs below.

A *run* of C is defined to be a sequence $(q_0, \ldots, q_n)$ of states in Q such that $q_0 = q0$. A run is said to be accepting if $q_n \in F$. Given a W-trace $(X \subseteq W)$, $\rho = a_0, \ldots, a_n$, is said to be a trace of C if there exists an accepting run $(q_0, \ldots, q_n)$ of C, such that for all $j < n$, there exists a predicate ϕ, such that $\delta(q_j, \phi, q_{j+1})$ and $\phi(a_j, a'_{j+1})$ holds. In other words, the labels a_j and a_{j+1} must satisfy some transition predicate between q_j and q_{j+1}. The W-trace language $\mathbb{L}_W(C)$ is the set of all W-traces of C. Note that this definition of W-trace allows a sequence of labels on X to be *extended* by all possible valuations of variables in $W \setminus X$ and eases the definition of the composition operation below. In general, we assume W is the universal set of variables and write $\mathbb{L}(C)$ to denote the language of C.

A CFA can be viewed as an ordinary finite automaton with alphabet Σ which accepts a regular language over Σ. While the states are represented explicitly, the *follow* function allows clustering a set of alphabet symbols into one transition symbolically. The common automata-theoretic operations, viz., union, intersection, complementation and determinization via subset-construction can be directly extended to CFAs. The complement of C is denoted by $\overline{C}$, where $\mathbb{L}(\overline{C}) = \overline{\mathbb{L}(C)}$. An illustration of a CFA is given in the extended version [28].

Symbolic Transition System. A *symbolic transition system* (STS) M is a tuple $\langle X, S, I, R, F \rangle$, defined over a set of variables X called its *support*, where S consists of all labels over X, $I(X)$ is the initial state predicate, $R(X, X')$ is the transition predicate and $F(X)$ is the final state predicate. Given a variable set W $(X \subseteq W)$, a W-trace $\rho = a_0, \ldots, a_n$ is said to be a trace of M if $I(a_0)$ and $F(a_n)$ hold and for all $j < n$, $R(a_j, a'_{j+1})$ holds. The trace language $\mathbb{L}(M)$ of M is the set of all traces of M.[1]

CFA as an STS. Given a CFA $C = \langle X_C, Q_C, q0_C, \delta_C, F_C \rangle$, there exists an STS $M = \langle X, S, I, R, F \rangle$ such that $\mathbb{L}(C) = \mathbb{L}(M)$. We construct M as follows: (i) $X = X_C \cup \{q\}$ where q is a fresh variable which ranges over Q_C, (ii) $I(X) = (q = q0)$, (iii) $F(X) = \exists q_i \in F_C.(q = q_i)$, and (iv) $R(X, X') =$

$$(\exists q_1, q_2 \in Q_C, \phi \in \Phi.\ (q = q_1 \wedge q' = q_2 \wedge \delta_C(q_1, \phi, q_2) \wedge \phi(X_C, X'_C))$$

Synchronous Composition of STSs. Suppose we are given two STSs $M_1 = \langle X_1, S_1, I_1, R_1, F_1 \rangle$ and $M_2 = \langle X_2, S_2, I_2, R_2, F_2 \rangle$. We define the composition $M_1 \parallel M_2$ to be a STS $M = \langle X, S, I, R, F \rangle$ where: (i) $X = X_1 \cup X_2$, (ii) S consists of all labels over X, (iii) $I = I_1 \wedge I_2$, (iv) $R = R_1 \wedge R_2$, and (v) $F = F_1 \wedge F_2$.

Lemma 1. *Given two STSs M_1 and M_2, $\mathbb{L}(M_1 \parallel M_2) = \mathbb{L}(M_1) \cap \mathbb{L}(M_2)$.*

We use STSs to represent system components and CFA on shared variables to represent automata computed in the various AGR sub-tasks. We assume that all STSs have total transition predicates. We define the composition of an STS M with a CFA C, denoted by $M \parallel C$, to be $M \parallel M_C$, where M_C is the STS obtained from C. Although we use a synchronous notion of composition in this paper, our work can be directly extended to asynchronous composition also.

Definition 1 (Model Checking STSs). *Given an STS M and a property CFA P, the model checking question is to determine if $M \vDash P$ where $\vDash$ denotes*

[1] We overload the symbol $\mathbb{L}()$ to describe the trace language of both CFAs and STSs.

a conformance relation. Using the trace semantics for STSs and CFAs and set containment as the conformance relation, the problem can be reduced to checking if $\mathbb{L}(M) \subseteq \mathbb{L}(P)$.

Since CFAs are closed under negation and there is a language-equivalent STS for each CFA, we can further reduce the model checking question to checking if $\mathbb{L}(M \parallel M_{\overline{P}})$ is empty, where the STS $M_{\overline{P}}$ is obtained by complementing P to form $\overline{P}$ and then converting it into an STS. Let STS $\mathcal{M} = M \parallel M_{\overline{P}}$. In other words, we are interested in checking if there is an *accepting* trace in $\mathcal{M}$, i.e., a trace that ends in a state that satisfies $F_{\mathcal{M}}$.

2.1 SAT-based Model Checking

It is possible to check for existence of an accepting trace in an STS $\mathcal{M}$ using satisfiability checking. A particular instance of this problem is bounded model checking [9] where we check for existence of an accepting trace of length k using a SAT solver.

Bounded Model Checking(BMC). Given an integer bound k, the BMC problem can be formulated in terms of checking satisfiability of the following formula [9]:

$$BMC(\mathcal{M}, k) \ := \ I_{\mathcal{M}}(s_0) \wedge \bigwedge_{0 \le j \le k-1} R_{\mathcal{M}}(s_j, s_{j+1}) \wedge \bigvee_{0 \le j \le k} F_{\mathcal{M}}(s_j) \qquad (1)$$

Here s_j $(0 \le j \le k\ )$ represents the set of variables $X_{\mathcal{M}}$ at depth j. The transition relation of $\mathcal{M}$ is unfolded up to k steps, conjuncted with the initial and the final state predicates at the first and the last steps respectively, and finally encoded as a propositional formula that can be solved by a SAT solver. If the formula is SAT then the satisfying assignment corresponds to an accepting trace of length k (a counterexample to $M \vDash P$). Otherwise, no accepting trace exists of length k or less. It is possible to check for accepting traces of longer lengths by increasing k and checking iteratively.

Unbounded Model Checking(UMC). The unbounded model checking problem involves checking for an accepting trace of any length. Several SAT-based approaches have been proposed to solve this problem [25]. In this paper, we consider two approaches, one based on k-induction [27, 14, 15] and the other based on interpolation [20].

The k-induction technique [27] tries to show that there are no accepting traces of any length with the help of two SAT checks corresponding to the base and induction cases of the UMC problem. In the base case, it shows that no accepting trace of length k or less exists. This exactly corresponds to the BMC formula (Eq. 1) being UNSAT. In the induction step, it shows that if no accepting trace of length k or less exists, then there cannot be an accepting trace of length $k+1$ in $\mathcal{M}$, and is represented by the following formula:

$$Step(\mathcal{M}, k) := \bigwedge_{0 \le j \le k} R_{\mathcal{M}}(s_j, s_{j+1}) \wedge \bigwedge_{0 \le j \le k} \neg F_{\mathcal{M}}(s_j) \wedge F_{\mathcal{M}}(s_{k+1}) \wedge \bigwedge_{0 \le i \le j \le k} s_i \neq s_{j+1} \qquad (2)$$

The induction step succeeds if $\mathit{Step}(\mathcal{M}, k)$ is UNSAT. Otherwise, the depth k is increased iteratively until it succeeds or the base step is SAT (a counterexample is found). The set of constraints of form $s_i \neq s_{j+1}$ in (Eq. 2) (also known as simple path or uniqueness constraints) are necessary for completeness of the method and impose the condition that all states in the accepting trace must be unique. The method can be implemented efficiently using an incremental SAT solver [14], which allows reuse of recorded conflict clauses in the SAT solver across iterations of increasing depths. The k-induction technique has the drawback that it may require as many iterations as the length of the longest simple path between any two states in $\mathcal{M}$ (also known as recurrence diameter [9]), which may be exponentially larger than the longest of all the shortest paths (or the diameter) between any two states. Translating the above formulas to propositional logic may involve loss of structural information; we avoid it by using a SMT solver [29, 3, 28] as our main decision procedure.

Another approach to SAT-based UMC is based on using interpolants [20]. The method computes an over-approximation $\mathcal{I}$ of the reachable set of states in $\mathcal{M}$, which is also an inductive invariant for $\mathcal{M}$, by using the UNSAT proof of the BMC instance (Eq. 1). If $\mathcal{I}$ does not overlap with the set of final states, then it follows that there exists no accepting trace in $\mathcal{M}$. An important feature of this approach is that it does not require unfolding the transition relation beyond the diameter of the state space of $\mathcal{M}$, and, in practice, often succeeds with shorter unfoldings. We do not present the details of this approach here; they can be found in [20, 5].

3 Assume-Guarantee Reasoning using Learning

Assume-Guarantee reasoning allows dividing the verification task of a system with multiple components into subtasks each involving a small number of components. AGR rules may be syntactically circular or non-circular in form. In this paper, we will be concerned mainly with the following non-circular AGR rule:

Definition 2. Non-circular AGR (NC) *Given STSs M_1, M_2 and CFA P, show that $M_1 \parallel M_2 \vDash P$, by picking an assumption CFA A, such that both* **(n1)** *$M_1 \parallel A \vDash P$ and* **(n2)** *$M_2 \vDash A$ hold.*

The **NC** rule is sound and complete [23, 4, 28] and can be extended to a system of n STSs $M_1 \ldots M_n$ by picking a set of assumptions $\langle A_1 \ldots A_{n-1} \rangle$ [12]. The proof of completeness of **NC** relies on the notion of weakest assumptions [2].

Lemma 2. (Weakest Assumptions) *Given a finite STS M with support set X_M and a CFA P with support set X_P, there exists a unique weakest assumption CFA, WA, such that (i) $M \parallel WA \vDash P$ holds, and (ii) for all CFA A where $M \parallel A \vDash P$, $\mathbb{L}(A) \subseteq \mathbb{L}(WA)$ holds. Moreover, $\mathbb{L}(WA)$ is regular and the support variable set of WA is $X_M \cup X_P$.*

[2] Although we focus our presentation on **NC** rule, our results can be applied to a circular rule **C** presented in literature [7, 22] in a straightforward way. We implement and experiment with both the rules (cf. Section 5).

As mentioned earlier (cf. Section 1), a learning algorithm for regular languages, L^*, assisted by a model checker based teacher, can be used to automatically generate the assumptions [12, 7]. However, there are problems in scaling this approach to large shared memory systems. Firstly, the teacher must be able to discharge the queries efficiently even if it involves exploring a large state space. Secondly, the alphabet Σ of an assumption A is exponential in its support set of variables. Since L^* explicitly enumerates Σ during learning, we need a technique to curb this alphabet explosion. We address these problems by proposing a SAT-based implementation of the teacher and a lazy algorithm based on alphabet clustering and iterative partitioning (Section 4).

3.1 SAT-based Assume-Guarantee Reasoning

We now show how the teacher can be implemented using SAT-based model checking. The teacher needs to answer membership and candidate queries.

Membership Query. Given a trace t, we need to check if $t \in \mathbb{L}(WA)$ which corresponds to checking if $M_1 \parallel \{t\} \vDash P$ holds. To this end, we first convert t into a language-equivalent STS M_t, obtain $M = M_1 \parallel M_t$ and perform a single BMC check $BMC(M, k)$ (cf. Section 2.1) where k is the length of trace t. Note that since M_t accepts only at the depth k, we can remove the final state constraints at all depths except k. The teacher replies with a TRUE answer if the above formula instance is UNSAT; otherwise a FALSE answer is returned.

Candidate Query. Given a deterministic CFA A, the candidate query involves checking the two premises of **NC**, i.e., whether both $M_1 \parallel A \vDash P$ and $M_2 \vDash A$ hold. The latter check maps to SAT-based UMC (cf. Section 2.1) in a straightforward way. Note that since A is deterministic, complementation does not involve a blowup. For the previous check, we first obtain an STS $M = M_1 \parallel M_A$ where the STS M_A is language-equivalent to A (cf. Section 2) and then use SAT-based UMC for checking $M \vDash P$.

In our implementation, we employ both induction and interpolation for SAT-based UMC. Although the interpolation approach requires a small number of iterations, computing interpolants, in many cases, takes more time in our implementation. The induction-based approach, in contrast, is faster if it converges within small number of iterations. Using the above SAT-based query implementations, automated AGR is carried out in the standard way [12, 22, 28]. Note that the support variable set for the assumption A is initialized to that of the weakest assumption WA, i.e., $X_{M_1} \cup X_P$. Also, in practice, the AGR procedure terminates with an assumption A with significantly fewer states than WA.

4 Lazy Learning

This section presents our new lazy learning approach to address the alphabet explosion problem (cf. Section 1); in contrast to the eager BDD-based learning algorithm [4], the lazy approach (i) avoids use of quantifier elimination to compute the set of edges and (ii) introduces new states and transitions lazily only when necessitated by a counterexample. We first propose a generalization of the L^* [26] algorithm and then present the lazy l^* algorithm based on it. Due to lack of space, we omit the full details of our generalization here. The details can be found in the technical report [28].

Notation. We represent the empty trace by ϵ. For a trace $u \in \Sigma^*$ and symbol $a \in \Sigma$, we say that $u \cdot a$ is an extension of u. The membership function $[\![\cdot]\!]$ is defined as follows: if $u \in \mathbb{L}_U$, $[\![u]\!] = 1$, otherwise $[\![u]\!] = 0$. We define a *follow* function $follow : \Sigma^* \to 2^{\Sigma}$, where $follow(u)$ consists of the set of alphabet symbols $a \in \Sigma$ that u is extended by, in order to form $u \cdot a$. A counterexample trace ce is positive if $[\![ce]\!] = 1$, otherwise, it is said to be negative.

Generalized L^*. Given an unknown language $\mathbb{L}_U$ defined over alphabet Σ, L^* maintains an observation table $\mathcal{T} = (U, UA, V, T)$ consisting of trace *samples* from $\mathbb{L}_U$, where $U \subseteq \Sigma^*$ is a prefix-closed set, $V \subseteq \Sigma^*$ is a set of suffixes, UA contains extensions of elements in U and T is a map so that $T(u, v) = [\![u \cdot v]\!]$ for some $u \in U \cup UA$ and $v \in V$. In contrast to L^*, which extends each $u \in U$ by the full alphabet Σ to obtain UA, the generalized algorithm only allows each u to be extended by the elements in the corresponding *follow set*, $follow(u)$. The follow sets may vary for different $u \in U$. We assume that a procedure `Close_Table` makes $\mathcal{T}$ closed by introducing new elements u into U and adding extensions of u on elements in $follow(u)$ to UA. Note that with $follow(u) = \Sigma$, the generalized algorithm is able to compute a deterministic and complete hypothesis CFA C from a closed table $\mathcal{T}$. Given any $t \in \Sigma^*$, we define its representative trace $[t]^r$ to be the unique $u \in U$ corresponding to the final state q of a run on t in C. Also, the procedure `Learn_CE` analyzes a counterexample ce obtained from the teacher, obtains a split $ce = u_i \cdot v_i$ with distinguishing suffix v_i using the classification function $\alpha_i = [\![[u_i]^r \cdot v_i]\!]$, and adds v_i to V [26, 28]. An illustration of the algorithm can be found in the extended version.

Lazy l^* Algorithm. The main bottleneck in generalized L^* algorithm is due to alphabet explosion, i.e., it enumerates and asks membership queries on all extensions of an element $u \in U$ on the (exponential-sized) Σ explicitly. The lazy approach avoids this as follows. Initially, the follow set for each u contains a singleton element, the alphabet cluster TRUE, which requires only a single enumeration step. This cluster may then be partitioned into smaller clusters in the later learning iterations, if necessitated by a counterexample. In essence, the lazy algorithm not only determines the states of the unknown CFA, but also computes the set of distinct alphabet clusters outgoing from each state lazily.

More formally, l^* performs queries on trace sets, wherein each transition corresponds to an alphabet cluster. We therefore augment our learning setup to handle sets of traces. Let $\hat{\Sigma}$ denote the set 2^{Σ} and concatenation operator $\cdot$ be extended to sets of traces S_1 and S_2 by concatenating each pair of elements from S_1 and S_2 respectively. The follow function is redefined as $follow : \hat{\Sigma}^* \to 2^{\hat{\Sigma}}$ whose range now consists of alphabet cluster elements (or alphabet predicates). The observation table $\mathcal{T}$ is a tuple (U, UA, V, T) where $U \subseteq \hat{\Sigma}^*$ is prefix-closed, $V \subseteq \hat{\Sigma}^*$ and UA contains all extensions of elements in U on elements in their follow sets. $T(u, v)$ is defined on a sets of traces u and v, so that $T(u, v) = [\![u \cdot v]\!]$

where the membership function $[\![\cdot]\!]$ is extended to a set of traces as follows: given a trace set S, $[\![S]\!] = 1$ iff $\forall t \in S.\ [\![t]\!] = 1$. In other words, a $[\![S]\!] = 1$ iff $S \subseteq \mathbb{L}_U$. This definition is advantageous in two ways. Firstly, the SAT-based teacher (cf. Section 3.1) can answer membership queries in the same way as before by converting a single trace set into the corresponding SAT formula instance. Secondly, in contrast to a more discriminating 3-valued interpretation of $[\![S]\!]$ in terms of 0, 1 and *undefined* values, this definition enables l^* to be more lazy with respect to state partitioning.

Figure 1 shows the pseudocode for the procedure `Learn_CE`, which learns from a counterexample ce and improves the current hypothesis CFA C. Note that for each u, $follow(u)$ is set to TRUE initially. The procedure `Learn_CE` calls the `Learn_CE_0` and `Learn_CE_1` procedures to handle negative and positive counterexamples respectively. `Learn_CE_0` is the same as `Learn_CE` in generalized L^*: it finds a split of ce at position i (say, $ce = u_i \cdot v_i = u_i \cdot o_i \cdot v_{i+1}$), so that $\alpha_i \neq \alpha_{i+1}$ and adds a new distinguishing suffix v_{i+1} (which must exist by Lemma 3 below) to V to partition the state corresponding to $[u_i \cdot o_i]$. The procedure `Learn_CE_1`, in contrast, may either partition a state or partition an alphabet cluster. The case when v_{i+1} is not in V is handled as above and leads to a state partition. Otherwise, if v_{i+1} is already in V, `Learn_CE_1` first identifies states in the current hypothesis CFA C corresponding to $[u_i]$ and $[u_i \cdot o_i]$, say, q and q' respectively, and the transition predicate ϕ corresponding to the transtion on symbol o_i from q to q'. Let $u_r = [u_i]^r$. Note that ϕ is also an alphabet cluster in $follow(u_r)$ and if $o_i = (a_i, b'_i)$, then $\phi(a_i, b'_i)$ holds (cf. Section 2).

The procedure `Partition_Table` partition ϕ using o_i (into ϕ_1 and ϕ_2) and updates the follow set of u_r. Also, it modifies the sets U and UA so that U remains prefix-closed and UA only contains extensions of U on the new follow set [28]. Note that since all the follow sets are disjoint and complete at each iteration, the hypothesis CFA obtained from a closed table $\mathcal{T}$ is always deterministic and complete (cf. Section 2).

```
Init: $\forall u \in \Sigma^*$, set $follow(u)$ = TRUE
Learn_CE($ce$)
 if ( $[\![ce]\!] = 0$ )
  Learn_CE_0(ce)
 else Learn_CE_1(ce)

 Learn_CE_1($ce$)
  Find $i$ so that $\alpha_i = 1$ and $\alpha_{i+1} = 0$
  if $v_{i+1} \notin V$
   $V := V \cup \{v_{i+1}\}$
   For all $u \in U \cup UA$: Fill(u, $v_{i+1}$)
  else
   Let $ce = u_i \cdot o_i \cdot v_{i+1}$
   Let $q = [u_i]$ and $q' = [u_i \cdot o_i]$
   Suppose $R_C(q, \phi, q')$ and $o_i \in \phi$
   Partition_Table($[u_i]^r$, $\phi$, $o_i$)

Learn_CE_0($ce$)
 Find $i$ so that $\alpha_i = 0$ and $\alpha_{i+1} = 1$
 $V := V \cup \{v_{i+1}\}$
 For all $u \in U \cup UA$: Fill(u, $v_{i+1}$)

 Partition_Table($u_r, \phi, a$)
  $\phi_1 := \phi \wedge a$, $\phi_2 := \phi \wedge \neg a$
  $follow(u_r) := follow(u_r) \cup \{\phi_1, \phi_2\} \setminus \{\phi\}$
  Let $Uext = \{u \in U \mid \exists v \in \hat{\Sigma}^*.\ u = u_r \cdot \phi \cdot v\}$
  Let $UAext = \{u \cdot \phi_f \mid u \in Uext \wedge \phi_f \in follow(u)\}$
  $U := U \setminus Uext$
  $UA := UA \setminus UAext$
  For $u \in \{u_r \cdot \phi_1, u_r \cdot \phi_2\}$
   $UA := UA \cup \{u\}$
   For all $v \in V$: Fill($u, v$)
```

Fig. 1. Pseudocode for the lazy l^* algorithm (mainly the procedure `Learn_CE`).

Example. Figure 2 illustrates the l^* algorithm for the unknown language $\mathbb{L}_U = (a|b|c|d) \cdot (a|b)^*$. Recall that the labels a, b, c and d are, in fact, predicates over program variables. The upper and lower parts of the table represent U and UA respectively, while the columns contain elements from V. The Boolean table entries correspond to the membership query $[\![u \cdot v]\!]$ where u and v are the row and column entries respectively. The algorithm initializes both U and V with element ϵ and fills the corresponding table entry by asking a membership query. Then, it asks query for a single extension of ϵ on cluster T (the L^* algorithm will instead asks queries on each alphabet element explicitly). Since $\epsilon \not\equiv T$, in order to make the table closed, the algorithm further needs to query on the trace $T \cdot T$. Now, it constructs the first hypothesis (Figure 2(i)) and asks a candidate query with it. The teacher replies with a counterexample $a \cdot a$, which is then used to partition the follow set of T into elements a and $\bar{a}$. The table is updated and the algorithm continues iteratively. The algorithm converges to the final CFA using four candidate queries; the figure shows the hypotheses CFAs for first, third and last queries. The first three queries are unsuccessful and return counterexamples $a \cdot a$ (positive), $a \cdot b$ (positive), $a \cdot d \cdot c$ (negative). The first two counterexamples lead to cluster partitioning (by a and b respectively) and the third one leads to state partitioning. Note that the algorithm avoids explicitly enumerating the alphabet set for computing extensions of elements in Σ. Also, note that the algorithm is insensitive to the size of alphabet set to some extent: if $\mathbb{L}_U$ is of the form $\Sigma \cdot (a|b)^*$, the algorithm always converges in the same number of iterations since only two cluster partitions from state q_1 need to be made.

The drawback of this lazy approach is that it may require more candidate queries as compared to the generalized L^* in order to converge. This is because the algorithm is lazy in obtaining information on the extensions of elements in U and therefore builds candidates using less information, e.g., it needs two candidate queries to be able to partition the cluster T on both a and b (note that the corresponding counterexamples $a \cdot a$ and $a \cdot b$ differ only in the last transition). We have developed a SAT-based method [28] that accelerates learning in such cases by generalizing a counterexample *ce* to include a set of similar counterexamples (ce') and then using ce' to perform a *coarser* cluster partition.

Lemma 3. *The procedure* `Learn_CE_0` *must lead to addition of at least one new state in the next hypothesis CFA.*

Lemma 4. *The procedure* `Learn_CE_1` *either leads to addition of at least one new state or one transition in the next hypothesis CFA.*

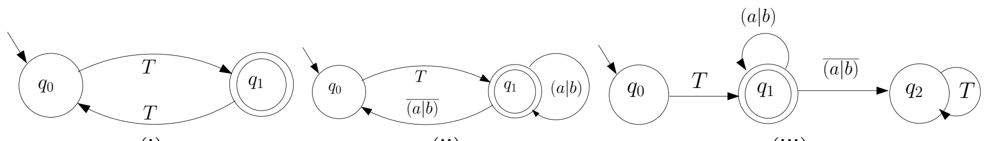
Theorem 1. *l^* terminates in $O(k \cdot 2^n)$ iterations where k is the alphabet size and n is the number of states in the minimum deterministic CFA C_m corresponding to $\mathbb{L}_U$.*

Optimizing l^*. Although the theoretical complexity of l^* is high (mainly due to the reason that l^* may introduce a state corresponding to each subset of states reachable at a given depth in C_m), our experimental results show that the algorithm is effective in computing small size assumptions on real-life examples. Moreover, in the context of AGR, we seldom need to learn C_m completely; often, an approximation obtained at an intermediate learning step is sufficient.

	ϵ
ϵ	0 (q_0)
T	1 (q_1)
T· T	0

	ϵ
ϵ	0 (q_0)
T	1 (q_1)
T·a	1
T·b	1
T·$\overline{(a\mid b)}$	0

	ϵ	c
ϵ	0	1 (q_0)
T	1	0 (q_1)
T·$\overline{(a\mid b)}$	0	0 (q_2)
T·a	1	0
T·b	1	0
T·$\overline{(a\mid b)}$· T	0	0



(i) (ii) (iii)

Fig. 2. Illustration of the l^* algorithm for $\mathbb{L}_U = (a|b|c|d)(a|b)^*$. Rows and column represent elements of $U \cup UA$ and V respectively. Alphabets are represented symbolically: $\mathrm{T} = (a|b|c|d)$, $\overline{(a|b)} = (c|d)$.

5 Implementation and Experiments

We have implemented our SAT-based AGR approach based on **NC** and **C** rules in a tool called SYMODA, written in C++. The l^* algorithm is implemented together with related optimizations. SMV and Verilog benchmarks are translated into an intermediate input language of the tool using automated scripts [28]. We use the incremental SMT solver YICES [3, 13] as the main decision procedure. Interpolants are obtained using the library interface to the FOCI tool [2]. We represent states of a CFA explicitly while BDDs are used to represent transitions compactly and avoid redundancy.

Experiments. All experiments were performed on a 1.4GHz AMD machine with 3GB of memory running Linux. Table 1 compares three algorithms for automated AGR: a BDD-based approach [4, 22] (BDD-AGR), our SAT-based approach using l^* (Lazy-AGR) and (P-AGR), which uses a learning algorithm for parameterized systems [8]. The last algorithm was not presented in context of AGR earlier; we have implemented it using a SAT-based teacher and other optimizations for comparison purposes. The BDD-AGR approach automatically partitions the given model before learning assumptions while we manually assign each top-level module to a different partition. Benchmarks *s1a*, *s1b*, *guidance*, *msi* and *syncarb* are derived from the NuSMV tool set and used in the previous BDD-based approach [22] while *peterson* and *CC* are obtained from the VIS and Texas97 benchmark sets [1]. All examples except *guidance* and *CC* can be proved using monolithic SAT-based UMC in small amount of time. Note that in some of these benchmarks, the size of the assumption alphabet is too large to be even enumerated in a short amount of time.

The SAT-based Lazy-AGR approach performs better than the BDD-based approach on *s*1*a* and *s*2*a* (cf. Table 1); although they are difficult for BDD-based model checking [4], SAT-based UMC quickly verifies them. On the *msi* example, the Lazy-AGR approach scales more uniformly compared to BDD-AGR. BDD-AGR is able to compute an assumption with 67 states on the *syncarb* benchmark while our SAT-based approaches with interpolation timeout with assumption sizes of around 30. The bottleneck is SAT-based UMC in the candidate query checks; the k-induction approach keeps unfolding transition relations to increasing depths while the interpolants are either large or take too much time to compute. On the *peterson* benchmark, BDD-AGR finishes earlier but with larger assumptions of size up to 34 (for two partitions) and 13 (for four partitions). In contrast, Lazy-AGR computes assumptions of size up to 6 while P-AGR computes assumptions of size up to 8. This shows that it is possible to generate much smaller assumptions using the lazy approach as compared to the eager BDD-based approach. Both the *guidance* and *syncarb* examples require interpolation-based UMC and timeout inside a candidate query with the k-induction based approach. P-AGR timeouts in many cases where Lazy-AGR finishes since the former performs state partitions more eagerly and introduces unnecessary states in the assumptions. We also compare the impact of various optimizations in the extended version [28].

Example	TV	GV	Mono	BDD-AGR				P-AGR				Lazy-AGR			
				NC		**C**		**NC**		**C**		**NC**		**C**	
				#A	Time	#A	Time	#A	Time	#A	Time	#A	Time	#A	Time
s1a	86	5	0.54	2	754	2	223	3	3	3	3	3	3.5	3	1.3
s1b	94	5	0.58	2	TO	2	1527	3	3.3	3	3.3	3	3.9	3	2
guidance	122	22	129	2	196	2	6.6	1	31.5^i	5	146^i	1	40^i	3	55^i
msi(3)	57	22	1.2	2	2.1	2	0.3	1	8	*	TO	1	8	3	17
msi(5)	70	25	2.2	2	1183	2	32	1	16	*	TO	1	15	3	43
syncarb	21	15	3.16	-	-	67	30	*	TO^i	*	TO^i	*	TO^i	*	TO^i
peterson	13	7	0.54	-	-	34	2	6	53^i	8	210^i	6	13	6	88^i
CC(2a)	78	30	3.9	-	-	-	-	1	8	*	TO	1	8	4	26
CC(3a)	115	44	3.7	-	-	-	-	1	8	*	TO	1	7	4	20
CC(2b)^i	78	30	337	-	-	-	-	*	TO	*	TO	10	1878	5	87
CC(3b)^i	115	44	526	-	-	-	-	*	TO	*	TO	6	2037	11	2143

Table 1. Comparison of BDD-based and Lazy AGR schemes. P-AGR uses a learning algorithm for parameterized systems [8] while Lazy-AGR uses l^*. TV and GV represent the number of total and global boolean variables respectively. The Mono column shows the time taken with SAT-based UMC. All times are in seconds. TO denotes a timeout of 3600 seconds.#A denotes states of the largest assumption. '-' denotes that data could not be obtained due to the lack of tool support (The tool does not support the **NC** rule or Verilog programs as input). The superscript ^{i} denotes that interpolant-based UMC was used.

Conclusions. We have presented a new SAT-based approach to automated AGR for shared memory systems based on lazy learning of assumptions; alphabet explosion during learning is avoided by representing alphabet clusters symbolically and performing on-demand cluster partitioning during learning. Experimental results demonstrate the effectiveness of our approach on hardware benchmarks. Since we employ off-the-shelf SMT solvers, we can directly leverage future improvements in SAT/SMT technology. Our techniques can be applied to software and other infinite state systems provided the weakest assumption has a finite bisimulation quotient. Future work includes investigating techniques to exploit incremental SAT solving for discharging each AGR premise, faster counterexample detection and obtaining good system decompositions for AGR.

Acknowledgements. We would like to thank Flavio Lerda for help wth the translator from SMV to SIL and the C interface to FOCI and also for numerous helpful discussions. We would like to thank Constantinos Bartzis and Tamir Heyman for several informative discussions. We also thank Dilsun Kaynar for carefully reading through a draft of this paper and providing useful comments.

References

[1] `http://vlsi.coloradu.edu/~vis/`.
[2] Foci: An interpolating prover. `http://www.kenmcmil.com/foci.html`.
[3] Yices: An smt solver. `http://yices.csl.sri.com/`.
[4] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *CAV*, 2005.
[5] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. McMillan. An analysis of sat-based model checking techniques in an industrial environment. In *CHARME*, pages 254–268, 2005.
[6] Dana Angluin. Learning regular sets from queries and counterexamples. In *Information and Computation*, volume 75(2), pages 87–106, November 1987.
[7] H. Barringer, D. Giannakopoulou, and C.S Pasareanu. Proof rules for automated compositional verification. In *SAVCBS, 2003*.
[8] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines with parameters. In *FASE*, pages 107–121, 2006.
[9] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Y. Zue. *Bounded Model Checking*, volume 58 of *Advances in computers*. 2003.
[10] Sagar Chaki and Ofer Strichman. Optimized L* for assume-guarantee reasoning. In *TACAS*, 2007. To Appear.
[11] J. Cobleigh, G. Avrunin, and L. Clarke. Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In *ISSTA*, pages 97–108, 2006.
[12] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, volume 2619. Springer-Verlag, 2003.
[13] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, pages 81–94, 2006.
[14] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electr. Notes Theor. Comput. Sci.*, 89(4), 2003.
[15] R. Armoni et al. Sat-based induction for temporal safety properties. *Electr. Notes Theor. Comput. Sci.*, 119(2):3–16, 2005.
[16] Mihaela Gheorghiu, Dimitra Giannakopoulou, and Corina S. Pasareanu. Refining interface alphabets for compositional verification. In *TACAS*, 2007. To Appear.
[17] JE Hopcroft and JD Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.
[18] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.
[19] Patrick Maier. A set-theoretic framework for assume-guarantee reasoning. In *ICALP*, pages 821–834, 2001.
[20] K. L. McMillan. Interpolation and sat-based model checking. In *CAV*, pages 1–13, 2003.
[21] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.
[22] Wonhong Nam and Rajeev Alur. Learning-based symbolic assume-guarantee reasoning with automatic decomposition. In *ATVA*, pages 170–185, 2006.
[23] Kedar S. Namjoshi and Richard J. Trefler. On the completeness of compositional reasoning. In *CAV2000*, number 1855, pages 139–153. Springer-Verlag, 2000.
[24] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and models of concurrent systems*. Springer-Verlag, 1985.
[25] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat-based formal verification. *STTT*, 7(2):156–173, 2005.
[26] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. In *Inf. Comp.*, volume 103(2), pages 299–347, 1993.
[27] Mary Sheeran, Satnam Singh, and Gunnar Stalmarck. Checking safety properties using induction and a sat-solver. In *FMCAD*, pages 108–125, 2000.
[28] Nishant Sinha and Edmund Clarke. SAT-based compositional verification using lazy learning. Technical report CMU-CS-07-109, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, February 2007.
[29] C. Tinelli and S. Ranise. SMT-LIB: The Satisfiability Modulo Theories Library. `http://goedel.cs.uiowa.edu/smtlib/`, 2005.