SAT-based Compositional Verification using Lazy Learning

Nishant Sinha[*], **Edmund Clarke**[†]

Feburary 2007
CMU-CS-07-109

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*]Elec. and Computer Engg. Dept., Carnegie Mellon University, Pittsburgh, PA, USA
[†]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

This research was sponsored by the National Science Foundation under grant nos. CNS-0411152, CCF-0429120, CCR-0121547, and CCR-0098072, the Semiconductor Research Corporation under grant no. 2006-TJ-1366, the US Army Research Office under grant no. DAAD19-01-1-0485, and the Office of Naval Research under grant no. N00014-01-1-0796. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Assume-Guarantee Reasoning, SAT, SMT, BMC, Learning

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) [26, 22, 29] 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 [15] has been proposed to automatically generate these assumptions using learning algorithms for regular languages assisted by a model checker. Figure 1 shows a simplified view of this approach for 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^* [5, 33] 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 [21]) 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.

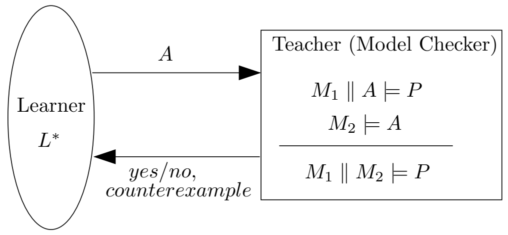


Figure 1: Learning-based Assume-Guarantee Reasoning procedure (simplified).

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, 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 [27].

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) [10] 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 [34] and interpolation [25] respectively. Moreover, we use a SMT solver as the main decision procedure [36, 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 [31]. 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 [31, 27] 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. [15] 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 [31, 27]. The problem of whether it is possible to obtain good decompositions of systems for this approach has been studied in [16]. An overview of other related work can be found in [19, 27, 13]. SAT-based bounded model checking for LTL properties was proposed by Biere et al. [10] and several improvements, including techniques for making it complete have been proposed [30, 4]. All the previous approaches are non-compositional, i.e., they build a monolithic 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 [31] for shared memory systems using automated system decomposition [27] 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.

Recently, two approaches for improved learning based on alphabet under-approximation and iterative enlargement [12, 19] have been proposed. Our lazy approach is complementary: while the above techniques try to reduce the overall alphabet by under-approximation, our technique tries to compactly represent a large alphabet set symbolically and performs localized partitioning. In cases where a small alphabet set is not sufficient, the previous techniques may not be effective. We also note that both the above approaches can be combined with our approach by removing assumption variables during learning and adding them back iteratively. A learning algorithm for parameterized systems (alphabet consists of a small set of basis symbols, each of which is parameterized by a set of boolean variables) was proposed in [9]. Our lazy learning algorithm is different: we reason about a set of traces directly using a SAT-based model checker and perform more efficient counterexample analysis by differentiating positive and negative counterexamples (cf. Section 4).

In contrast to the counterexample-guided abstraction refinement (CEGAR) approach [23, 14, 7], the assumption languages may change non-monotonically across iterations of the learning algorithm. The CEGAR approach removes spurious behaviors from an abstraction by adding new predicates. In contrast, the learning-based approach uses state- and cluster-partitioning based on the Nerode congruence [21] to both remove and add behaviors. Similar to a lazy approach to CEGAR [20], the lazy learning algorithm localizes the cluster partitioning to the follow set of a particular state and adds only a single cluster to the follow sets at each iteration.

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 [28, 24]. 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 $\bigvee\{\phi \in \mathit{fol}(q)\}$ = TRUE and disjoint iff for all $\phi_i, \phi_j \in \mathit{fol}(q)$, $\phi_i \wedge \phi_j$ = 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 $\mathbb{L}(\overline{C}) = \overline{\mathbb{L}(C)}$.

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

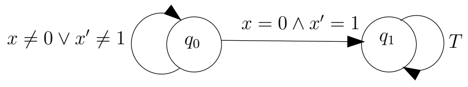


Figure 2: A CFA on support $X = \{x\}$; x is a boolean. $\Sigma = \{(x = 0, x' = 0), (x = 0, x' = 1), (x = 1, x' = 0), (x = 1, x' = 1)\}$. *fol*$(q_0) = \{(x = 0 \wedge x' = 1), (x \neq 0 \vee x' \neq 1)\}$. *fol*$(q_1) = \{\text{TRUE}\}$. Note that the first element of *fol*(q_0) corresponds to an alphabet symbol while the second element is an alphabet cluster. Also, both *fol*(q_0) and *fol*(q_1) are disjoint and complete.

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 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 $F_{\mathcal{M}}$.

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

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 [10] 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 [10]:

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

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 $M \models P$). 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 [30]. In this paper, we consider two approaches, one based on k-induction [34, 18, 6] and the other based on interpolation [25].

The k-induction technique [34] 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:

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

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 $s_i \ne s_{j+1}$ 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 [18], 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 [10]), which may be exponentially larger than the longest of all the shortest paths (or the diameter) between any two states.

Another approach to SAT-based UMC is based on using interpolants [25]. 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 [25, 4].

In order to use a SAT solver, the above formula instances have to be translated into propositional logic. A lot of structural information is lost (e.g., relation between bits of an encoded variable) due to this translation and may lead to useless computation by the SAT solver. We can avoid this translation by using an SMT solver [36, 3]. Besides allowing propositional constraints, an SMT solver also supports input formulas in one or more (ground) first order theories, e.g., the quantifier-free fragment of linear arithmetic over integers. Therefore, both BMC and UMC based on k-induction can be carried out using a SMT solver, provided it supports the theories over which the above formulas are defined. A particular mixed boolean/integer encoding of hardware RTL constructs can be found in [11]. Similarly, interpolation-based UMC may be carried out using an interpolating prover provided it can generate interpolants in the required theories.

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 following circular rule has also been proposed in literature [8, 27].

Definition 3 Circular AGR (C) *Show that $M_1 \parallel M_2 \vDash P$ holds by picking an CFA assumption tuple, $\langle A_1, A_2 \rangle$, such that each of the following hold:* **(c1)** $M_1 \parallel A_1 \vDash P$ **(c2)** $M_2 \parallel A_1 \vDash P$ *and* **(c3)** $\overline{A_1} \parallel \overline{A_2} \vDash P$.

Both **NC** and **C** rules are sound and complete [28, 8, 27]. Moreover, both can be extended to a system of n STSs $M_1 \ldots M_n$ by picking a set of assumptions (represented as a tuple) $\langle A_1 \ldots A_{n-1} \rangle$ for **NC** and $\langle A_1 \ldots A_n \rangle$ for **C** respectively [15, 8, 27]. The proofs of completeness for both these rules rely on the notion of weakest assumptions.

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 $X_M \cup X_P$.*

Proof. By definition $\mathbb{L}(M) \cap \mathbb{L}(A) \subseteq P$. On rearranging, we get, $\mathbb{L}(A) \subseteq \overline{\mathbb{L}(M)} \cup \mathbb{L}(P)$. Hence, for the weakest assumption *WA*, $\mathbb{L}(\textit{WA}) = \overline{\mathbb{L}(M)} \cup \mathbb{L}(P)$. Since, the support set of $\overline{\mathbb{L}(M)}$ and $\mathbb{L}(P)$ is X_M and X_P respectively, the support of $\mathbb{L}(\textit{WA})$ and therefore *WA* is $X_M \cup X_P$.[2]

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 [15, 8]. 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 $t \in \mathbb{L}(\textit{WA})$ 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 $\textit{BMC}(M, k)$ (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. Now, automated AGR is carried out in the standard way (details can be found in [15, 27]) based on the above queries. The learner sets the support variable set for the assumption A to the support of the weakest assumption ($X_{wa} = X_{M_1} \cup X_P$) and iteratively computes hypotheses assumptions by asking membership and candidate queries until **n1** holds. The last assumption is then presented in a candidate query which checks if **n2** holds. If **n2** holds, then the procedure terminates. Otherwise, a counterexample ce is returned. ce may be spurious; a membership query on ce is used to check if it is spurious. In that case, ce is projected to X_{wa} to obtain ce' and learning continues with the ce'. Otherwise, ce is returned as an actual counterexample to $M_1 \parallel M_2 \vDash P$. The termination of this procedure is guaranteed by the existence of a unique weakest assumption *WA*. However, it is important to note

[2]We would like to thank Kedar Namjoshi for suggesting a simple proof.

that we seldom need to compute *WA*. In practice, this procedure terminates with any assumption A that satisfies **n1** and **n2** and the size of A is much smaller than that of *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 [31], 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^* [33] algorithm and then present the lazy l^* algorithm based on it.

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 $[\![u]\!] = 0$. For each $u \in \Sigma^*$, we define a *follow* function $follow : \Sigma^* \rightarrow 2^{\Sigma}$, 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.

The basis of our generalization of L^* is the *follow* function; instead of allowing each $u \in \Sigma^*$ to be extended by the full alphabet Σ as in original L^*, we only allow u to be extended by the elements in $follow(u)$. With $follow(u)$ = Σ (for each u) the generalized algorithm reduces to the original algorithm.

Recall that L^* is an algorithm for learning the minimum DFA D corresponding to an unknown regular language $\mathbb{L}_U$ defined over alphabet Σ. The algorithm is based on the Nerode congruence [21]: For u, $u' \in \Sigma^*$, $u \equiv u'$ iff

$$\forall v \in \Sigma^*, u \cdot v \in \mathbb{L}_U \;\Leftrightarrow\; u' \cdot v \in \mathbb{L}_U$$

L^* iteratively identifies the different congruence classes in $\mathbb{L}_U$ by discovering a representative prefix trace ($u \in \Sigma^*$) for each of the classes with the help of a set of distinguishing suffixes $V \subseteq \Sigma^*$ that differentiate between these classes.

4.1 Generalized L^* Algorithm

L^* maintains an observation table $\mathcal{T}$ = (U, UA, V, T) consisting of trace *samples* from $\mathbb{L}_U$. Here $U \subseteq \Sigma^*$ is a prefix-closed set of traces and $V \subseteq \Sigma^*$ is a set of suffixes. The algorithm also maintains extensions of $u \in U$ in *UA* on the *follow* set of u, i.e., *UA* $= \{u \cdot a | u \in U, a \in follow(u)\}$. T maps each $u \in (U \cup \mathit{UA})$ to a function $T(u) : V \rightarrow \{0, 1\}$ so that $T(u)(v)$ = $[\![u \cdot v]\!]$. We write $T(u)(v)$ as $T(u, v)$ for ease of notation. We define a congruence $\equiv$ as follows: for $u, u' \in U \cup \mathit{UA}$, $u \equiv u'$ iff $\forall v \in V$, $T(u, v) = T(u', v)$. We can view $\equiv$ as the restriction of Nerode congruence to prefixes in $U \cup \mathit{UA}$ and suffixes in V.

Well-formed Table. An observation table $\mathcal{T}$ is said to be well-formed if for all u, $u' \in U$, $u \not\equiv u'$. In the generalized algorithm, $\mathcal{T}$ is always well-formed [3].

[3]A notion of *consistency* is usually used in presentation of L^* [5]. We ignore it since U never contains distinct elements u, u' so that $u \equiv u'$. Therefore table consistency is always maintained.

```
Learner $L^*$
Let $\mathcal{T} = (U, V, T)$ be an observation table
  Init:
      $U := V := \{\epsilon\}$
      $\forall u \in \Sigma^*$, set $follow(u) = \Sigma$
      Fill ($\epsilon$,$\epsilon$)
      Fill_All_Succs ($\epsilon$)
  Loop:
      Close_Table($\mathcal{T}$)
      DFA $D$ := Mk_DFA($\mathcal{T}$)
      if ( Ask_Cand_Q ($D$) = TRUE )
        return $D$;
      else
        Let the counterexample be $ce$
        Learn_CE ($ce$)

Close_Table($\mathcal{T}$)
  while $\mathcal{T}$ is not closed
    Pick $u' \in UA$ such that $\forall u \in U.\ u \not\equiv u'$
    $U := U \cup \{u'\}$, $UA := UA \setminus \{u'\}$
    Fill_All_Succs(u')

Fill_All_Succs (u)
  For all $a \in follow(u)$
    $UA := UA \cup \{u \cdot a\}$
    For each $v \in V$: Fill($u \cdot a$, $v$)

Fill (u, v)
  $T(u, v)$ := Ask_Mem_Q($u \cdot v$)
```

Figure 3: The generalized L^* algorithm

Table Closure. The observation table $\mathcal{T}$ is said to be closed if for each $u \cdot a \in UA$, there is a $u' \in U$, so that $u \cdot a \equiv u'$. In this case, we write $u' = [u \cdot a]^r$ and say that u' is the *representative trace* for $u \cdot a$. Note that u' is unique since $\mathcal{T}$ is well-formed. We extend $[\cdot]^r$ to U by defining $[u]^r = u$ for each $u \in U$. This is possible since $\mathcal{T}$ is well-formed. For all $u \in (U \cup UA)$, we denote the set of traces equivalent to u by $[u]$, where

$$[u] = \{u' \in (U \cup UA) \mid u \equiv u'\}$$

Given any observation table $\mathcal{T}$, we assume that a procedure Close_Table makes it closed.

DFA Construction. Given a closed table $\mathcal{T}$, L^* obtains a DFA $D = \langle Q, q_0, \delta, F\rangle$ from it as follows: $Q = \{[u] \mid u \in U\}$, where a state $q \in Q$ corresponds to the equivalence class $[u]$ of a trace $u \in U$, $q_0 = [\epsilon]$, $\delta([u], a) = [u \cdot a]$ for each $u \in U$ and $a \in follow(u)$. $F = \{[u] \mid u \in U \wedge T(u, \epsilon) = 1\}$. Suppose that a procedure called Mk_DFA implements this construction. Note that D is deterministic and if $follow(u) = \Sigma$, then D is complete.

Figure 3 shows the pseudocode of the generalized L^* algorithm. The $Init$ block performs the initialization steps while the $Loop$ block performs the learning task iteratively. We assume that the teacher provides procedures to perform the membership query (Ask_Mem_Q) and the candidate query (Ask_Cand_Q). In the $Init$ block, L^* initializes the sets U and V with the empty trace ϵ. It then updates the map T by first asking a membership query for table element (ϵ, ϵ) (using the Fill procedure) and then for all elements in its follow set $follow(\epsilon)$ (using the Fill_All_Succs procedure). The $Loop$ block executes the following tasks iteratively: it first makes $\mathcal{T}$ closed (using the Close_Table procedure), computes a candidate DFA D (using the Mk_DFA procedure) and then performs a candidate query (Ask_Cand_Q) with D. If the candidate query succeeds, L^* finishes by returning D; otherwise, the $Loop$ block continues iteratively by learning from the counterexample obtained (using the procedure Learn_CE).

Learn_CE **procedure**. In order to describe Learn_CE, we first extend $[\cdot]^r$ (defined under **Table Closure** above) to any $w \in \Sigma^*$ as follows. Given $w \in \Sigma^*$, we define $[w]^r = u$ ($u \in U$), such that if $q = \delta_D^*(q_0, w)$, then $q = [u]^r$. It follows from the construction of DFA D that such a u must exist[4]. Intuitively, $[w]^r$ is the representative element of the unique state q (equivalence class) that w reaches when it is run on D starting at q_0. We define an i-*split* of the counterexample ce to be a tuple (u_i, v_i), where $ce = u_i \cdot v_i$ and $|u_i| = i$. In words, an i-split of ce ($0 \leq i \leq |ce|$), consists of its prefix u_i of length i and the corresponding suffix v_i. Further, for an i-split, we define $\alpha_i = [\![[u_i]^r \cdot v_i]\!]$ ($\alpha_i \in \{0, 1\}$) [33]. It can be shown that $\alpha_0 = [\![[\epsilon]^r \cdot ce]\!] = [\![ce]\!]$ and $\alpha_{|ce|} = [\![[ce]^r \cdot \epsilon]\!] \neq [\![ce]\!]$ [33]. Intuitively, α_i checks whether the prefix u_i is classified into the correct equivalence class $[u_i]$ [5]; if $\alpha_i = [\![ce]\!]$, it implies that u_i is classified correctly, otherwise it is mis-classified and L^* must *re-classify* u_i to a different equivalence class. But, the counterexample ce is mis-classified by definition and hence $\alpha_{|ce|} \neq [\![ce]\!]$. The `Learn_CE` procedure is given by the following pseudocode:

```
Learn_CE (ce)
  Find $i$ by binary search such that $\alpha_i \neq \alpha_{i+1}$
  $V := V \cup \{v_{i+1}\}$
  For all $u \in U \cup UA$. Fill(u, v_{i+1})
```

Since $\alpha_0 \neq \alpha_{|ce|}$, there must exist some i, $0 \leq i \leq |ce|$, so that $\alpha_i \neq \alpha_{i+1}$. In that case, `Learn_CE` will add v_{i+1} to the set of suffixes V and update the table. Intuitively, u_{i+1} is wrongly classified into the equivalence class $[u_{i+1}]$, (which corresponds to a state in D, say q), and v_{i+1} is a witness for this mis-classification. Adding v_{i+1} to V distinguishes the correct equivalence class (say q') from q and redirects u_{i+1} to the correct equivalence class q'. We call this a *state partition* (of q). Rivest and Schapire [33] show that `Learn_CE` must add at least one state to the new candidate DFA constructed in the next iteration. Since the number of states in any candidate DFA is bounded by the number of states in the minimum DFA for $\mathbb{L}_U$, generalized L^* must eventually terminate with the minimum DFA.

Learning CFAs with Generalized L*. L^* can be directly extended to learn a CFA corresponding to $\mathbb{L}_U$ over a support variable set X by setting Σ to all total labels on $X \cup X'$ (cf. Section 2). However, Σ is exponential in the size of X and will pose a bottleneck for L^*. More precisely, the loop in the procedure `Fill_All_Succs` will execute an exponential number of times, leading to inefficiency.

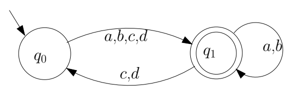
Figure 4 illustrates the generalized L^* algorithm computation for the language $\mathbb{L}_U = (a|b|c|d)(a|b)^*$ with $\Sigma = \{a, b, c, d\}$. Note that the symbols, e.g., a,b, etc., actually represent predicates over program variables, e.g., $a \equiv (x = 0 \wedge x' = 1)$. The algorithm begins with $U = V = \{\epsilon\}$ (top part of the table) and *fills* the table entry corresponding to row and column elements ϵ to 0 by asking a membership query. Then, it asks four membership queries for extensions of ϵ on each symbol in alphabet followed by column element ϵ explicitly ($a \cdot \epsilon$, $b \cdot \epsilon$, etc.) and stores the result in the table. Note that $T(a) = (1)$ but $T(\epsilon) = (0)$. Therefore, `Close_Table` adds a to U and then algorithm again asks explicit membership queries to fill the table on extensions of a, i.e., $a \cdot a$, $a \cdot b$, etc. Once $\mathcal{T}$ is closed, L^* constructs a hypothesis DFA (see **DFA Construction** above), shown in the figure on left, and makes a candidate query with it. The teacher provides a counterexample $ce = a \cdot d \cdot c$. `Learn_CE` analyzes ce and adds a distinguishing suffix c to V. Again,

[4]For traces $w \in U \cup UA$, this extension coincides with the earlier definition.

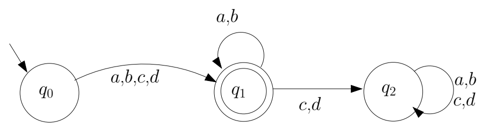
[5]For example, if ce is rejected in $\mathbb{L}_U$, i.e.,. $[\![ce]\!] = 0$, then each of its prefixes u_i must fall into an equivalence class $[u_i]$ such that $[u_i]$ rejects the corresponding suffix v_i, i.e., $[\![[u_i]^r \cdot v_i]\!] = \alpha_i = 0$.

	ϵ	
ϵ	0	(q_0)
a	1	(q_1)
$(b\|c\|d)$	1	
$a \cdot (a\|b)$	1	
$a \cdot (c\|d)$	0	

	ϵ	c	
ϵ	0	1	(q_0)
a	1	0	(q_1)
$a \cdot c$	0	0	(q_2)
$(b\|c\|d)$	1	0	
$a \cdot (a\|b)$	1	0	
$a \cdot d$	0	0	
$a \cdot c \cdot (a\|b\|c\|d)$	0	0	



(i)



(ii)

Figure 4: Illustration of the generalized L^* algorithm for $\mathbb{L}_U = (a|b|c|d)(a|b)^*$. Rows and columns represent elements of $U \cup UA$ and V respectively. For row element u and column element v, the table entries correspond to $[\![u \cdot v]\!]$. Elements in U are labeled with corresponding states q_i. Iterations (i) and (ii) correspond to the first and second candidates respectively and their observation tables. The teacher provides a counterexample $a \cdot d \cdot c$ for the first query; `Learn_CE` adds c to V. Similar rows are clustered together by overloading the $|$ symbol for compact illustration.

L^* obtains a closed table and asks a candidate query on the next hypothesis. Since this hypothesis is correct, the algorithm terminates.

4.2 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 $follow : \hat{\Sigma}^* \rightarrow 2^{\hat{\Sigma}}$ 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 in-

```
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($u, v$)
```

Figure 5: Pseudocode for the lazy l^* algorithm (mainly the procedure Learn_CE).

terpretation of $[\![S]\!]$ in terms of 0, 1 and *undefined* values, this definition enables l^* to be more lazy with respect to state partitioning.

Figure 5 shows the pseudocode for the procedure Learn_CE, which learns from a counterexample ce and improves the current hypothesis CFA C. 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 is then used to partition ϕ using o_i (into $\phi_1 = \phi \wedge o_i$ and $\phi_2 = \phi \wedge \neg o_i$) and update the follow set of u_r by removing ϕ and adding ϕ_1 and ϕ_2. Note that U and *UA* may also contain extensions of $u_r \cdot \phi$, given by $Uext$ and *UAext* respectively. In order to keep U prefix-closed and have only extensions of U in *UA*, the procedure removes $Uext$ and *UAext* from U and *UA* respectively. Finally, it adds the extensions of u_r on the new follow set elements ϕ_1 and ϕ_2 to *UA* and performs the corresponding membership queries. 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).

Example. Figure 6 illustrates the l^* algorithm for the unknown language $\mathbb{L}_U = (a|b|c|d) \cdot (a|b)^*$.

	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·$\overline{(a\|b)}$	0	

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

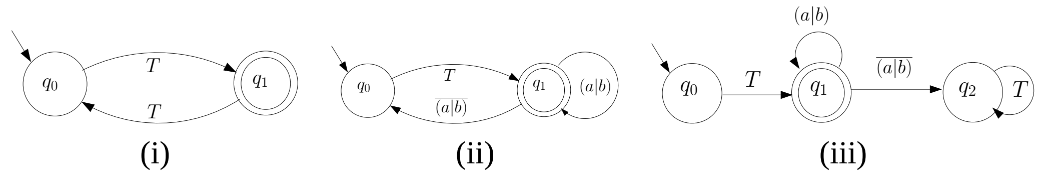


Figure 6: Illustration of the l^* algorithm for $\mathbb{L}_U = (a|b|c|d)(a|b)^*$. Rows and column represent elements of $U \cup UA$ and V respectively. Alphabets are represented symbolically: T = $(a|b|c|d)$, $\overline{(a|b)} = (c|d)$.

Recall that the labels a, b, c and d are, in fact, valuations of program variables. 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). Note that the algorithm avoids explicitly enumerating the alphabet set for computing extensions of elements in U until required. 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(presented below) 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 artition.

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

Proof. We first show that $v_i \notin V$. Suppose $v_i \in V$. We know that $\alpha_i = \llbracket [u_i]^r \cdot o_i \cdot v_i \rrbracket = 0$ and $\alpha_{i+1} = \llbracket [u_i \cdot o_i]^r \cdot v_i \rrbracket = 1$. Also there must exist $\phi \in follow([u_i]^r)$ so that $o_i \in \phi$ and $T([u_i]^r \cdot \phi, v_i) = T([u_i \cdot o_i]^r, v_i) = 1$. Therefore, by definition, $\forall a \in \phi$, $\llbracket [u_i]^r \cdot a \cdot v_i \rrbracket = 1$. But, $o_i \in \phi$ and $\llbracket [u_i]^r \cdot o_i \cdot v_i \rrbracket = 0$. Contradiction.

Let ua = $([u_i]^r \cdot \phi)$ and $u' = ([[u_i]^r \cdot \phi]^r)$. Adding v_i to V makes $ua \not\equiv u'$ which were equivalent earlier. Moreover, since u' must be in U already, both ua and u' must be inequivalent to all other $u \in U$. Therefore, Close_Table must add ua to U and therefore Mk_DFA will add at least one new state in the next hypothesis.

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.*

Proof. If $v_i \notin V$, we can argue that at least one state will be added in a way similar to the previous lemma. If $v_i \in V$, then we know that $[\![[u_i]^r \cdot o_i \cdot v_i]\!] = 1$ and there exists $\phi \in follow([u_i]^r)$ so that $o_i \in \phi$ and $[\![[u_i]^r \cdot \phi \cdot v_i]\!] = 0$. In this case, Learn_CE_1 splits the cluster ϕ into $\phi_1 = \phi \wedge o_i$ and $\phi_2 = \phi \wedge \neg o_i$. It follows from definition of $[\![\cdot]\!]$ that $[\![[u_i]^r \cdot \phi_1 \cdot v_i]\!] = 1$ and $[\![[u_i]^r \cdot \phi_2 \cdot v_i]\!] = 0$. Hence, ϕ_1 and ϕ_2 must go to different states, causing addition of at least one transition.

Remark. Although Learn_CE_1 may add a transition, the number of states in the next hypothesis may decrease. This is because partitioning a cluster may also cause a state partition causing $[u_i]^r$ to split into two previously existing states, i.e., the new partitioned traces may become equivalent to some previously existing elements of U.

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 $\mathbb{L}_U$.*

Proof. Consider the prefix tree PT obtained from the prefix-closed set of elements in U. Note that each node in PT corresponds to a different state (equivalence class) in a hypothesis CFA C. Also, consider computation tree CT obtained by unrolling the transition structure of C_m. Note that PT of depth d can be embedded into CT where different nodes in PT at a given depth k ($k \leq d$) correspond to different (possibly overlapping) subset of states in CT at depth k. Learn_CE_0 introduces a new node in PT while Learn_CE_1 partitions an alphabet cluster outgoing from some node in PT, so that the size of each of the new clusters is smaller. It is sufficient (with respect to adding and removing states) to consider an PT_f of depth $d = 2^n$ since each node in PT_f corresponds to (i) an element $u \in U$ where $T(u)$ is unique for each u and also (ii)to a subset of states reachable at depth $|u|$ in C_m. Note that a node may be removed from PT only if an outgoing cluster of one of its ancestor nodes can be partitioned. Now since Learn_CE_1 always partitions some cluster in the prefix tree into smaller ones, this can happen only k number of times for the nodes at a given depth in PT_f until each transition corresponds to a single alphabet symbol. Using induction on depth of PT_f, it follows that the clusters at all nodes in PT_f will be fully partitioned in at most $k \cdot 2^n$ iterations. Therefore, the algorithm will make at most $(k \cdot 2^n)$ calls to Learn_CE (or candidate queries) before terminating.

4.3 Optimizing l^*

Although the complexity is bad (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 context of AGR, we seldom need to learn C_m completely; often, an approximation obtained at an intermediate learning step is sufficient. We now propose several optimizations to the basic l^* algorithm outlined above.

Coarser Cluster partitioning using ce generalization. Recall that $ce = u_i \cdot a \cdot v_{i+1}$ where a is a label on $X \cup X'$. Let $u_r = [u_i]^r$. Cluster partitioning occurs in the `Learn_CE_1` procedure where $[\![u_r \cdot a \cdot v_i]\!] = 1$ and $[\![u_r \cdot \phi \cdot v_i]\!] = 0$. The `Partition_Table` procedure uses the symbol a (called the *refining predicate*) to partition the cluster ϕ in $follow(u_r)$ into ϕ_1 and ϕ_2. Since a is an alphabet symbol, this leads to a fine-grained partitioning of $follow(u_r)$. Moreover, note that multiple *similar* counterexamples may cause repeated partitioning of $follow(u_r)$, which may lead to explicit enumeration of Σ in the worst case. For example, there may be several positive counterexamples of form $u_i \cdot a' \cdot v_{i+1}$ where $a' \in \phi$ and a' differs from a only in a few variable assignments. Therefore, we propose a SAT-based technique that performs a *coarser* partitioning of ϕ by first *enlarging* the refining predicate a to a new predicate, say, A, and then using A to partition ϕ.

Recall that the value of $[\![u_r \cdot a \cdot v_{i+1}]\!]$ is computed using a BMC instance. Given a predicate p over $X \cup X'$, let $E(p)$ represent the BMC formula corresponding to the evaluating $[\![u_r \cdot p \cdot v_{i+1}]\!]$. We know that the formula $E(a)$ is UNSAT while $E(\phi)$ is SAT (cf. Section 3.1). We say that a predicate A is an enlargement of a if $a \Rightarrow A$. We are interested in computing the maximum enlargement A of a so that $E(A)$ is UNSAT. This is equivalent to solving an All-SAT problem [30] and is computationally expensive with SAT. Instead, we propose a greedy approach to compute a *maximal* enlargement of a by using a variable lifting technique [32] in the following way. Since a may be viewed as a conjunction of variable assignment constraints, we iteratively remove these variable assignments to obtain larger enlargements A as long as the formula $E(A)$ remains UNSAT. The procedure `Enlarge` shows the pseudocode for this technique. It can be implemented efficiently using an incremental SAT solver and made efficient by observing the UNSAT core obtained at each iteration [35].

```
Enlarge ($E$, $a$)
  $A = a$
  // $A$ is a set of constraints of form $(x_i = d_i)$
  Loop:
    Pick a new constraint $x_i = d_i$ in $A$; If impossible, return $A$
    $A := A \setminus \{(x_i = d_i)\}$
    if ($E(A)$ is SAT)
      $A := A \cup \{(x_i = d_i)\}$
```

Lemma 5 *The procedure* `Enlarge` *finds a maximal enlargement A_m of a when it terminates. Also, A_m must partition the cluster ϕ into two disjoint clusters.*

Proof. Note that $E(p)$ can be written as $F \wedge p$ for some formula F. We know that $E(a)$ is UNSAT and $E(\phi)$ is SAT. `Enlarge` must terminate with at least one constraint in A, since $E(\text{TRUE})$ is SAT. ($E(\text{TRUE}) = F \wedge \text{TRUE} = F \wedge (\phi \vee \neg\phi) = E(\phi) \vee f'$ for some formula f'). It is clear from the pseudocode that A_m computed on termination is maximal.

Since $a \Rightarrow \phi$ and $a \Rightarrow A_m$, so $\phi \wedge A_m \neq \text{FALSE}$. Hence A_m must split ϕ into two disjoint clusters $\phi_1 = \phi \wedge A_m$ and $\phi_2 = \phi \wedge \neg A_m$.

Follow transfer on partitioning. Recall that `Partition_Table` procedure partitions the cluster ϕ in follow set of u_r into ϕ_1 and ϕ_2 and removes all extensions $Uext$ of u_r. However, this may lead to loss of information about follow sets of elements of $Uext$ which may be useful later. We therefore copy the follow set information for each $u \in Uext$ (u = $u_r \cdot \phi \cdot v$ for some v) to the corresponding partitioned traces, $u_r \cdot \phi_1 \cdot v$ and $u_r \cdot \phi_2 \cdot v$.

Reusing $v \in V$**.** In the `Learn_CE_1` algorithm, it is possible that $v_{i+1} \notin V$. Instead of eagerly adding v_{i+1} to set V, we check if some $v \in V$ can act as a substitute for v_{i+1}, i.e., $\alpha_i = 1$ and $\alpha_{i+1} = 0$ with v substituted for v_{i+1}. If we find such v, we use the other case in `Learn_CE_1` which performs cluster partitioning. Intuitively, adding an element to V may cause unnecessary state partitions corresponding to other elements in U, while reusing a previous element in V will lead to a cluster partition whose effect will be local to $u_i \cdot \phi$ and its successors.

Membership Cache and Counterexample History The results of all membership queries are stored in a *membership cache* so that multiple queries on the same trace are not brought to the teacher each time, but instead looked up in the cache. We also keep a *counterexample history* set, which stores all the counterexamples provided by the teacher. Before making a candidate query, we check that the new hypothesis agrees with the unknown language on all the previous counterexamples in the counterexample history set. This is useful because of the lazy nature of the algorithm: it may become necessary to learn again from an older counterexample since the previous learning step only extracted partial information from it.

Another Lazy learning algorithm: l^*_r**.** We briefly discuss another algorithm l^*_r that may be viewed as an extension to a learning algorithm for parameterized systems [9]. Instead of representing the follow sets for each $u \in U$ with one or more alphabet cluster predicates, we only keep a set of representative alphabet symbols in the follow set for each u and a function F_u which maps each representative symbol to an alphabet cluster. As a result, although the observation table only contains a selected set of extensions for each $u \in U$ from the follow set, we can still obtain a complete CFA using the function F_u for each u. Also, in contrast to l^*, the table elements correspond to single traces (as in the original L^* algorithm). Moreover, its complexity is the same as that of the original L^* algorithm [33]. In order to perform counterexample analysis, the original algorithm [9] adds all prefixes of a counterexample to U. Instead, we propose to perform more sophisticated counterexample analysis in tune with l^* as follows. In order to learn from a counterexample ce, two cases need to be differentiated. First case simply adds a suffix to the set V causing a state-partition; the second case leads to addition of a new representative element in a follow set, which causes splitting of alphabet cluster in the range set of the function F_u for some u. Like l^*, l^*_r does not need to restart learning after a follow set is updated. In contrast to l^*_r, the elements of the follow set in l^* are alphabet clusters, where each alphabet cluster corresponds to a set of representative elements instead of a single one as in l^*_r. We compare l^* (Lazy-AGR) with l^*_r (P-AGR) in the next section.

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.

The input language of the tool is a simple intermediate language (SIL), which allows specification of a set of concurrent modules which execute synchronously. Each module may have its internal variables and communicates with other modules using global variables. Variables are of boolean, enumerated and bit-vector types and sequential logic are specified in terms of control flow based on guarded commands. Each module may also have a block of combinational logic in terms of boolean equations. In order to evaluate our approach, we translate both SMV and Verilog programs into SIL. Translator from Verilog to SIL is implemented using the ICARUS Verilog parser. Translation from SMV is done using a python script. The encoding of programs into formula is done as follows. We translate enumerated types to integers with bound constraints. Bit-vector variables are "bit-blasted" currently. We check the correctness of the translation by monolithic SAT-based model checking on the translated models. We use the incremental SMT solver YICES [3, 17] as the main decision procedure. Interpolants are obtained using the library interface to the FOCI tool [1]. 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 [31, 27] (BDD-AGR), our SAT-based approach using l^* (Lazy-AGR) and (P-AGR), which uses a learning algorithm for parameterized systems [9]. 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 [27] while $peterson$ and CC are obtained from the VIS and Texas97 benchmark sets [2]. 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 $s1a$ and $s2a$ (cf. Table 1); although they are difficult for BDD-based model checking [31], 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.

Example	TV	GV	T/F	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	T	2	754	2	223	3	3	3	3	3	3.5	3	1.3
s1b	94	5	T	2	TO	2	1527	3	3.3	3	3.3	3	3.9	3	2
guidance	122	22	T	2	196	2	6.6	1	31.5^i	5	146^i	1	40^i	3	55^i
msi(3)	57	22	T	2	2.1	2	0.3	1	8	*	TO	1	8	3	17
msi(5)	70	25	T	2	1183	2	32	1	16	*	TO	1	15	3	43
syncarb	21	15	T	-	-	67	30	*	TO^i	*	TO^i	*	TO^i	*	TO^i
peterson	13	7	T	-	-	34	2	6	53^i	8	210^i	6	13	6	88^i
CC(2a)	78	30	T	-	-	-	-	1	8	*	TO	1	8	4	26
CC(3a)	115	44	T	-	-	-	-	1	8	*	TO	1	7	4	20
CC(2b)^i	78	30	T	-	-	-	-	*	TO	*	TO	10	1878	5	87
CC(3b)^i	115	44	T	-	-	-	-	*	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 [9] while Lazy-AGR uses l^*. TV and GV represent the number of total and global boolean variables respectively. 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.

Example	T/F	with CE Gen	w/o CE Gen
s1a	T	1.3	1.1
s1b	T	2	1.87
s2a	F	26	TO
s2b	T	36	TO
msi(5)	T	43	86
guidance	T	55	57
Peterson	T	13	175
CC(3b)	T	2143	TO

Table 2: Effect of the counterexample generalization optimization on the l^* algorithm.

6 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 an off-the-shelf SMT solver, we can directly leverage future improvements in SAT/SMT technology. We are also investigating techniques to exploit incremental SAT solving for answering queries for a particular AGR premise, e.g., since we need to check $M \parallel A \vDash P$ repeatedly for many different assumptions A, we could add and remove constraints corresponding to A at each iteration while retaining the rest of the constraints corresponding to M and P. Finally, the problem of finding good system decompositions for allowing small assumptions needs to be investigated. Although presented for the case of finite-state systems, our technique can be extended to infinite-state systems, where the weakest assumption has a finite bisimulation quotient. It can also be applied to compositional verification of concurrent software by first obtaining a finite state abstraction based on a set of predicate variables and then learning assumptions based on these predicate variables. We also plan to use interpolants to improve coarse cluster partitioning.

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] Foci: An interpolating prover. `http://www.kenmcmil.com/foci.html`.

[2] `http://vlsi.coloradu.edu/~vis/`.

[3] Yices: An smt solver. `http://yices.csl.sri.com/`.

[4] Nina Amla, Xiaoqun Du, Andreas Kuehlmann, Robert P. Kurshan, and Kenneth L. McMillan. An analysis of sat-based model checking techniques in an industrial environment. In *CHARME*, pages 254–268, 2005.

[5] Dana Angluin. Learning regular sets from queries and counterexamples. In *Information and Computation*, volume 75(2), pages 87–106, November 1987.

[6] Roy Armoni, Limor Fix, Ranan Fraer, Scott Huddleston, Nir Piterman, and Moshe Y. Vardi. Sat-based induction for temporal safety properties. *Electr. Notes Theor. Comput. Sci.*, 119(2): 3–16, 2005.

[7] Thomas Ball and Sriram K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical report MSR-TR-2002-09, Microsoft Research, Redmond, Washington, USA, January 2002.

[8] H. Barringer, D. Giannakopoulou, and C.S Pasareanu. Proof rules for automated compositional verification. In *2nd Workshop on Specification and Verification of Component-Based Systems, ESEC/FSE 2003.*

[9] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines with parameters. In *FASE*, pages 107–121, 2006.

[10] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Y. Zue. *Bounded Model Checking*, volume 58 of *Advances in computers*. Academic Press, 2003.

[11] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Ziyad Hanna, Zurab Khasidashvili, Amit Palti, and Roberto Sebastiani. Encoding rtl constructs for mathsat: a preliminary report. *Electr. Notes Theor. Comput. Sci.*, 144(2):3–14, 2006.

[12] Sagar Chaki and Ofer Strichman. Optimized L* for assume-guarantee reasoning. In *TACAS*, 2007. To Appear.

[13] Sagar Chaki, Edmund Clarke, Nishant Sinha, and Prasanna Thati. Automated assume-guarantee reasoning for simulation conformance. In *Proc. of 17th Int. Conf. on Computer Aided Verification*, 2005.

[14] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, September 2003.

[15] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, volume 2619. Springer-Verlag, 2003.

[16] Jamieson M. Cobleigh, George S. Avrunin, and Lori A. Clarke. Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In *ISSTA*, pages 97–108, 2006.

[17] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, pages 81–94, 2006.

[18] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electr. Notes Theor. Comput. Sci.*, 89(4), 2003.

[19] Mihaela Gheorghiu, Dimitra Giannakopoulou, and Corina S. Pasareanu. Refining interface alphabets for compositional verification. In *TACAS*, 2007. To Appear.

[20] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '02)*, volume 37(1) of *SIGPLAN Notices*, pages 58–70. ACM Press, January 2002. ISBN 1-58113-450-9.

[21] JE Hopcroft and JD Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[22] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[23] Robert P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994. ISBN 0-691-03436-2.

[24] Patrick Maier. A set-theoretic framework for assume-guarantee reasoning. In *ICALP*, pages 821–834, 2001.

[25] Kenneth L. McMillan. Interpolation and sat-based model checking. In *CAV*, pages 1–13, 2003.

[26] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.

[27] Wonhong Nam and Rajeev Alur. Learning-based symbolic assume-guarantee reasoning with automatic decomposition. In *ATVA*, pages 170–185, 2006.

[28] Kedar S. Namjoshi and Richard J. Trefler. On the completeness of compositional reasoning. In *Proceedings of the 12th Int. Conference on Computer Aided Verification (CAV2000)*, number 1855, pages 139–153. Springer-Verlag, 2000.

[29] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and models of concurrent systems*. Springer-Verlag New York, Inc., 1985.

[30] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat-based formal verification. *STTT*, 7(2):156–173, 2005.

[31] W. Nam R. Alur, P. Madhusudan. Symbolic compositional verification by learning assumptions. In *Proc. of 17th Int. Conf. on Computer Aided Verification*, 2005.

[32] Kavita Ravi and Fabio Somenzi. Minimal assignments for bounded model checking. In *TACAS*, pages 31–45, 2004.

[33] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. In *Information and Computation*, volume 103(2), pages 299–347, 1993.

[34] Mary Sheeran, Satnam Singh, and Gunnar Stalmarck. Checking safety properties using induction and a sat-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 108–125, London, UK, 2000. Springer-Verlag.

[35] ShengYu Shen, Ying Qin, and Sikun Li. Minimizing counterexample with unit core extraction and incremental sat. In *VMCAI*, pages 298–312, 2005.

[36] C. Tinelli and S. Ranise. SMT-LIB: The Satisfiability Modulo Theories Library. `http://goedel.cs.uiowa.edu/smtlib/`, 2005.

Appendix

Given a label a over X and $Y \subseteq X$, we define the label projection $a\,|_Y = a'$ where $dom(a') = Y$ and $a(y) = a'(y)$ for each $y \in Y$.

Definition 4 (Product of CFAs.) *Given CFAs $C_1 = \langle X_1, Q_1, q0_1, \delta_1, F_1\rangle$ and $C_2 = \langle X_2, Q_2, q0_2, \delta_2, F_2\rangle$, their product $C = C_1 \times C_2$ is a tuple $\langle X, Q, q0, \delta, F\rangle$ where $X = X_1 \cup X_2$, $Q = Q_1 \times Q_2$, $q0 = (q0_1, q0_2)$, $F = F_1 \times F_2$ and for a label c over $X \cup X'$, $q_1, q_1' \in Q_1$ and $q_2, q_2' \in Q_2$, $(q_1', q_2') \in \delta((q_1, q_2), c)$ iff $q_1' \in \delta_1(q_1, c|_{X_1 \cup X_1'})$ and $q_2' \in \delta_2(q_2, c|_{X_2 \cup X_2'})$.*

Lemma 6 *For CFAs C_1 and C_2, $\mathbb{L}(C_1 \times C_2) = \mathbb{L}(C_1) \cap \mathbb{L}(C_2)$.*

Definition 5 (Support set of a Language) *We define the support $Spt(L)$ of a regular language L recursively as follows:*

- *If $L = \mathbb{L}(C)$ for a CFA C with support set X, then $Spt(L) = X$.*
- *If $L = L_1 \cap L_2$ for languages L_1 and L_2, then $Spt(L) = Spt(L_1) \cup Spt(L_2)$.*
- *If $L = \overline{L_1}$, for a language L_1, then $Spt(L) = Spt(L_1)$.*

It follows that for $L = L_1 \cup L_2 = \overline{\overline{L_1} \cap \overline{L_2}}$, $Spt(L) = Spt(L_1) \cup Spt(L_2)$.

Lemma 7 *A regular language with support X is accepted by a CFA with support X.*

Proof. We prove by structural induction over the definition of $Spt(L)$. The base case holds trivially since $L = \mathbb{L}(C)$ for a C with support set X. If $L = L_1 \cap L_2$, by inductive hypothesis, there must exist CFAs C_1 and C_2 where $L_1 = \mathbb{L}(C_1)$ and $L_2 = \mathbb{L}(C_2)$, with support sets X_1 and X_2 respectively, so that $X = X_1 \cup X_2$. Let $C = C_1 \times C_2$. Now, $\mathbb{L}(C) = \mathbb{L}(C_1) \cap \mathbb{L}(C_2) = L$. Therefore, L is accepted by the CFA C whose support set is X. Again, if $L = \overline{L_1}$ on support set X, there exist a CFA C_1 on support set X, so that $\mathbb{L}(C_1) = L_1$. Let C denote the CFA obtained by determinizing and complementing C_1. Note that C has support X and $\mathbb{L}(C) = \overline{\mathbb{L}(C_1)} = L$.

In the proof of the following theorems, we use propositional logic notation for representing intersection, complementation and union of languages. Let $lm_1 = \mathbb{L}(M_1)$ and $lm_2 = \mathbb{L}(M_2)$ and $lp = \mathbb{L}(P)$. Given an assumption CFA A_1 and A_2, $la_1 = \mathbb{L}(A_1)$ and $la_2 = \mathbb{L}(A_2)$. The weakest assumption language is $lwa_i = \neg(lm_i) \vee lp$ ($i \in \{1, 2\}$).

Theorem 2 *Rule* **NC** *is sound and complete.*

Proof.

- *Soundness.* We need to show that if $lm_1 \wedge la_1 \implies lp$ and $lm_2 \implies la_1$ hold, then $lm_1 \wedge lm_2 \implies lp$ holds. Since, $lm_2 \implies la_1$, therefore $lm_1 \wedge lm_2 \implies lm_1 \wedge la_1 \implies lp$.

- *Completeness.* We show that if $lm_1 \wedge lm_2 \implies lp$, then both the premises of **NC** hold for the weakest assumption. The first premise, $lm_1 \wedge la_1 \implies lp$ holds since $lm_1 \wedge \neg(lm_1 \vee lp)$ = $lm_1 \wedge lp$ and $lm_1 \wedge lp \implies lp$.

Theorem 3 *Rule* **C** *is sound and complete.*

Proof.

- *Soundness.* We need to show that if $lm_i \wedge la_i \implies lp$ ($i \in \{1, 2\}$) and $\neg la_1 \wedge \neg la_2 \implies lp$, then $lm_1 \wedge lm_2 \implies lp$. Given a trace $t \in lm_1 \wedge lm_2$, we consider three cases:
  - $t \in \neg la_1 \wedge \neg la_2$: It follows from the third premise that $t \in lp$.
  - $t \in la_1$: Since $t \in lm_1$, it follows from the first premise that $t \in lp$.
  - $t \in la_2$: Since $t \in lm_2$, it follows from the second premise that $t \in lp$.
- *Completeness.* Given (i) $lm_1 \wedge lm_2 \implies lp$, we show that all the three premises hold for the weakest assumption languages lwa_1 and lwa_2. The first two premises hold by definition. Note that for $i \in \{0, 1\}$, $\neg lwa_i$ = $lm_i \wedge \neg lp$. Now, $\neg lwa_1 \wedge \neg lwa_2$ = $lm_1 \wedge lm_2 \wedge \neg lp$ which implies lp due to (i).