Automated Assume-Guarantee Reasoning for Simulation Conformance

Sagar Chaki, Edmund Clarke, Nishant Sinha, Prasanna Thati

chaki@sei.cmu.edu {emc,nishants,thati}@cs.cmu.edu

Abstract. We address the issue of efficiently automating assume-guarantee reasoning for simulation conformance between finite state systems and specifications. We focus on a non-circular assume-guarantee proof rule, and show that there is a weakest assumption that can be represented canonically by a deterministic tree automata (DTA). We then present an algorithm L^T that learns this DTA automatically in an incremental fashion, in time that is polynomial in the number of states in the equivalent minimal DTA. The algorithm assumes a teacher that can answer membership and candidate queries pertaining to the language of the unknown DTA. We show how the teacher can be implemented using a model checker. We have implemented this framework in the ComFoRT toolkit and we report encouraging results (over an order of magnitude improvement in memory consumption) on non-trivial benchmarks.

1 Introduction

Formal verification is an important tool in the hands of software practitioners for ascertaining correctness of safety critical software systems. However, scaling formal techniques like model checking [11] to concurrent software of industrial complexity remains an open challenge. The primary hurdle is the state-space explosion problem whereby the number of reachable states of a concurrent system increases exponentially with the number of components.

Two paradigms hold the key to alleviating state-space explosion – abstraction [10, 9] and compositional reasoning [23, 8]. Both of these techniques have been extensively studied by the formal verification community and there have been significant breakthroughs from time to time. One of the most important advancements in the domain of compositional analysis is the concept of assume-guarantee [23] (AG) reasoning. The essential idea here is to model-check each component independently by making an assumption about its environment, and then discharge the assumption on the collection of the rest of the components. A variety of AG proof-rules are known, of which we will concern ourselves with the following non-circular rule called **AG-NC**:

$$\frac{M_1 \parallel M_A \preccurlyeq S \qquad\qquad M_2 \preccurlyeq M_A}{M_1 \parallel M_2 \preccurlyeq S}$$

where $M_1 \parallel M_2$ is the concurrent system to be verified, S is the specification, and $\preccurlyeq$ an appropriate notion of conformance between the system and the specification. **AG-NC** is known to be sound and complete for a number of conformance notions, including trace containment and simulation. The rule essentially states that if there is an *assumption* M_A that satisfies the two premises, then the system conforms to the specification. However, the main drawback here from a practical point of view is that, in general, the assumption M_A has to be constructed manually. This requirement of manual effort has been a major hindrance towards wider applicability of AG-style reasoning on realistic systems.

An important development in this context is the recent use of automata-theoretic learning algorithms by Cobleigh et al. [12] to automate AG reasoning for *trace* containment, when both the system and the specification are finite state machines. Briefly, the idea is to automatically learn an assumption M_A that can be used to discharge **AG-NC**. The specific learning algorithm that is employed is Angluin's L^* [2], which learns finite state machines up to trace equivalence. Empirical evidence [12] indeed suggests that, often in practice, this learning based approach automatically constructs simple (small in size) assumptions that can be used to discharge **AG-NC**.

In this article, we apply the learning paradigm to automate AG-reasoning for *simulation* conformance between finite systems and specifications. We first show that there is a weakest assumption M_W for **AG-NC** such that $M_1 \parallel M_2 \preccurlyeq S$ if and only if $M_2 \preccurlyeq M_W$. Further, M_W is regular in that the set of trees it can simulate can be accepted by a tree automata. Although one can compute M_W and use it to check if $M_2 \preccurlyeq M_W$, doing so would be computationally as expensive as directly checking if $M_1 \parallel M_2 \preccurlyeq S$. We therefore learn the weakest assumption in an *incremental* fashion, and use the successive approximations that are learnt to try and discharge **AG-NC**. If at any stage an approximation is successfully used, then we are done. Otherwise, we extract a counterexample from the premise of **AG-NC** that has failed, and use it to further improve the current approximation.

To realize the above approach, we need an algorithm that learns the weakest assumption up to simulation equivalence. As mentioned above the weakest assumption corresponds to a regular tree language. We present an algorithm L^T that learns the minimal deterministic tree automata (DTA) for this assumption in an incremental fashion. Although a similar learning algorithm for tree languages has been proposed earlier [14], L^T was developed by us independently and has a much better worst-case complexity than the previous algorithm. The algorithm L^T may be of independent interest besides the specific application we consider in this paper. It assumes that an unknown regular tree language U is presented by a *minimally adequate teacher* (teacher for short) that can answer membership queries about U, and that can also test conjectures about U and provide counterexamples to wrong conjectures. The algorithm L^T learns the minimal DTA for U in time polynomial in the number of states in the minimal DTA.

We will show how the teacher can be efficiently implemented in a model checker, i.e., how the membership and candidate queries can be answered without paying the price of explicitly composing M_1 and M_2. Further, we show how while processing the candidate queries, the teacher can try to discharge **AG-NC** with the proposed candidate. We have empirical evidence supporting our claim that **AG-NC** can often be discharged with a coarse approximation (candidate), well before the weakest assumption is learnt. We have implemented the proposed framework in the ComFoRT [7] toolkit and experimented with realistic examples. Specifically, we have experimented with a set of benchmarks constructed from the OPENSSL source code and the SSL specification. The experimental results indicate memory savings by over an order of magnitude compared to a non-AG based approach.

Related Work. A number of applications of machine learning techniques to verification problems have been proposed in the recent past. These include automatic synthesis of interface specifications for application programs [1], automatically learning the set of reachable states in regular model checking [20], *black-box-testing* [22] and its subsequent extension to *adaptive model-checking* [19] to learn an accurate finite state model of an unknown system starting from an approximate one, and learning likely program invariants based on observed values in sample executions [15].

The work we present in this paper closely parallels the approach proposed by Cobleigh et al. [12], where they automate assume-guarantee reasoning for finite state concurrent systems in a trace-containment setting. They show the existence of a weakest environment assumption for an LTS and *automatically* learn successive approximations to it using Angluin's L^* algorithm [2, 24]. Our contribution is to apply this general paradigm to a branching time setting. Further, the L^T algorithm that we present may be of independent interest. L^T may be viewed as a branching time analogue of L^* where the minimally adequate teacher must be capable of answering queries on trees and tree automata (as opposed to traces and finite state machines in L^*). Finally, Rivest et al. [24] proposed an improvement to Angluin's L^* that substantially improves its complexity; our L^T has the same spirit as this improved version of L^*.

Language *identification in the limit* paradigm was introduced by Gold [17]. This forms the basis of *active* algorithms which learn in an online fashion by querying an oracle (teacher); both L^* and L^T fall in this category. Gold also proposed another paradigm, namely *identification from given data*, for learning from a fixed training sample set [18]. The training set consists of a set of positive and negative samples from the unknown language and must be a *characteristic* [18] set of the language. Algorithms have been proposed in this setting for learning word languages [21], tree languages [16, 4] and stochastic tree languages [5]. Unlike the algorithms in [16, 4] which learn tree languages offline from a training set, L^T learns actively by querying a teacher. An anonymous reviewer pointed us to a recently proposed active algorithm for learning tree languages [14], which is closely related to L^T. However, L^T has a better worst-case complexity of $O(n^3)$ as compared to $O(n^5)$ of the previous algorithm. Finally, we note that learning from derivation trees was investigated initially in the context of context-free grammars [25] and forms the basis of several inference algorithms for tree languages [16, 4, 14] including ours.

2 Preliminaries

Definition 1 (Labeled Transition System). *A labeled transition system (LTS) is a 4-tuple $(S, Init, \Sigma, T)$ where (i) S is a finite set of states, (ii) $Init \subseteq S$ is the set of initial states, (iii) Σ is a finite alphabet, and (iv) $T \subseteq S \times \Sigma \times S$ is the transition relation. We write $s \xrightarrow{\alpha} s'$ as a shorthand for $(s, \alpha, s') \in T$.*

Definition 2 (Simulation). *Let $M_1 = (S_1, Init_1, \Sigma_1, T_1)$ and $M_2 = (S_2, Init_2, \Sigma_2, T_2)$ be LTSs such that $\Sigma_1 = \Sigma_2 = \Sigma$ say. A relation $\mathcal{R} \subseteq S_1 \times S_2$ is said to be a simulation relation if:*

$$\forall s_1, s_1' \in S_1 \,\centerdot\, \forall a \in \Sigma \,\centerdot\, \forall s_2 \in S_2 \,\centerdot\, s_1 \mathcal{R} s_2 \wedge s_1 \xrightarrow{a} s_1' \Rightarrow \exists s_2' \in S_2 \,\centerdot\, s_2 \xrightarrow{a} s_2' \wedge s_1' \mathcal{R} s_2'$$

We say M_1 is simulated by M_2, and denote this by $M_1 \preccurlyeq M_2$, if there is a simulation relation $\mathcal{R}$ such that $\forall s_1 \in I_1 \,\centerdot\, \exists s_2 \in I_2 \,\centerdot\, s_1 \mathcal{R} s_2$. We say M_1 and M_2 are simulation equivalent if $M_1 \preccurlyeq M_2$ and $M_2 \preccurlyeq M_1$.

Definition 3 (Tree). *Let λ denote the empty tree and Σ be an alphabet. The set of trees over Σ is defined by the grammar: $T := \lambda \mid \Sigma \bullet T \mid T + T$. The set of all trees over the alphabet Σ is denote by Σ^T, and we let t range over it.*

Definition 4 (Context). *The set of contexts over an alphabet Σ can be defined by the grammar: $C := \Box \mid \Sigma \bullet C \mid C + T \mid T + C$. We let c range over the set of contexts.*

A context is like a tree except that it has exactly one hole denoted by $\Box$ at one of its nodes. When we plug in a tree t in a context c, we essentially replace the single $\Box$ in c by t. The resulting tree is denoted by $c[t]$. A tree t can naturally be seen as an LTS. Specifically, the states of the LTS are the nodes of t, the only initial state is the root node of t, and there is a labeled transition from node t_1 to t_2 labeled with α if $t_1 = \alpha \bullet t_2$ or $t_1 = \alpha \bullet t_2 + t_3$ or $t_1 = t_2 + \alpha \bullet t_3$.

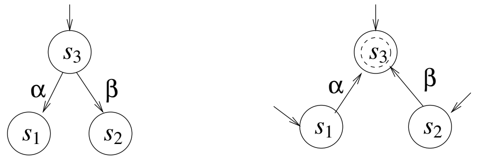
Definition 5 (Tree Language of an LTS). *An LTS M induces a tree language, which is denoted by $\mathcal{T}(M)$ and is defined as: $\mathcal{T}(M) = \{t \mid t \preccurlyeq M\}$. In other words, the tree language of an LTS contains all the trees that can be simulated by the LTS.*

For example, the language of M (Figure 1(a)) contains the trees λ, $\alpha \bullet \lambda$, $\alpha \bullet (\lambda + \lambda)$, $\alpha \bullet \lambda + \beta \bullet \lambda$, $\beta \bullet \lambda + \beta \bullet \lambda$ and so on. The notion of tree languages of LTSs and simulation between LTSs are fundamentally connected. Specifically, it follows from the definition of simulation between LTSs that for any two LTSs M_1 and M_2, the following holds:

$$M_1 \preccurlyeq M_2 \iff \mathcal{T}(M_1) \subseteq \mathcal{T}(M_2) \tag{1}$$

Definition 6 (Tree Automaton). *A (bottom-up) tree automaton (TA) is a 6-tuple $A = (S, Init, \Sigma, \delta, \otimes, F)$ where: (i) S is a set of states, (ii) $Init \subseteq S$ is a set of initial states, (iii) Σ is an alphabet, (iv) $\delta \subseteq S \times \Sigma \times S$ is a forward transition relation, (v) $\otimes \subseteq S \times S \times S$ is a cross transition relation, and (vi) $F \subseteq S$ is a set of accepting states.*

Tree automata accept trees and can be viewed as two-dimensional extensions of finite automata. Since trees can be extended either forward (via the $\bullet$ operator) and across (via the $+$ operator), a TA must have transitions defined when either of these two kinds of extensions of its input tree are encountered. This is achieved via the forward and cross transitions respectively. The automaton starts at each leaf of the input tree at some initial state, and then runs bottom-up in accordance with its forward and cross transition relations. The forward transition is applied when a tree of the form $\alpha \bullet T$ is



(a)

$\otimes$	s_1	s_2	s_3
s_1	s_1		
s_2		s_2	
s_3			s_3

(b)

Fig. 1. (a-left) an LTS M with initial state s_3; (a-right) forward transitions of a tree automaton A accepting $\mathcal{T}(M)$; all states are initial; (b) table showing cross transition relation $\otimes$ of A. Note that some table entries are absent since the relation $\otimes$ is not total.

encountered. The cross transition is applied when a tree of the form $T_1 + T_2$ is found. The tree is accepted if the run ends at the root of the tree in some accepting state of A.

Before we formally define the notions of runs and acceptance, we introduce a few notational conventions. We may sometimes write $s \xrightarrow{\alpha} s'$ or $s' \in \delta(s, \alpha)$ as a shorthand for $(s, \alpha, s') \in \delta$, and $s_1 \otimes s_2 \longrightarrow s$ as a shorthand for $(s_1, s_2, s) \in \otimes$. Similarly, for sets of states S_1, S_2, we use the following shorthand notations:

$$\delta(S_1, \alpha) = \{s' \mid \exists s \in S_1 \centerdot s \xrightarrow{\alpha} s'\}$$
$$S_1 \otimes S_2 = \{s \mid \exists s_1 \in S_1 \centerdot \exists s_2 \in S_2 \centerdot (s_1, s_2, s) \in \otimes\}$$

Definition 7 (Run/Acceptance). *Let $A = (S, Init, \Sigma, \delta, \otimes, F)$ be a TA. The run of A is a function $r : \Sigma^T \to 2^S$ from trees to sets of states of A that satisfies the following conditions: (i) $r(\lambda) = Init$, (ii) $r(\alpha \bullet t) = \delta(r(t), \alpha)$, and (iii) $r(t_1 + t_2) = r(t_1) \otimes r(t_2)$. A tree T is accepted by A iff $r(T) \cap F \neq \emptyset$. The set of trees accepted by A is known as the language of A and is denoted by $\mathcal{L}(A)$.*

A *deterministic* tree automaton (DTA) is one which has a single initial state and where the forward and cross transition relations are *functions* $\delta : S \times \Sigma \to S$ and $\otimes : S \times S \to S$ respectively. If $A = (S, Init, \Sigma, \delta, \otimes, F)$ is a DTA then $Init$ refers to the single initial state, and $\delta(s, \alpha)$ and $s_1 \otimes s_2$ refer to the unique state s' such that $s \xrightarrow{\alpha} s'$ and $s_1 \otimes s_2 \longrightarrow s'$ respectively. Note that if A is deterministic then for every tree t the set $r(t)$ is a singleton, i.e., the run of A on any tree t ends at a unique state of A. Further, we recall [13] the following facts about tree-automata. The set of languages recognized by TA (referred to as *regular tree languages* henceforth) is closed under union, intersection and complementation. For every TA A there is a DTA A' such that $\mathcal{L}(A) = \mathcal{L}(A')$. Given any regular tree language L there is always a *unique* (up to isomorphism) *smallest* DTA A such that $\mathcal{L}(A) = L$.

The following lemma, which is easy to prove, asserts that for any LTS M, the set $\mathcal{T}(M)$ is a regular tree language. Thus, using (1), the simulation problem between LTSs can also be viewed as the language containment problem between tree automata.

Lemma 1. *For any LTS M there is a TA A such that $\mathcal{L}(A) = \mathcal{T}(M)$.*

For example, for the LTS M and TA A as shown in Figure 1, we have $\mathcal{L}(A) = \mathcal{T}(M)$. We now provide the standard notion of parallel composition between LTSs, where components synchronize on shared actions and proceed asynchronously on local actions.

Definition 8 (Parallel Composition of LTSs). *Given LTSs* $M_1 = (S_1, Init_1, \Sigma_1, T_1)$ *and* $M_2 = (S_2, Init_2, \Sigma_2, T_2)$*, their parallel composition* $M_1 \parallel M_2$ *is an LTS* $M = (S, Init, \Sigma, T)$ *where* $S = S_1 \times S_2$*,* $Init = Init_1 \times Init_2$*,* $\Sigma = \Sigma_1 \cup \Sigma_2$*, and the transition relation* T *is defined as follows:* $((s_1, s_2), \alpha, (s_1', s_2')) \in T$ *iff for* $i \in \{1, 2\}$ *the following holds:*

$$(\alpha \in \Sigma_i) \wedge (s_i, \alpha, s_i') \in T_i \quad \bigvee \quad (\alpha \notin \Sigma_i) \wedge (s_i = s_i')$$

Working with different alphabets for each component would needlessly complicate the exposition in Section 4. For this reason, without loss of generality, we make the simplifying assumption that $\Sigma_1 = \Sigma_2$. This is justified because we can construct LTSs M_1' and M_2', each with the same alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$ such that $M_1' \parallel M_2'$ is simulation equivalent (in fact bisimilar) to $M_1 \parallel M_2$. Specifically, $M_1' = (S_1, Init_1, \Sigma, T_1')$ and $M_2' = (S_2, Init_2, \Sigma, T_2')$ where

$$T_1' = T_1 \cup \{(s, \alpha, s) \mid s \in S_1 \text{ and } \alpha \in \Sigma_2 \setminus \Sigma_1\}$$
$$T_2' = T_2 \cup \{(s, \alpha, s) \mid s \in S_2 \text{ and } \alpha \in \Sigma_1 \setminus \Sigma_2\}$$

Finally, the reader can check that if M_1 and M_2 are LTSs with the same alphabet then $\mathcal{T}(M_1 \parallel M_2) = \mathcal{T}(M_1) \cap \mathcal{T}(M_2)$.

3 Learning Minimal DTA

We now present the algorithm L^T that learns the minimal DTA for an unknown regular language U. It is assumed that the alphabet Σ of U is fixed, and that the language U is presented by a minimally adequate teacher that answers two kinds of queries:

1. *Membership.* Given a tree t, is t an element of U, i.e., $t \in U$?
2. *Candidate.* Given a DTA A does A accept U, i.e., $\mathcal{L}(A) = U$? If $\mathcal{L}(A) = U$ the teacher returns TRUE, else it returns FALSE along with a counterexample tree CE that is in the symmetric difference of $\mathcal{L}(A)$ and U.

We will use the following notation. Given any sets of trees S_1, S_2 and an alphabet Σ we denote by $\Sigma \bullet S_1$ the set of trees $\Sigma \bullet S_1 = \{\alpha \bullet t \mid \alpha \in \Sigma \wedge t \in S_1\}$, and by $S_1 + S_2$ the set $S_1 + S_2 = \{t_1 + t_2 \mid t_1 \in S_1 \wedge t_2 \in S_2\}$, and by $\widehat{\mathcal{S}}$ the set $\mathcal{S} \cup (\Sigma \bullet \mathcal{S}) \cup (\mathcal{S} + \mathcal{S})$.

<u>Observation Table</u> : The algorithm L^T maintains an observation table $\tau = (\mathcal{S}, \mathcal{E}, \mathcal{R})$ where (i) $\mathcal{S}$ is a set of trees such that $\lambda \in \mathcal{S}$, (ii) $\mathcal{E}$ is a set of contexts such that $\Box \in \mathcal{E}$, and (iii) $\mathcal{R}$ is a function from $\widehat{\mathcal{S}} \times \mathcal{E}$ to $\{0, 1\}$ that is defined as follows: $\mathcal{R}(t, c) = 1$ if $c[t] \in U$ and 0 otherwise. Note that given $\mathcal{S}$ and $\mathcal{E}$ we can compute $\mathcal{R}$ using membership queries. The information in the table is eventually used to construct a candidate DTA A_τ. Intuitively, the elements of $\mathcal{S}$ will serve as states of A_τ, and the contexts in $\mathcal{E}$ will play the role of *experiments* that distinguish the states in $\mathcal{S}$. Henceforth, the term experiment will essentially mean a context. The function $\mathcal{R}$ and the elements in $\widehat{\mathcal{S}} \setminus \mathcal{S}$ will be used to construct the forward and cross transitions between the states.

For any tree $t \in \widehat{\mathcal{S}}$, we denote by $Row(t)$ the function from the set of experiments $\mathcal{E}$ to $\{0, 1\}$ defined as: $\forall c \in \mathcal{E} \centerdot Row(t)(c) = \mathcal{R}(t, c)$.

	$\Box$
λ	1 (s_0)
$\alpha \bullet \lambda$	1
$\beta \bullet \lambda$	1
$\lambda + \lambda$	1

(a)

δ	α	β
s_0	s_0	s_0

(b)

$\otimes$	s_0
s_0	s_0

(c)

Fig. 2. (a) A well-formed and closed observation table τ; (b) forward transition relation of the candidate A^1_τ constructed from τ; (c) cross transition relation of A^1_τ.

Definition 9 (Well-formed). *An observation table $(\mathcal{S}, \mathcal{E}, \mathcal{R})$ is said to be well-formed if: $\forall t, t' \in \mathcal{S} \centerdot t \neq t' \Rightarrow Row(t) \neq Row(t')$. From the definition of $Row(t)$ above, this boils down to: $\forall t, t' \in \mathcal{S} \centerdot t \neq t' \Rightarrow \exists c \in \mathcal{E} \centerdot \mathcal{R}(t, c) \neq \mathcal{R}(t', c)$.*

In other words, any two different row entries of a well-formed observation table must be distinguishable by at least one experiment in $\mathcal{E}$. The following crucial lemma imposes an upper-bound on the size of any well-formed observation table corresponding to a given regular tree language U.

Lemma 2. *Let $(\mathcal{S}, \mathcal{E}, \mathcal{R})$ be any well-formed observation table for a regular tree language U. Then $|\mathcal{S}| \leq n$, where n is the number of states of the smallest DTA which accepts U. In other words, the number of rows in any well-formed observation table for U cannot exceed the number of states in the smallest DTA that accepts U.*

Proof. The proof is by contradiction. Let A be the smallest DTA accepting U and let $(\mathcal{S}, \mathcal{E}, \mathcal{R})$ be a well-formed observation table such that $|\mathcal{S}| > n$. Then there are two distinct trees t_1 and t_2 in $\mathcal{S}$ such that the runs of A on both t_1 and t_2 end on the same state of A. Then for any context c, the runs of A on $c[t_1]$ and $c[t_2]$ both end on the same state. But on the other hand, since the observation table is well-formed, there exists an experiment $c \in \mathcal{E}$ such that $\mathcal{R}(t_1, c) \neq \mathcal{R}(t_2, c)$, which implies that the runs of A on $c[t_1]$ and $c[t_2]$ end on different states of A. Contradiction. $\Box$

Definition 10 (Closed). *An observation table $(\mathcal{S}, \mathcal{E}, \mathcal{R})$ is said to be closed if*

$$\forall t \in \widehat{\mathcal{S}} \setminus \mathcal{S} \centerdot \exists t' \in \mathcal{S} \centerdot Row(t') = Row(t)$$

Note that, given any well-formed observation table $(\mathcal{S}, \mathcal{E}, \mathcal{R})$, one can always construct a well-formed and closed observation table $(\mathcal{S}', \mathcal{E}, \mathcal{R}')$ such that $\mathcal{S} \subseteq \mathcal{S}'$. Specifically, we repeatedly try to find an element t in $\widehat{\mathcal{S}} \setminus \mathcal{S}$ such that $\forall t' \in \mathcal{S} \centerdot Row(t') \neq Row(t)$. If no such t can be found then the table is already closed and we stop. Otherwise, we add t to $\mathcal{S}$ and repeat the process. Note that, the table always stays well-formed. Then by Lemma 2, the size of $\mathcal{S}$ cannot exceed the number of states of the smallest DTA that accepts U. Hence this process always terminates.

Figure 2a shows a well-formed and closed table with $\mathcal{S} = \{\lambda\}$, $\mathcal{E} = \{\Box\}$, $\Sigma = \{\alpha, \beta\}$, and for the regular tree language defined by the TA in Figure 1. Note that $Row(t) = Row(\lambda)$ for every $t \in \{\alpha \bullet \lambda, \beta \bullet \lambda, \lambda + \lambda\}$, and hence the table is closed.

Conjecture Construction: From a well-formed and closed observation table $\tau = (\mathcal{S}, \mathcal{E}, \mathcal{R})$, the learner constructs a candidate DTA $A_\tau = (S, \mathit{Init}, \Sigma, \delta, \otimes, F)$ where (i) $S = \mathcal{S}$, (ii) $\mathit{Init} = \lambda$, (iii) $F = \{t \in \mathcal{S} \mid \mathcal{R}(t, \Box) = 1\}$, (iv) $\delta(t, \alpha) := t'$ such that $\mathit{Row}(t') = \mathit{Row}(\alpha \bullet t)$, and (v) $t_1 \otimes t_2 := t'$ such that $\mathit{Row}(t') = \mathit{Row}(t_1 + t_2)$. Note that in (iv) and (v) above there is guaranteed to be a unique such t' since τ is closed and well-formed, hence A_τ is well-defined.

Consider again the closed table in Figure 2a. The learner extracts a conjecture A_τ from it with a single state s_0, which is both initial and final. Figures 2b and 2c show the forward and cross transitions of A_τ.

The Learning Algorithm: The algorithm L^T is iterative and always maintains a well-formed observation table $\tau = (\mathcal{S}, \mathcal{E}, \mathcal{R})$. Initially, $\mathcal{S} = \{\lambda\}$ and $\mathcal{E} = \{\Box\}$. In each iteration, L^T proceeds as follows:

1. Make τ closed as described previously.
2. Construct a conjecture DTA A_τ from τ, and make a candidate query with A_τ. If A_τ is a correct conjecture, then L^T terminates with A_τ as the answer. Otherwise, let CE be the counterexample returned by the teacher.
3. Extract a context c from CE, add it to $\mathcal{E}$, and proceed with the next iteration from step 1. The newly added c is such that when we make τ closed in the next iteration, the size of $\mathcal{S}$ is guaranteed to increase.

Extracting an Experiment From CE: Let r be the run function of the failed candidate A_τ. For any tree t, let $\tau(t) = r(t)$, i.e., $\tau(t)$ is the state at which the run of A_τ on t ends. Note that since states of A_τ are elements in $\mathcal{S}$, $\tau(t)$ is itself a tree. The unknown language U induces a natural equivalence relation $\approx$ on the set of trees as follows: $t_1 \approx t_2$ iff $t_1 \in U \iff t_2 \in U$.

The procedure **ExpGen** for extracting a new experiment from the counterexample is iterative. It maintains a context c and a tree t that satisfy the following condition: **(INV)** $c[t] \not\approx c[\tau(t)]$. Initially $c = \Box$ and $t = CE$. Note that this satisfies **INV** because $CE \in U \iff CE \notin \mathcal{L}(A_\tau)$. In each iteration, **ExpGen** either generates an appropriate experiment or updates c and t such that **INV** is maintained and the size of t strictly decreases. Note that t cannot become λ since at that point **INV** can no longer be maintained; this is because if $t = \lambda$ then $\tau(t) = \lambda$ and therefore $c[t] \approx c[\tau(t)]$, which would contradict **INV**. Hence, **ExpGen** must terminate at some stage by generating an appropriate experiment. Now, there are two possible cases:

Case 1: ($t = \alpha \bullet t'$) Let $c' = c[\alpha \bullet \Box]$. We consider two sub-cases. Suppose that $c[\tau(t)] \approx c'[\tau(t')]$. From **INV** we know that $c[t] \not\approx c[\tau(t)]$. Hence $c'[\tau(t')] \not\approx c[t] \approx c'[t']$. Hence, **ExpGen** proceeds to the next iteration with $c = c'$ and $t = t'$. Note that **INV** is preserved and the size of t strictly decreases.

Otherwise, suppose that $c[\tau(t)] \not\approx c'[\tau(t')]$. In this case, **ExpGen** terminates by adding the experiment c to $\mathcal{E}$. Note that A_τ has the transition $\tau(t') \xrightarrow{\alpha} \tau(t)$, i.e., $\mathit{Row}(\tau(t)) = \mathit{Row}(\alpha \bullet \tau(t'))$. But now, since $c[\tau(t)] \not\approx c'[\tau(t')] \approx c[\alpha \bullet \tau(t')]$, the experiment c is guaranteed to distinguish between $\tau(t)$ and $\alpha \bullet \tau(t')$. Therefore, the size of $\mathcal{S}$ is guaranteed to increase when we attempt to close τ in the next iteration.

	$\square$	$\alpha \bullet \square$	
λ	1	1	(s_0)
$\alpha \bullet \lambda$	1	0	(s_1)
$\alpha \bullet \alpha \bullet \lambda$	0	0	(s_2)
$\beta \bullet \lambda$	1	0	
$\beta \bullet \alpha \bullet \lambda$	0	0	
$\alpha \bullet \alpha \bullet \alpha \bullet \lambda$	0	0	
$\beta \bullet \alpha \bullet \alpha \bullet \lambda$	0	0	
$\lambda + \lambda$	1	1	
$\lambda + \alpha \bullet \lambda$	1	0	
$\alpha \bullet \lambda + \alpha \bullet \lambda$	1	0	
$\lambda + \alpha \bullet \alpha \bullet \lambda$	0	0	
$\alpha \bullet \lambda + \alpha \bullet \alpha \bullet \lambda$	0	0	
$\alpha \bullet \alpha \bullet \lambda + \alpha \bullet \alpha \bullet \lambda$	0	0	

(a)

δ	α	β
s_0	s_1	s_1
s_1	s_2	s_2
s_2	s_2	s_2

(b)

$\otimes$	s_0	s_1	s_2
s_0	s_0	s_1	s_2
s_1	s_1	s_1	s_2
s_2	s_2	s_2	s_2

(c)

Fig. 3. (a) observation table τ and (b) transitions for the second conjecture A^2_τ.

Case 2: ($t = t_1 + t_2$) There are two sub-cases. Suppose that $c[\tau(t)] \not\approx c[\tau(t_1) + \tau(t_2)]$. In this case, **ExpGen** terminates by adding the experiment c to $\mathcal{E}$. The experiment c is guaranteed to distinguish between $\tau(t)$ and $\tau(t_1) + \tau(t_2)$ and therefore strictly increase the size of $\mathcal{S}$ when we attempt to close τ in the next iteration.

Otherwise, suppose that $c[\tau(t)] \approx c[\tau(t_1) + \tau(t_2)]$. We again consider two sub-cases. Suppose that $c[\tau(t_1) + \tau(t_2)] \not\approx c[\tau(t_1) + t_2]$. In this case, **ExpGen** proceeds to the next iteration with $c = c[\tau(t_1) + \square]$ and $t = t_2$. Note that **INV** is preserved and the size of t strictly decreases.

Otherwise, we have $c[\tau(t_1)+t_2] \approx c[\tau(t_1)+\tau(t_2)] \approx c[\tau(t)]$, and by **INV** we know that $c[\tau(t)] \not\approx c[t] \approx c[t_1+t_2]$. Hence, it must be the case that $c[\tau(t_1)+t_2] \not\approx c[t_1+t_2]$. In this case, **ExpGen** proceeds to the next iteration with $c = c[\square + t_2]$ and $t = t_1$. Note that, once again **INV** is preserved and the size of t strictly decreases. This completes the argument for all cases.

Example 1. We show how L^T learns the minimal DTA corresponding to the language U of TA A of Figure 1. L^T starts with an observation table τ with $\mathcal{S} = \{\lambda\}$ and $\mathcal{E} = \{\square\}$. The table is then made closed by asking membership queries, first for λ and then for its (forward and cross) extensions $\{\alpha \bullet \lambda, \beta \bullet \lambda, \lambda + \lambda\}$. The resulting closed table τ_1 is shown in Figure 2a. L^T then extracts a candidate A^1_τ from τ_1, which is shown in Figure 2b.

When the conjecture A^1_τ is presented to the teacher, it checks if $\mathcal{L}(A^1_\tau) = U$. In our case, it detects otherwise and returns a counterexample CE from the symmetric difference of $\mathcal{L}(A^1_\tau)$ and U. For the purpose of illustration, let us assume CE to be $\alpha \bullet \beta \bullet \lambda$. Note that $CE \in \mathcal{L}(A^1_\tau) \setminus U$. The algorithm **ExpGen** extracts the context $\alpha \bullet \square$ from CE and adds it to the set of experiments $\mathcal{E}$. L^T now asks membership queries corresponding to the new experiment and checks if the new table τ is closed. It finds that $Row(\alpha \bullet \lambda) \neq Row(t)$ for all $t \in \mathcal{S}$, and hence it moves $\alpha \bullet \lambda$ from $\widehat{\mathcal{S}} \setminus \mathcal{S}$ to $\mathcal{S}$ in order to make τ closed. Again, membership queries for all possible forward and cross extensions of $\alpha \bullet \lambda$ are asked. This process is repeated till τ becomes closed. Figure 3a shows the final closed τ. As an optimization, we omit rows for the trees $t_1 + t_2$ whenever there is already a row for $t_2 + t_1$; we know that the rows for both these trees will have the same markings. The corresponding conjecture A_τ^2 contains three states s_0, s_1 and s_2 and its forward and cross transitions are shown in Figure 3b and Figure 3c. s_0 is the initial state and both s_0 and s_1 are final states. The candidate query with A_τ^2 returns TRUE since $\mathcal{L}(A_\tau^2) = U$, and L^T terminates with A_τ^2 as the output.

Correctness and Complexity:

Theorem 1. *Algorithm L^T terminates and outputs the minimal DTA that accepts the unknown regular language U.*

Proof. Termination is guaranteed by the facts that each iteration of L^T terminates, and in each iteration $|\mathcal{S}|$ must strictly increase, and, by Lemma 2, $|\mathcal{S}|$ cannot exceed the number of states of the smallest DTA that accepts U. Further, since L^T terminates only after a correct conjecture, if the DTA A_τ is its output then $\mathcal{L}(A_\tau) = U$. Finally, since the number of states in A_τ equals $|S|$, by Lemma 2 it also follows that A_τ is the minimal DTA for U. $\square$

To keep the space consumption of L^T within polynomial bounds, the trees and contexts in $\widehat{\mathcal{S}}$ and $\mathcal{E}$ are kept in a DAG form, where common subtrees between different elements in $\widehat{\mathcal{S}}$ and $\mathcal{E}$ are shared. Without this optimization, the space consumption can be exponential in the worst case. The other point to note is that the time taken by L^T depends on the counterexamples returned by the teacher; this is because the teacher can return counterexamples of any size in response to a failed candidate query.

To analyze the complexity of L^T, we make the following standard assumption: every query to the teacher, whether a membership query or a candidate query, takes unit time and space. Further, since the alphabet Σ of the unknown language U is fixed, we assume that the size of Σ is a constant. Then the following theorem summarizes the complexity of L^T.

Theorem 2. *The algorithm L^T takes $O(mn + n^3)$ time and space where n is the number of states in the minimal DTA for the unknown language U and m is the size of the largest counterexample returned by the teacher.*

Proof. By Lemma 2, we have $|\mathcal{S}| \leq n$. Then the number of rows in the table, which is $|\widehat{\mathcal{S}}| = |\mathcal{S} \cup (\Sigma \bullet \mathcal{S}) \cup (\mathcal{S} + \mathcal{S})|$, is of $O(n^2)$. Further, recall that every time a new experiment is added to $\mathcal{E}$, $|\mathcal{S}|$ increases by one. Hence the number of table columns $|\mathcal{E}| \leq n$, and the number of table entries $|\widehat{\mathcal{S}}||\mathcal{E}|$ is of $O(n^3)$.

The trees and contexts in $\widehat{\mathcal{S}}$ and $\mathcal{E}$ are kept in a DAG form, where common subtrees between different elements in $\widehat{\mathcal{S}}$ and $\mathcal{E}$ are shared in order to keep the space consumption within polynomial bounds. Specifically, recall that whenever a tree t is moved from $\widehat{\mathcal{S}} \backslash \mathcal{S}$ to $\mathcal{S}$, all trees of the form $\alpha \bullet t$ for each $\alpha \in \Sigma$ and $t + t'$ for each $t' \in \mathcal{S}$ (which are $O(|\mathcal{S}|)$ in number) are to be added to $\widehat{\mathcal{S}}$. Adding the tree $\alpha \bullet t$ to $\widehat{\mathcal{S}}$ only needs constant space since t is already in $\widehat{\mathcal{S}}$ and hence is shared in the DAG representation. Similarly adding a tree of form $t + t'$ takes only constant space, since both t and t' are already in $\widehat{\mathcal{S}}$. Thus, each time $\mathcal{S}$ is expanded, a total of $O(|\mathcal{S}|)$ space is required to add all the new trees to $\widehat{\mathcal{S}}$. Since at most n trees can be added $\mathcal{S}$ in all, it follows that the total space consumed by elements in $\widehat{\mathcal{S}}$ is $O(n^2)$.

Now, we compute the total space consumed by the contexts in $\mathcal{E}$. Note that the teacher can return counterexamples of arbitrary size in response to a wrong conjecture. Suppose m is the size of the largest counterexample. Observe that an experiment is extracted from CE (procedure **ExpGen**) essentially by replacing some of the subtrees of CE with trees in $\mathcal{S}$, and exactly one subtree of CE with $\Box$. But, since in the DAG form, common subtrees are shared between trees and contexts in $\mathcal{S}$ and $\mathcal{E}$, none of the above replacements consume any extra space. Hence, the size of the experiment extracted from CE is utmost the size of CE. Since there are at most n contexts in $\mathcal{E}$, the total space consumed by contexts in $\mathcal{E}$ is $O(mn)$. Putting together all observations so far, we get that the total space consumed by L^T is $O(mn + n^3)$.

Now, we compute the time consumed by L^T. It takes $O(n^3)$ membership queries to fill in the $O(n^3)$ table entries. Since each query is assumed to take $O(1)$ time, this takes a total of $O(n^3)$ time. The time taken to extract an experiment from a counterexample CE is linear on the size of CE. This is because procedure **ExpGen** involves making a constant number of membership queries for each node of CE (branch conditions in lines 3, 6, and 8) as CE is processed in a top down fashion. Thus, the time taken to extract an experiment from CE is at most $O(m)$. Since there can be at most n wrong conjectures, the total time spent on processing counterexamples is $O(mn)$. Putting these observations together we conclude that L^T takes $O(mn+n^3)$ time. We thus have the following theorem.

4 Automating Assume-Guarantee for Simulation

For M_1, M_2 and M_S, suppose we are to check if $M_1 \parallel M_2 \preccurlyeq M_S$. Recall from Section 2 that $M_1 \parallel M_2 \preccurlyeq M_S$ if and only if $\mathcal{T}(M_1 \parallel M_2) \subseteq \mathcal{T}(M_S)$, and $\mathcal{T}(M_1 \parallel M_2) = \mathcal{T}(M_1) \cap \mathcal{T}(M_2)$. Therefore, the verification problem is equivalent to checking if $\mathcal{T}(M_1) \cap \mathcal{T}(M_2) \subseteq \mathcal{T}(M_S)$. Now, define $\mathcal{T}_{max} = \overline{\mathcal{T}(M_1) \cap \overline{\mathcal{T}(M_S)}}$. Then

$$\mathcal{T}(M_1) \cap \mathcal{T}(M_2) \subseteq \mathcal{T}(M_S) \iff \mathcal{T}(M_2) \subseteq \mathcal{T}_{max}$$

Thus, $\mathcal{T}_{max}$ represents the maximal environment under which M_1 satisfies M_S, and

$$M_1 \parallel M_2 \preccurlyeq M_S \Leftrightarrow \mathcal{T}(M_2) \subseteq \mathcal{T}_{max}$$

Checking $\mathcal{T}(M_2) \subseteq \mathcal{T}_{max}$ is as expensive as directly checking $M_1 \parallel M_2 \preccurlyeq M_S$ since it involves both M_1 and M_2. In the following, we show how the L^T algorithm can be used for a more efficient solution.

Since regular tree languages are closed under intersection and complementation, $\mathcal{T}_{max}$ is a regular tree language. We therefore use the L^T algorithm to learn the canonical DTA for $\mathcal{T}_{max}$ in an incremental fashion. The key idea is that when a candidate query is made by L^T, the teacher checks if the **AG-NC** proof rule can be discharged by using the proposed candidate as the assumption. Empirical evidence (see Section 5) suggests that this often succeeds well before $\mathcal{T}_{max}$ is learnt, leading to substantial savings in time and memory consumption.

We now elaborate on how the teacher assumed by L^T is implemented. Specifically, the membership and candidate queries of L^T are processed as follows.

Membership Query. For a given tree t we are to check if $t \in \mathcal{T}_{max}$. This is equivalent to checking if $t \notin \mathcal{T}(M_1)$ or $t \in \mathcal{T}(M_S)$. In our implementation, both $\mathcal{T}(M_1)$ and $\mathcal{T}(M_S)$ are maintained as tree automata, and the above check amounts to membership queries on these automata.

Candidate Query. Given a DTA D we are to check if $\mathcal{L}(D) = \mathcal{T}_{max}$. We proceed in three steps as follows.

1. Check if **(C1)** $\mathcal{L}(D) \subseteq \mathcal{T}_{max} = \overline{\mathcal{T}(M_1) \cap \overline{\mathcal{L}(M_S)}}$. This is implemented using the complementation, intersection and emptyness checking operations on tree automata. If **C1** holds, then we proceed to step 2. Otherwise, we return some $t \in \mathcal{T}_{max} \setminus \mathcal{L}(D)$ as a counterexample to the candidate query D.
2. Check if **(C2)** $\mathcal{T}(M_2) \subseteq \mathcal{L}(D)$. If this is true, then **(C1)** and **(C2)** together imply that $\mathcal{T}(M_2) \subseteq \mathcal{T}_{max}$, and thus our overall verification procedure terminates concluding that $M_1 \parallel M_2 \preccurlyeq M_S$. Note that even though the procedure terminates $\mathcal{L}(D)$ may not be equal to $\mathcal{T}_{max}$. On the other hand, if **(C2)** does not hold, we proceed to step 3 with some $t \in \mathcal{T}(M_2) \setminus \mathcal{L}(D)$.
3. Check if $t \in \mathcal{T}_{max}$, which is handled as in the membership query above. If this is true, then it follows that $t \in \mathcal{T}_{max} \setminus \mathcal{L}(D)$, and hence we return t as a counterexample to the candidate query D. Otherwise, if $t \notin \mathcal{T}_{max}$ then $\mathcal{T}(M_2) \not\subseteq \mathcal{T}_{max}$, and therefore we conclude that $M_1 \parallel M_2 \not\preccurlyeq M_S$.

Thus, the procedure for processing the candidate query can either answer the query or terminate the entire verification procedure with a positive or negative outcome. Further, the reader may note that M_1 and M_2 are never considered together in any of the above steps. For instance, the candidate D is used instead of M_1 in step 1, and instead of M_2 in step 2. Since D is typically very small in size, we achieve significant savings in time and memory consumption, as reported in Section 5.

5 Experimental Results

Our primary target has been the analysis of concurrent message-passing C programs. Specifically, we have experimented with a set of benchmarks derived from the OpenSSL-0.9.6c source code. We analyzed the source code that implements the critical handshake that occurs when an SSL server and client establish a secure communication channel between them. The server and client source code contained roughly 2500 LOC each. Since these programs have an infinite state space, we constructed finite conservative labeled transition system (LTS) models from them using various abstraction techniques [6][1]. The abstraction process was carried out component-wise.

We designed a set of eight LTS specifications on the basis of the SSL documentation. We verified these specifications on a system composed of one server (M_1) and one client (M_2) using both the brute-force composition ($M_1 \parallel M_2$), and our proposed automated AG approach. All experiments were carried out on a 1800+ XP AMD machine

[1] Spurious counterexamples arising due to abstraction are handled by iterative counterexample guided abstraction refinement.

Name		Direct		AG		Gain			
	Result	T_1	M_1	T_2	M_2	M_1/M_2	$\|A\|$	MQ	CQ
SSL-1	*Invalid*	*	2146	**325**	**207**	10.4	8	265	3
SSL-2	*Valid*	*	2080	**309**	**163**	12.8	8	279	3
SSL-3	*Valid*	*	2077	**309**	**163**	12.7	8	279	3
SSL-4	*Valid*	*	2076	**976**	**167**	12.4	16	770	4
SSL-5	*Valid*	*	2075	**969**	**167**	12.4	16	767	4
SSL-6	*Invalid*	*	2074	**3009**	**234**	8.9	24	1514	5
SSL-7	*Invalid*	*	2075	**3059**	**234**	8.9	24	1514	5
SSL-8	*Invalid*	*	2072	**3048**	**234**	8.9	24	1514	5

Fig. 4. Experimental results. Result = specification valid/invalid; T_1 and T_2 are times in seconds; M_1 and M_2 are memory in mega bytes; $|A|$ is the assumption size that sufficed to prove/disprove specification; MQ is the number of membership queries; CQ is the number of candidate queries. A * indicates out of memory (2 GB limit). Best figures are in bold.

with 3 GB of RAM running RedHat 9.0. Our results are summarized in Table 4. The learning based approach shows superior performance in all cases in terms of **memory** consumption (up to a **factor of 12.8**). An important reason behind such improvement is that the sizes of the (automatically learnt) assumptions that suffice to prove or disprove the specification (shown in column labeled $|A|$) are much smaller than the size of the second (client) component (3136 states).

6 Conclusion

We have presented an automated AG-style framework for checking simulation conformance between LTSs. Our approach uses a learning algorithm L^T to incrementally construct the weakest assumption that can discharge the premises of a non-circular AG proof rule. The learning algorithm requires a minimally adequate teacher that is implemented in our framework via a model checker. We have implemented this framework in the COMFORT [7] toolkit and experimented with a set of benchmarks based on the OPENSSL source code and the SSL specification. Our experiments indicate that in practice, extremely small assumptions often suffice to discharge the AG premises. This can lead to orders of magnitude improvement in the memory and time required for verification. Extending learning-based AG proof frameworks to other kinds of conformances, such as LTL model checking and deadlock detection, and to other AG-proof rules [3] remains an important direction for future investigation.

Acknowledgement. We thank the CAV 2005 referees for their invaluable comments and suggestions. The first author is also grateful to Corina Păsăreanu and Dimitra Giannakopoulou for informative discussions on assume-guarantee and learning.

References

1. R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL*, pages 98–109, 2005.

2. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
3. H. Barringer, D. Giannakopoulou, and C.S Pasareanu. Proof rules for automated compositional verification. In *Proc. of the 2nd Workshop on SAVCBS*, 2003.
4. M. Bernard and C. de la Higuera. Gift: Grammatical inference for terms. In *International Conference on Inductive Logic Programming*, 1999.
5. R. C. Carrasco, J. Oncina, and J. Calera-Rubio. Stochastic inference of regular tree languages. In *Proc. of ICGI*, pages 187–198. Springer-Verlag, 1998.
6. S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *FMSD*, 25(2–3), 2004.
7. S. Chaki, J. Ivers, N. Sharygina, and K. Wallnau. The ComFoRT Reasoning Framework. In *Proc. of CAV*, 2005. to appear.
8. E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *LICS*, 1989.
9. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV*, 2000.
10. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and System (TOPLAS)*, 16(5):1512–1542, 1994.
11. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
12. J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of TACAS '03*.
13. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*, chapter 1. 2002. available at http://www.grappa.univ-lille3.fr/tata.
14. F. Drewes and J. Hogberg. Learning a regular tree language. In *LNCS 2710, pp. 279–291, Proc. Developments in Language Theory (DLT) '03*.
15. M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proc. of ICSE*, 1999.
16. P. Garca and J. Oncina. Inference of recognizable tree sets. Technical Report II/47/1993, Dept. de Sistemas Informticos y Computacin, Universidad Politcnica de Valencia, 1993.
17. E. M. Gold. Language identification in the limit. *Information and Control*, 10(5), 1967.
18. E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, June 1978.
19. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 357–370, 2002.
20. P. Habermehl and T. Vojnar. Regular model checking using inference of regular languages. In *Proc. of INFINITY'04*, 2004.
21. P. Oncina, J.; Garca. Identifying regular languages in polynomial time. World Scientific Publishing, 1992. Advances in Structural and Syntactic Pattern Recognition,.
22. D. Peled, M.Y. Vardi, and M. Yannakakis. Black box checking. In *FORTE/PSTV*, 1999.
23. A. Pnueli. In transition from global to modular temporal reasoning about programs. *Logics and models of concurrent systems*, pages 123–144, 1985.
24. R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *Information and Computation*, volume 103(2), pages 299–347, 1993.
25. Y. Sakakibara. Learning context-free grammars from structural data in polynomial time. *Theoretical Computer Science (TCS)*, 76(2-3):223–242, 1990.