Formal Methods in System Design manuscript No.
(will be inserted by the editor)

Three Optimizations for Assume-Guarantee Reasoning with L*

Sagar Chaki[1], **Ofer Strichman**[2]

[1] Software Engineering Institute, Pittsburgh, USA [*]

[2] Information Systems Engineering, IE, Technion, Israel

Received: August 7, 2009/ Revised version: August 7, 2009

Abstract The learning-based automated Assume-Guarantee reasoning paradigm has been applied in the last few years for the compositional verification of concurrent systems. Specifically, L* has been used for learning the assumption, based on strings derived from counterexamples, which are given to it by a model-checker that attempts to verify the Assume-Guarantee rules. We suggest three optimizations to this paradigm. First, we derive from each counterexample multiple strings to L*, rather than a single one as in previous approaches. This small improvement saves candidate queries and hence model-checking runs. Second, we observe that in existing instances of this paradigm, the learning algorithm is coupled weakly with the teacher. Thus, the learner completely ignores the details of the internal structure of the system and specification being verified, which are available already to the teacher. We suggest an optimization that uses this information in order to avoid many unnecessary – and expensive, since they involve model checking – membership and candidate queries. Finally, and most importantly, we develop a method for minimizing the alphabet used by the assumption, which reduces the size of the assumption and the number of queries required to construct it. We present these three optimizations in the context of verifying trace containment for concurrent systems composed of finite state machines. We have implemented our approach in the ComFoRT tool, and

[*] This research was supported by the Predictable Assembly from Certifiable Components (PACC) initiative at the Software Engineering Institute, Pittsburgh.

experimented with real-life examples. Our results exhibit an average speedup of between 4 to 12 times, depending on the assume-guarantee rule used and the combination of optimizations options.

1 Introduction

Formal reasoning about concurrent programs is particularly hard due to the number of reachable states in the overall system. In particular, the number of such states can grow exponentially with each added component. Assume-Guarantee (AG) is a method for compositional reasoning that can be helpful in such cases. Consider a system with two components M_1 and M_2 that need to synchronize on a given set of shared actions, and a property φ that the system should be verified against. In its simplest form, AG requires checking one of the components, say M_1, separately, while making some assumption on the behaviors permitted by M_2. The assumption should then be discharged when checking M_2 in order to conclude the conformance of the product machine with the property. This idea is formalized with the following non-circular AG rule:

$$\frac{\begin{array}{r} A \times M_1 \preceq \varphi \\ M_2 \preceq A \end{array}}{M_1 \times M_2 \preceq \varphi} \quad \text{(AG-NC)} \tag{1}$$

where $\preceq$ stands for some conformance relation[1]. For trace containment, simulation and some other known relations, AG-NC is a sound and complete rule. In this paper, we consider the case in which M_1, M_2 and φ are non-deterministic finite automata, and interpret $\preceq$ as the trace containment (i.e., language inclusion) relation.

Recently, Cobleigh et al. proposed [12] a completely automatic method for finding the assumption A, using Angluin's L* algorithm [4]. L* constructs a minimal Deterministic Finite Automaton (DFA) that accepts an unknown regular language U. L* interacts iteratively with a Minimally Adequate Teacher (MAT). In each iteration, L* queries the MAT about *membership* of strings in U and whether the language of a specific *candidate* DFA is equal to U. The MAT is expected to supply a "Yes/No" answer to both types of questions.

[1] Clearly, for this rule to be effective, $A \times M_1$ must be easier to compute than $M_1 \times M_2$.

It is also expected to provide a counterexample along with a negative answer to a question of the latter type. L* then uses the counterexample to refine its candidate DFA while enlarging it by at least one state. L* is guaranteed to terminate within no more than n iterations, where n is the size of the minimal DFA accepting U.

In this paper we suggest three improvements to the automated AG procedure. The first improvement is based on the observation that counterexamples can sometimes be reused in the refinement process, which saves candidate queries.

The second improvement is based on the observation that the core L* algorithm is completely unaware of the internal details of M_1, M_2 and φ. With a simple analysis of these automata, most queries to the MAT can in fact be avoided. Indeed, we suggest to allow the core L* procedure access to the internal structure of M_1, M_2 and φ. This leads to a tighter coupling between the L* procedure and the MAT, and enables L* to make queries to the MAT in a more intelligent manner. Since each MAT query incurs an expensive model checking run, overall performance is improved considerably.

The last and most important improvement is based on the observation that the alphabet of the assumption A is fixed conservatively to be the entire interface alphabet between M_1 and φ on one hand, and M_2 on the other. While the full interface alphabet is always sufficient, it is often possible to complete the verification successfully with a much smaller assumption alphabet. Since the overall complexity of the procedure depends on the alphabet size, a smaller alphabet can improve the overall performance. In other words, while L* guarantees the minimality of the learned assumption DFA with respect to a given alphabet, our improvement reduces the size of the alphabet itself, and hence also the size of the learned DFA. The technique we present is based on an automated abstraction/refinement procedure: we start with the empty alphabet and keep refining it based on an analysis of the counterexamples, using a pseudo-Boolean solver. The procedure is guaranteed to terminate with a minimal assumption alphabet that suffices to complete the overall verification. This technique effectively combines the two paradigms of automated AG reasoning and abstraction-refinement.

Although our optimizations are presented in the context of AG-NC, a non-circular AG rule, they are applicable for circular AG rules as well. Specifically, our tool supports the AG-C rule, which appears in Eq. 2. The experimental results that we present in Section 6 include experiments with this rule as well.

$$\frac{\begin{array}{c} M_1 \times A_1 \preceq \varphi \\ M_2 \times A_2 \preceq \varphi \\ \overline{A_1} \times \overline{A_2} \preceq \varphi \end{array}}{M_1 \times M_2 \preceq \varphi} \quad \text{(AG-C)} \tag{2}$$

We implemented our approach in the ComFoRT [8] reasoning framework and experimented with a set of benchmarks derived from real-life source code. The improvements reduce the overall number of queries to the MAT and the size of the learned automaton. While individual speedup factors exceeded 23, an average speedup of a factor of over 12 was observed (for the circular rule). The speedup achieved for the non-circular rule are more moderate and average on 4.5. Somewhat surprisingly, the relative effect of each of the optimizations change when switching from circular to non-circular rules. The detailed results and a discussion of this point appear in Section 6.

Related Work. The L* algorithm was developed originally by Angluin [4]. Most learning-based AG implementations, including ours, use a more sophisticated version of L* proposed by Rivest and Schapire [22]. Machine learning techniques have been used in several contexts related to verification [20,16,2,17,13]. The use of L* for AG reasoning was first proposed by Cobleigh et al. [12]. A symbolic version of this framework has also been developed by Alur et al. [3]. The use of learning for automated AG reasoning has also been investigated in the context of simulation checking [7] and deadlock detection [9]. The circular rule AG-C described in Section 1 was proposed by Barringer et. al [6]. The basic idea behind the automated AG reasoning paradigm is to learn an assumption [15], using L*, that satisfies the two premises of AG-NC. The AG paradigm was proposed originally by Pnueli [21] and has since been explored (in manual/semi-automated forms) widely. The third optimization we propose amounts to a form of counterexample-guided abstraction refinement (CEGAR). The core ideas behind CEGAR were proposed originally by Kurshan [19], and CE-

GAR has since been used successfully for automated hardware [11] and software [5] verification. An approach similar to our third optimization was proposed independently by Gheorghiu et. al [14]. However, they use polynomial (greedy) heuristics aimed at minimizing the alphabet size, whereas we find the optimal value, and hence we solve an NP-hard problem.

The main difference of this article from its earlier proceedings version [10] is the inclusion of the circular rule AG-C (Eq. 2) and the experiments with it, as well as a slower introduction to L* and the prior work by the NASA group.

2 Preliminaries

Let λ and $\cdot$ denote the empty string and the concatenation operator respectively. We use lower letters (α, β, etc.) to denote actions, and higher letters (σ, π, etc.) to denote strings.

Definition 1 (Finite Automaton) *A finite automaton (FA) is a 5-tuple $(S, Init, \Sigma, T, F)$ 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 of actions, (iv) $T \subseteq S \times \Sigma \times S$ is the transition relation, and (v) $F \subseteq S$ is a set of accepting states.*

For any FA $M = (S, Init, \Sigma, T, F)$, we write $s \xrightarrow{\alpha} s'$ to mean $(s, \alpha, s') \in T$. Then the function δ is defined as follows: $\forall \alpha \in \Sigma \centerdot \forall s \in S \centerdot \delta(\alpha, s) = \{s' | s \xrightarrow{\alpha} s'\}$. We extend δ to operate on strings and sets of states in the natural manner. Thus, for any $\sigma \in \Sigma^*$ and $S' \subseteq S$, $\delta(\sigma, S')$ denotes the set of states of M reached by simulating σ on M starting from any $s \in S'$. The language accepted by M, denoted $\mathcal{L}(M)$, is defined as follows: $\mathcal{L}(M) = \{\sigma \in \Sigma^* \mid \delta(\sigma, Init) \cap F \neq \emptyset\}$.

Determinism. An FA $M = (S, Init, \Sigma, T, F)$ is said to be a deterministic FA, or DFA, if $|Init| = 1$ and $\forall \alpha \in \Sigma \centerdot \forall s \in S \centerdot |\delta(\alpha, s)| \leq 1$. Also, M is said to be complete if $\forall \alpha \in \Sigma \centerdot \forall s \in S \centerdot |\delta(\alpha, s)| \geq 1$. Thus, for a complete DFA, we have the following: $\forall \alpha \in \Sigma \centerdot \forall s \in S \centerdot |\delta(\alpha, s)| = 1$. Unless otherwise mentioned, all DFA we consider in the rest of this paper are also complete. It is well-known that a language is regular if and only if it is accepted by some FA (or DFA, since FA and DFA have the same accepting power). Also, every regular language is accepted by a unique (up to isomorphism) minimal DFA.

Complementation. For any regular language L, over the alphabet Σ, we write $\overline{L}$ to mean the language $\Sigma^* - L$. If L is regular, then so is $\overline{L}$. For any FA M we write $\overline{M}$ to mean the (unique) minimal DFA accepting $\overline{\mathcal{L}(M)}$.

Projection. The projection of any string σ over an alphabet Σ is denoted by $\sigma\lfloor_\Sigma$ and defined inductively on the structure of σ as follows: (i) $\lambda\lfloor_\Sigma = \lambda$, and (ii) $(\alpha \cdot \sigma')\lfloor_\Sigma = \alpha \cdot (\sigma'\lfloor_\Sigma)$ if $\alpha \in \Sigma$ and $\sigma'\lfloor_\Sigma$ otherwise. The projection of any regular language L on an alphabet Σ is defined as: $L\lfloor_\Sigma = \{\sigma\lfloor_\Sigma \mid \sigma \in L\}$. If L is regular, so is $L\lfloor_\Sigma$. Finally, the projection $M\lfloor_\Sigma$ of any FA M on an alphabet Σ is the (unique) minimal DFA accepting the language $\mathcal{L}(M)\lfloor_\Sigma$.

For the purpose of modeling systems with components that need to synchronize, it is convenient to distinguish between *local* and *global* actions. Specifically, local actions belong to the alphabet of a single component, while global actions are shared between multiple components. As defined formally below, components synchronize on global actions, and execute asynchronously on local actions.

Definition 2 (Parallel Composition) *Given two finite automata $M_1 = (S_1, Init_1, \Sigma_1, T_1, F_1)$ and $M_2 = (S_2, Init_2, \Sigma_2, T_2, F_2)$, their parallel composition $M_1 \times M_2$ is the FA $(S_1 \times S_2, Init_1 \times Init_2, \Sigma_1 \cup \Sigma_2, T, F_1 \times F_2)$ such that $\forall s_1, s_1' \in S_1 .\ \forall s_2, s_2' \in S_2,\ (s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')$ if and only if for $i \in \{1, 2\}$ either $\alpha \notin \Sigma_i \wedge s_i = s_i'$ or $s_i \xrightarrow{\alpha} s_i'$.*

Trace Containment. For any FA M_1 and M_2, we write $M_1 \preceq M_2$ to mean $\mathcal{L}(M_1 \times \overline{M_2}) = \emptyset$. A counterexample to $M_1 \preceq M_2$ is a string $\sigma \in \mathcal{L}(M_1 \times \overline{M_2})$.

3 The L* Algorithm

The L* algorithm for learning DFAs was developed by Angluin [4] and later improved by Rivest and Schapire [22]. In essence, L* learns an unknown regular language U, over an alphabet Σ, by generating the minimal DFA that accepts U. In order to learn U, L* requires "Yes/No" answers to two types of queries:

1. *Membership query*: for a string $\sigma \in \Sigma^*$, 'is $\sigma \in U$?'

2. *Candidate query*: for a DFA C, 'is $\mathcal{L}(C) = U$?'

If the answer to a candidate query is "No", L* expects a counterexample string σ such that $\sigma \in U - \mathcal{L}(C)$ or $\sigma \in \mathcal{L}(C) - U$. In the first case, we call σ a *positive* counterexample, because it should be added to $\mathcal{L}(C)$. In the second case, we call σ a *negative counterexample* since it should be removed from $\mathcal{L}(C)$. As mentioned before, L* uses the MAT to obtain answers to these queries.

Observation Table. L* builds an observation table (S, E, T) where: (i) $S \subseteq \Sigma^*$ is the set of rows, (ii) $E \subseteq \Sigma^*$ is the set of columns (or experiments), and (iii) $T : (S \cup S \cdot \Sigma) \times E \to \{0, 1\}$ is a function defined as follows:

$$\forall s \in (S \cup S \cdot \Sigma) \centerdot\ \forall e \in E \centerdot\ T(s, e) = \begin{cases} 1 & s \cdot e \in U \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Consistency and Closure. For any $s_1, s_2 \in (S \cup S \cdot \Sigma)$, s_1 and s_2 are equivalent (denoted as $s_1 \equiv s_2$) if $\forall e \in E \centerdot\ T(s_1, e) = T(s_2, e)$. A table is *consistent* if $\forall s_1, s_2 \in S \centerdot\ s_1 \neq s_2 \Rightarrow s_1 \not\equiv s_2$. L* always maintains a consistent table. In addition, a table is *closed* if $\forall s \in S \centerdot\ \forall \alpha \in \Sigma \centerdot\ \exists s' \in S \centerdot\ s' \equiv s \cdot \alpha$.

Candidate Construction. Given a closed and consistent table (S, E, T), L* constructs a candidate DFA $C = (S, \{\lambda\}, \Sigma, \Delta, F)$ such that: (i) $F = \{s \in S \mid T(s, \lambda) = 1\}$, and (ii) $\Delta = \{(s, \alpha, s') \mid s' \equiv s \cdot \alpha\}$. Note that C is deterministic and complete since (S, E, T) is consistent and closed. Since a row corresponds to a state of C, we use the terms "row" and "candidate state" synonymously.

Example 1 Consider Figure 1. On the left is an observation table with the entries being the T values. Let $\Sigma = \{\alpha, \beta\}$. From this table we see that $\{e_2, \alpha, \alpha \cdot e_2, \beta \cdot e_2, \alpha\alpha, \ldots\} \in U$. On the right is the corresponding candidate DFA. $\square$

L* Step-By-Step. We now describe L* in more detail, using line numbers from its algorithmic description in Figure 2. This conventional version of L* is used currently in the context of automated AG reasoning. We also point out the specific issues that are addressed by the improvements we propose later on in this paper.

		E		
		λ	e_2	e_3
S	λ	0	1	0
	α	1	1	0
$S \cdot \Sigma$	β	0	1	0
	$\alpha\alpha$	1	1	0
	$\alpha\beta$	0	1	0

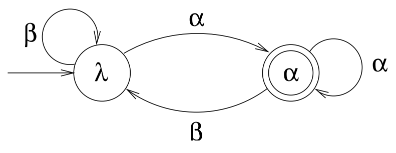

Fig. 1 An Observation Table and the Corresponding Candidate DFA

Recall that λ denotes the empty string. After the initialization at Line 1, the table has one cell corresponding to (λ, λ). In the top-level loop, the table entries are first computed (at Line 2) using membership queries.

Next, L* closes the table by trying to find (at Line 3) for each $s \in S$, some *uncovered* action $\alpha \in \Sigma$ such that $\forall s' \in S \centerdot\ s' \not\equiv s \cdot \alpha$. If such an uncovered action α is found for some $s \in S$, L* adds $s \cdot \alpha$ to S at Line 4 and continues with the closure process. Otherwise, it proceeds to the next Step. Note that each $\alpha \in \Sigma$ is considered when attempting to find an uncovered action.

Once the table is closed, L* constructs (at Line 5) a candidate DFA C using the procedure described previously. Next, at Line 6, L* conjectures that $\mathcal{L}(C) = U$ via a candidate query. If the conjecture is wrong L* extracts from the counterexample CE (returned by the MAT) a suffix e that, when added to E, causes the table to cease being closed. The process of extracting the feedback e has been presented elsewhere [22] and we do not describe it here. Once e has been obtained, L* adds e to E and iterates the top-level loop by returning to line 2. Note that since the table is no longer closed, the subsequent process of closing it strictly increases the size of S. It can also be shown that the size of S cannot exceed n, where n is number of states of the minimal DFA accepting U. Therefore, the top-level loop of L* executes no more than n times.

Non-uniform Refinement. It is interesting to note that the feedback from CE does not refine the candidate in the abstraction/refinement sense: refinement here does not necessarily add/eliminate a positive/negative CE; this occurs eventually, but not necessarily in one step. Indeed, the first improvement we propose leverages

```
(1)  let $S = E = \{\lambda\}$
     loop {
(2)    Update $T$ using queries
       while $(S, E, T)$ is not closed {
(3)      Find $(s, \alpha) \in S \times \Sigma$ such that $\forall s' \in S.\ s' \not\equiv s \cdot \alpha$
(4)      Add $s \cdot \alpha$ to $S$
       }
(5)    Construct candidate DFA $C$ from $(S, E, T)$
(6)    Make the conjecture $C$
(7)    if $C$ is correct return $C$
(8)    else Add $e \in \Sigma^*$ that witnesses the counterexample to E
     }
```

Fig. 2 The L* algorithm for learning an unknown regular language.

this observation to reduce the number of candidate queries. It is also interesting to note that the refinement does not work in one direction: it may remove strings that are in U or add strings that are not in U. The only guarantee that we have is that in each step at least one state is added to the candidate and that eventually L* learns U itself.

L* is based on the observation that in a DFA, two reachable states s, s' are distinct (and hence cannot be combined) if and only if there exists a *distinguishing sequence* $\sigma \in \Sigma^*$ such that $\delta(\sigma, s) \in F \iff \delta(\sigma, s') \notin F$. Indeed, from the counterexample that it receives from a candidate query, L* extracts a string σ that demonstrates that there exists two rows p_1, p_2 such that $\delta(p_1, q_0) = \delta(p_2, q_0) = s$ (s being a state of the current candidate automaton C), and $p_1 \cdot \sigma \in U$ whereas $p_2 \cdot \sigma \notin U$; hence, s must be split into two states. This means that at each iteration the number of states must increase by at least one, and that none of these states is redundant. This property guarantees that L* makes no more than $n - 1$ wrong conjectures, and terminates with the minimal DFA $M(U)$.

Complexity. Overall, the number of membership queries made by L* is $\mathcal{O}(kn^2 + n\log m)$, where $k = |\Sigma|$ is the size of the alphabet of U, and m is the length of the longest counterexample to a candidate query returned by the MAT [22]. The dominating fragment of this complexity is kn^2 which varies directly with the size of Σ. As noted before, the Σ used in the literature is sufficient, but often unnecessarily large. The third improvement we propose is aimed at reducing the number of membership queries by minimizing the size of Σ.

4 AG Reasoning with L*

In this section, we describe the key ideas behind the automated AG procedure proposed by Cobleigh et al. [12]. We begin with a fact that we use later on.

Fact 1 *For any FA M_1 and M_2 with alphabets Σ_1 and Σ_2, $\mathcal{L}(M_1 \times M_2) \neq \emptyset$ if and only if $\exists \sigma \in \mathcal{L}(M_1) \centerdot \sigma\downharpoonright_{(\Sigma_1 \cap \Sigma_2)} \in \mathcal{L}(M_2)\downharpoonright_{(\Sigma_1 \cap \Sigma_2)}$.*

Let us now restate AG-NC to reflect our implementation more accurately:

$$\frac{\begin{array}{r} A \times (M_1 \times \bar{\varphi}) \preceq \bot \\ M_2 \preceq A \end{array}}{(M_1 \times \bar{\varphi}) \times M_2 \preceq \bot} \tag{4}$$

where $\bot$ denotes a DFA accepting the empty language. The unknown language to be learned is

$$U = \mathcal{L}(\overline{(M_1 \times \bar{\varphi})\downharpoonright_{\Sigma}}) \tag{5}$$

over the alphabet $\Sigma = (\Sigma_1 \cup \Sigma_\varphi) \cap \Sigma_2$ where Σ_1, Σ_2 and Σ_φ are the alphabets of M_1, M_2 and φ respectively[2]. The choice of U and Σ is significant because, by Fact 1, the consequence of Eq. 4 does not hold if and only if the intersection between $\overline{U} = \mathcal{L}((M_1 \times \bar{\varphi})\downharpoonright_{\Sigma})$ and $\mathcal{L}(M_2\downharpoonright_{\Sigma})$ is non-empty. This situation is depicted in Fig. 3(a). Hence, if A is the DFA computed by L* such that $\mathcal{L}(A) = U$, any counterexample to the second

[2] Note that we do not compute U directly because complementing M_1, a non-deterministic automaton, is typically intractable.

premise $M_2 \preceq A$ is guaranteed to be a real one. However, in practice, the process terminates after learning U itself only in the worst case. As we shall see, it usually terminates earlier by finding either a counterexample to $M_1 \times M_2 \preceq \varphi$, or an assumption A that satisfies the two premises of Eq. 4. This later case is depicted in Fig. 3(b).

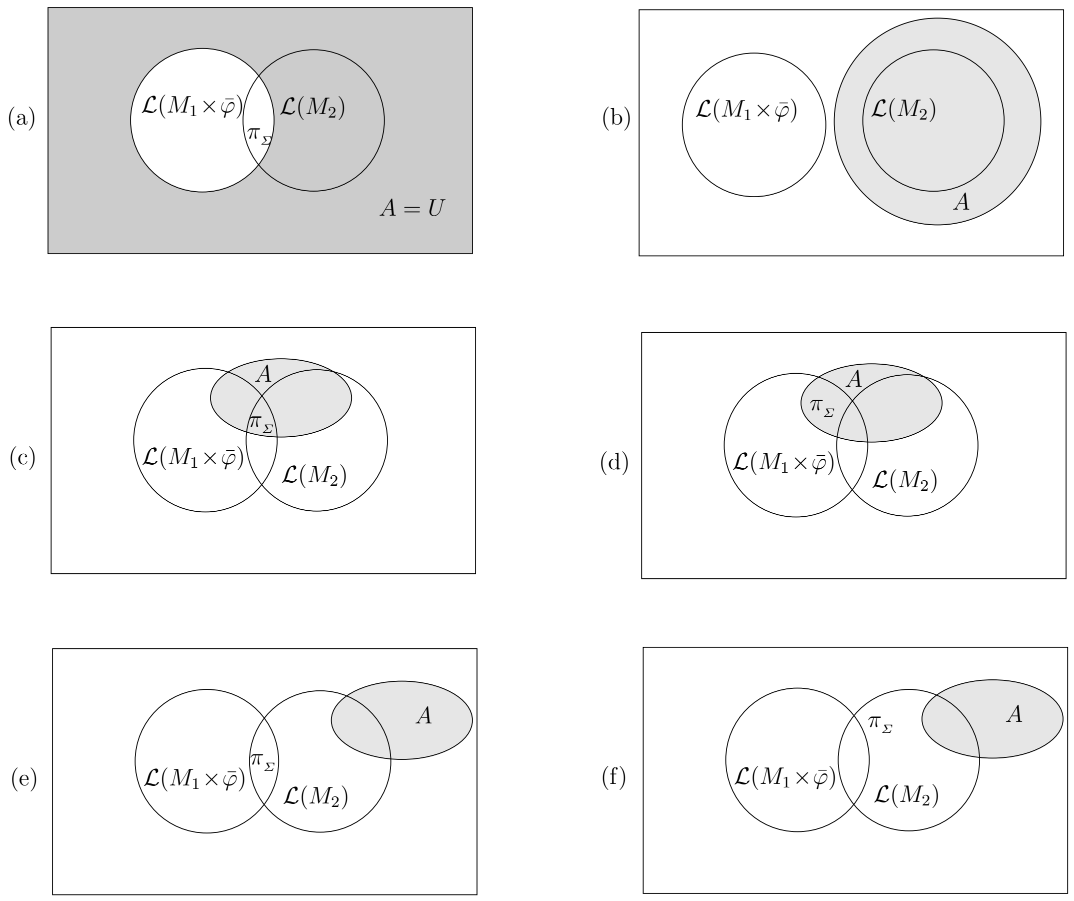
(a)
$\mathcal{L}(M_1 \times \bar{\varphi})$
$\mathcal{L}(M_2)$
π_Σ
$A = U$
(b)
$\mathcal{L}(M_1 \times \bar{\varphi})$
$\mathcal{L}(M_2)$
A
(c)
A
π_Σ
$\mathcal{L}(M_1 \times \bar{\varphi})$
$\mathcal{L}(M_2)$
(d)
A
π_Σ
$\mathcal{L}(M_1 \times \bar{\varphi})$
$\mathcal{L}(M_2)$
(e)
A
π_Σ
$\mathcal{L}(M_1 \times \bar{\varphi})$
$\mathcal{L}(M_2)$
(f)
π_Σ
A
$\mathcal{L}(M_1 \times \bar{\varphi})$
$\mathcal{L}(M_2)$

Fig. 3 Different L* scenarios. The gray area represents the candidate assumption A.

MAT Implementation. The answer to a membership query with a string σ is "Yes" if and only if σ cannot be simulated on $M_1 \times \bar{\varphi}$ (see Eq. 5). A candidate query with some candidate A, on the other hand, is more complicated, and is now described step-wise (see also Figure 4). From hereon we denote $\pi\downarrow_\Sigma$ by π_Σ.

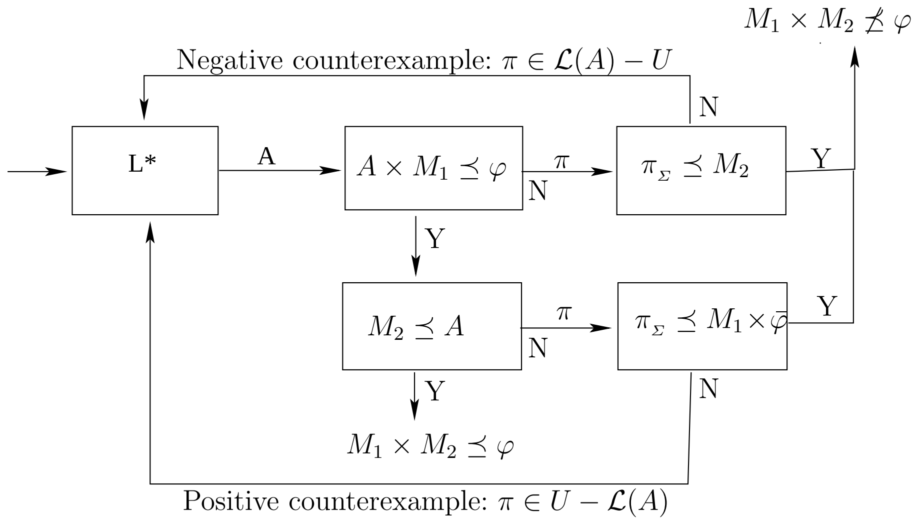
$M_1 \times M_2 \not\preceq \varphi$
Negative counterexample: $\pi \in \mathcal{L}(A) - U$
N
L*
A
$A \times M_1 \preceq \varphi$
π
N
$\pi_\Sigma \preceq M_2$
Y
Y
$M_2 \preceq A$
π
N
$\pi_\Sigma \preceq M_1 \times \bar{\varphi}$
Y
Y
N
$M_1 \times M_2 \preceq \varphi$
Positive counterexample: $\pi \in U - \mathcal{L}(A)$

Fig. 4 AG Reasoning with L*.

Step 1. Use model checking to verify that A satisfies the first premise of Eq. 4. If the verification of the first premise fails, obtain a counterexample trace $\pi \in \mathcal{L}(A \times M_1 \times \bar{\varphi})$ and proceed to Step 2. Otherwise, go to Step 3.

Step 2. Check via simulation if $\pi_\Sigma \in \mathcal{L}(M_2 \downharpoonleft_\Sigma)$. If so, then by Fact 1, $\mathcal{L}(M_1 \times \bar{\varphi} \times M_2) \neq \emptyset$ (i.e., $M_1 \times M_2 \not\preceq \varphi$) and the algorithm terminates. This situation is depicted in Fig. 3(c). Otherwise $\pi_\Sigma \in \mathcal{L}(A) - U$ is a negative counterexample, as depicted in Fig. 3(d). Control is returned to L* with π_Σ.

Step 3. At this point A is known to satisfy the first premise. Proceed to model check the second premise. If $M_2 \preceq A$ holds as well, then by Eq. 4 conclude that $M_1 \times M_2 \preceq \varphi$ and terminate. This possibility was already shown in Fig. 3(b). Otherwise obtain a counterexample $\pi \in \mathcal{L}(M_2 \times \overline{A})$ and proceed to Step 4.

Step 4. Check if $\pi_\Sigma \in \mathcal{L}((M_1 \times \bar{\varphi}) \downharpoonleft_\Sigma)$. If so, then by Fact 1, $\mathcal{L}(M_1 \times \bar{\varphi} \times M_2) \neq \emptyset$ (i.e., $M_1 \times M_2 \not\preceq \varphi$) and the algorithm terminates. This scenario is depicted in Fig. 3(e). Otherwise $\pi_\Sigma \in U - \mathcal{L}(A)$ is a positive counterexample, as depicted in Fig. 3(f) – return to L* with π_Σ.

Note that Steps 2 and 4 above are duals obtained by interchanging $M_1 \times \bar{\varphi}$ with M_2 and U with $\mathcal{L}(A)$. Also, note that Fact 1 could be applied in Steps 2 and 4 above only because $\Sigma = (\Sigma_1 \cup \Sigma_\varphi) \cap \Sigma_2$. In the next section, we propose an improvement that allows Σ to be varied. Consequently, we also modify the procedure for answering candidate queries so that Fact 1 is used only in a valid manner.

5 Optimized L*-based AG Reasoning

In this section we list three improvements to the algorithm described in Section 4. The first two improvements reduce the number of candidate and membership queries respectively. The third improvement is aimed at completing the verification process using an assumption alphabet that is smaller than $(\Sigma_1 \cup \Sigma_\varphi) \cap \Sigma_2$.

5.1 Reusing Counterexamples

Recall from Section 3 that every candidate query counterexample π returned to L* is used to find a suffix that makes the table not closed, and hence adds at least one state (row) to the current candidate C (observation table). Let C' denote the new candidate constructed in the next iteration of the top-level loop (see Figure 2). We say that C' is obtained by refining C on π. However, the refinement process does not guarantee the addition/elimination of a positive/negative counterexample from C'. Thus, a negative counterexample $\pi \in L(C) - U$ may still be accepted by C', and a positive counterexample $\pi \in U - L(C)$ may still be rejected by C'. This leads naturally to the idea of reusing counterexamples. Specifically, for every candidate C' obtained by refining on a negative counterexample π, we check, via simulation, whether $\pi \in \mathcal{L}(C')$. If this is the case, we repeat the refinement process on C' using π instead of performing a candidate query with C'. The same idea is applied to positive counterexamples as well. Thus, if we find that $\pi \notin \mathcal{L}(C')$ for a positive counterexample π, then π is used to further refine C'. This optimization reduces the number of candidate queries.

5.2 Selective Membership Queries

Recall the operation of closing the table (see Lines 3 and 4 of Figure 2) in L*. For every row s added to S, L* must compute T for every possible extension of s by a single action. Thus L* must decide if $s \cdot \alpha \cdot e \in U$

for each $\alpha \in \Sigma$ and $e \in E$ — a total of $|\Sigma| \cdot |E|$ membership queries. To see how a membership query is answered, for any $\sigma \in \Sigma^*$, let $Sim(\sigma)$ be the set of states of $M_1 \times \bar{\varphi}$ reached by simulating σ from an initial state of $M_1 \times \bar{\varphi}$ and by treating actions not in Σ as ϵ (i.e., ϵ-transitions are allowed where the actions are local to $M_1 \times \bar{\varphi}$). Then, $\sigma \in U$ if and only if $Sim(\sigma)$ does not contain an accepting state of $M_1 \times \bar{\varphi}$.

Let us return to the problem of deciding if $s \cdot \alpha \cdot e \in U$. Let $En(s) = \{\alpha' \in \Sigma \mid \delta(\alpha', Sim(s)) \neq \emptyset\}$ be the set of enabled actions from $Sim(s)$ in $M_1 \times \bar{\varphi}$. Now, for any $\alpha \notin En(s)$, $Sim(s \cdot \alpha \cdot e) = \emptyset$ and hence $s \cdot \alpha \cdot e$ is guaranteed to belong to U. This observation leads to our second improvement. Specifically, for every s added to S, we first compute $En(s)$. Note that $En(s)$ is computed by simulating $s\downharpoonright_{\Sigma_1}$ on M_1 and $s\downharpoonright_{\Sigma_\varphi}$ on φ separately, without composing M_1 and φ. We then make membership queries with $s \cdot \alpha \cdot e$, but only for $\alpha \in En(s)$. For all $\alpha \notin En(s)$ we directly set $T(s \cdot \alpha, e) = 1$ since we know that in this case $s \cdot \alpha \cdot e \in U$. The motivation behind this optimization is that $En(s)$ is usually much smaller that Σ for any s. The actual improvement in performance due to this tactic depends on the relative sizes of $En(s)$ and Σ for the different $s \in S$.

5.3 Minimizing the Assumption Alphabet

As mentioned before, existing automated AG procedures use a constant assumption alphabet $\Sigma = (\Sigma_1 \cup \Sigma_\varphi) \cap \Sigma_2$. There may exist, however, an assumption A over a smaller alphabet $\Sigma_c \subset \Sigma$ that satisfies the two premises of Eq. 4. Since Eq. 4 is sound, the existence of such an A would still imply that $M_1 \times M_2 \preceq \varphi$. However, recall that the number of L* membership queries varies directly with the alphabet size. Therefore, the benefit, in the context of learning A, is that a smaller alphabet leads to fewer membership queries.

In this section, we propose an abstraction-refinement scheme for building an assumption over a minimal alphabet. During our experiments, this improvement led to a 6 times reduction in the size of the assumption alphabet. The main problem with changing Σ is of course that AG-NC is no longer complete. Specifically, if $\Sigma_C \subset \Sigma$, then there might not exist any assumption A over Σ_C that satisfies the two premises of AG-NC even though the conclusion of AG-NC holds. The following theorem characterizes this phenomenon precisely.

Theorem 1 (Incompleteness of AG-NC) *Suppose there exists a string π and an alphabet Σ_C that maintains the following condition:*

$$(\textsc{inc})\ \pi{\downharpoonright_{\Sigma_C}} \in \mathcal{L}((M_1 \times \bar{\varphi}){\downharpoonright_{\Sigma_C}}) \cap \mathcal{L}(M_2{\downharpoonright_{\Sigma_C}}). \tag{6}$$

Then no assumption A over Σ_C satisfies the two premises of AG-NC.

Proof Suppose that there exists a π satisfying INC and an A over Σ_C satisfying the two premises of AG-NC. This leads to a contradiction as follows:

- *Case 1*: $\pi{\downharpoonright_{\Sigma_C}} \in \mathcal{L}(A)$. Since A satisfies the first premise of AG-NC, we have $\pi{\downharpoonright_{\Sigma_C}} \notin \mathcal{L}((M_1 \times \bar{\varphi}){\downharpoonright_{\Sigma_C}})$, a contradiction with INC.
- *Case 2*: $\pi{\downharpoonright_{\Sigma_C}} \notin \mathcal{L}(A)$. Hence $\pi{\downharpoonright_{\Sigma_C}} \in \mathcal{L}(\overline{A})$. Since A satisfies the second premise of AG-NC, we have $\pi{\downharpoonright_{\Sigma_C}} \notin \mathcal{L}(M_2{\downharpoonright_{\Sigma_C}})$, again contradicting INC. □

We say that an alphabet Σ_C is incomplete if $\Sigma_C \neq \Sigma$ and there exists a trace π satisfying condition INC above. Therefore, whenever we come across a trace π that satisfies INC, unless $\Sigma_C = \Sigma$, we know that the current Σ_C is incomplete and must be refined. We now describe our overall procedure which incorporates testing Σ_C for incompleteness and refining an incomplete Σ_C appropriately.

Detecting Incompleteness. Our optimized automated AG procedure is depicted in Fig. 5. Initially $\Sigma_c = \emptyset$[3]. Let us write π_c and π_Σ to mean $\pi{\downharpoonright_{\Sigma_C}}$ and $\pi{\downharpoonright_{\Sigma}}$ respectively. The process continues as in Section 4, until one of the following two scenarios occur while answering a candidate query:

- *Scenario 1*: We reach Step 2 with a trace $\pi \in \mathcal{L}(A \times M_1 \times \bar{\varphi})$. Note that this implies $\pi_c \in \mathcal{L}((M_1 \times \bar{\varphi}){\downharpoonright_{\Sigma_C}})$. Now we first check if $\pi_c \in \mathcal{L}(M_2{\downharpoonright_{\Sigma_C}})$. If not, we return π_c as a negative counterexample to L* exactly as in Section 4. However, if $\pi_c \in \mathcal{L}(M_2{\downharpoonright_{\Sigma_C}})$, then π satisfies the condition INC of Theorem 1, and hence Σ_C is incomplete. Instead of refining Σ_C at this point, we first check if $\pi_\Sigma \in \mathcal{L}(M_2{\downharpoonright_{\Sigma}})$. If so, then as in Section 4, by a valid application of Fact 1, $M_1 \times M_2 \not\preceq \varphi$ and the algorithm terminates. Otherwise, if $\pi_\Sigma \notin \mathcal{L}(M_2{\downharpoonright_{\Sigma}})$, we refine Σ_C.

[3] We could also start with $\Sigma_c = \Sigma_\varphi$ since it is very unlikely that φ can be proven or disproven without controlling the actions that define it.

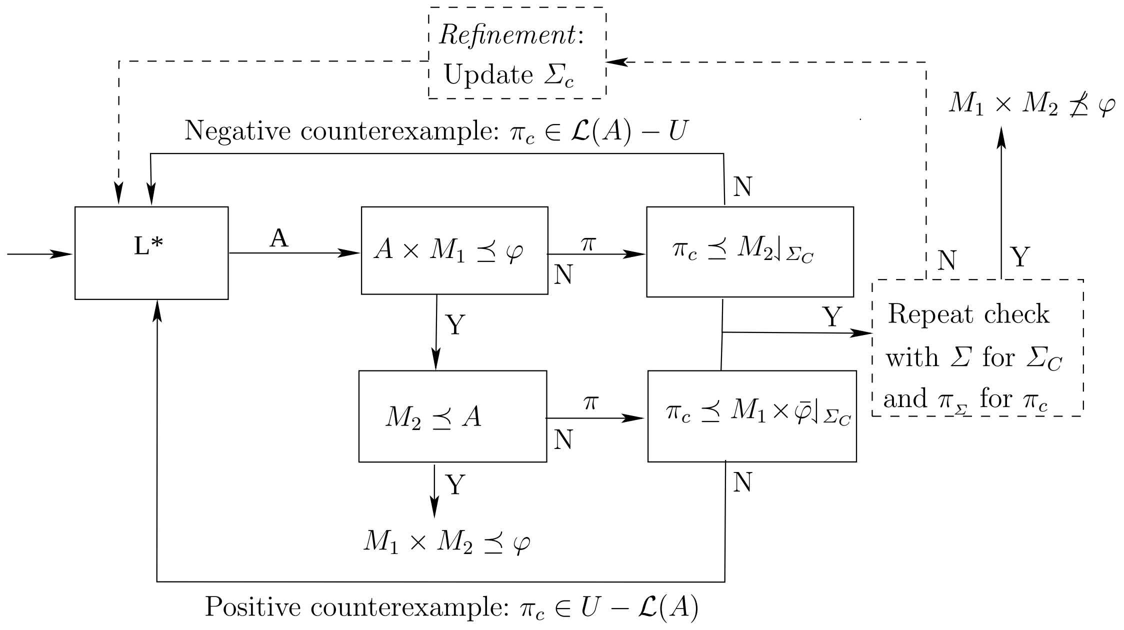
Refinement:
Update Σ_c
$M_1 \times M_2 \not\preceq \varphi$
Negative counterexample: $\pi_c \in \mathcal{L}(A) - U$
N
L*
A
$A \times M_1 \preceq \varphi$
π
N
$\pi_c \preceq M_2\rfloor_{\Sigma_C}$
N
Y
Y
Repeat check with Σ for Σ_C and π_Σ for π_c
Y
$M_2 \preceq A$
π
N
$\pi_c \preceq M_1 \times \bar{\varphi}\rfloor_{\Sigma_C}$
Y
N
$M_1 \times M_2 \preceq \varphi$
Positive counterexample: $\pi_c \in U - \mathcal{L}(A)$

Fig. 5 Generalized AG with L*, with an abstraction-refinement loop (added with dashed lines) based on the assumption alphabet $\Sigma_c \subseteq \Sigma$. Strings π_c and π_Σ denote $\pi\rfloor_{\Sigma_C}$ and $\pi\rfloor_\Sigma$ respectively.

– *Scenario 2*: We reach Step 4 with $\pi \in \mathcal{L}(M_2 \times \overline{A})$. Note that this implies $\pi_c \in \mathcal{L}(M_2\rfloor_{\Sigma_C})$. We first check if $\pi_c \in \mathcal{L}((M_1 \times \bar{\varphi})\rfloor_{\Sigma_C})$. If not, we return π_c as a positive counterexample to L* exactly as in Section 4. However, if $\pi_c \in \mathcal{L}((M_1 \times \bar{\varphi})\rfloor_{\Sigma_C})$, then π satisfies INC, and hence by Theorem 1, Σ_C is incomplete. Instead of refining Σ_C at this point, we first check if $\pi_\Sigma \in \mathcal{L}((M_1 \times \bar{\varphi})\rfloor_\Sigma)$. If so, then as in Section 4, by a valid application of Fact 1, $M_1 \times M_2 \not\preceq \varphi$ and we terminate. Otherwise, if $\pi_\Sigma \notin \mathcal{L}((M_1 \times \bar{\varphi})\rfloor_\Sigma)$, we refine Σ_C.

Note that the checks involving π_Σ in the two scenarios above correspond to the concretization attempts in a standard CEGAR loop. Also, Scenarios 1 and 2 are duals (as in the case of Steps 2 and 4 in Section 4) obtained by interchanging $M_1 \times \bar{\varphi}$ with M_2 and U with $\mathcal{L}(A)$. In essence, while solving a candidate query, an incomplete Σ_C results in a trace (specifically, π above) that satisfies INC and leads neither to an actual counterexample of $M_1 \times M_2 \preceq \varphi$, nor to a counterexample to the candidate query being solved. In accordance with the CEGAR terminology, we refer to such traces as *spurious* counterexamples and use them collectively to refine Σ_C as described next. In the rest of this section, all counterexamples we mention are spurious unless otherwise specified.

Refining the Assumption Alphabet. A counterexample arising from Scenario 1 above is said to be negative. Otherwise, it arises from Scenario 2 and is said to be positive. Our description that follows unifies the treatment of these two types of counterexamples, with the help of a common notation for $M_1 \times \bar{\varphi}$ and M_2. Specifically, let

$$M^{(\pi)} = \begin{cases} M_1 \times \bar{\varphi} & \pi \text{ is positive} \\ M_2 & \pi \text{ is negative} \end{cases}$$

We say that an alphabet Σ' eliminates a counterexample π, and denote this with $\mathit{Elim}(\pi, \Sigma')$, if $\pi\downarrow_{\Sigma'} \notin \mathcal{L}(M^{(\pi)}\downarrow_{\Sigma'})$. Therefore, any counterexample π is eliminated if we choose Σ_C such that $\mathit{Elim}(\pi, \Sigma_C)$ holds since π no longer satisfies the condition INC. Our goal, however, is to find a minimal alphabet Σ_C with this property. It turns out that finding such an alphabet is computationally hard.

Theorem 2 *Finding a minimal eliminating alphabet is NP-hard in $|\Sigma|$.*

Proof The proof relies on a reduction from the minimal hitting set problem.

Minimal Hitting Set. A Minimal Hitting Set (MHS) problem is a pair (U, T) where U is a finite set and $T \subseteq 2^U$ is a finite set of subsets of U. A solution to (U, T) is a minimal $X \subseteq U$ such that $\forall T' \in T \centerdot\ X \cap T' \neq \emptyset$. It is well-known that MHS is NP-complete in $|U|$ (see [18] for a discussion on complexity and approximations to this problem).

Now we reduce MHS to finding a minimal eliminating alphabet. Let (U, T) be any MHS problem and let $<$ be a strict order imposed on the elements of U. Consider the following problem $\mathcal{P}$ of finding a minimal eliminating alphabet. First, let $\Sigma = U$. Next, for each $T' \in T$ we introduce a counterexample $\pi(T')$ obtained by arranging the elements of U according to $<$, repeating each element of T' twice and the remaining elements of U just once. For example suppose $U = \{a, b, c, d, e\}$ such that $a < b < c < d < e$. Then for $T' = \{b, d, e\}$ we introduce the counterexample $\pi(T') = a \cdot b \cdot b \cdot c \cdot d \cdot d \cdot e \cdot e$. Also, for each counterexample $\pi(T')$ introduced, let $M(\pi(T'))$ accept a single string obtained by arranging the elements of U according to $<$, repeating each element of U just once. Thus, for the example U above, $M(\pi(T'))$ accepts the single string $a \cdot b \cdot c \cdot d \cdot e$.

Let us first show the following result: for any $T' \in T$ and any $X \subseteq U$, $X \cap T' \neq \emptyset$ if and only if $\mathit{Elim}(\pi(T'), X)$. In other words, $X \cap T' \neq \emptyset$ if and only if $\pi(T')\downarrow_X \notin \mathcal{L}(M(\pi(T'))\downarrow_X)$. Indeed suppose that some $\alpha \in X \cap T'$. Then $\pi(T')\downharpoonright_X$ contains two consecutive occurrences of α and hence cannot be accepted by $M(\pi(T'))\downharpoonright_X$. By the converse implication, if $M(\pi(T'))\downharpoonright_X$ does not accept $\pi(T')\downharpoonright_X$, then $\pi(T')\downharpoonright_X$ must contain two consecutive occurrences of some action α. But then $\alpha \in X \cap T'$ and hence $X \cap T' \neq \emptyset$. The above result implies immediately that any solution to the MHS problem (U, T) is also a minimal eliminating alphabet for $\mathcal{P}$. Also, the reduction from (U, T) to $\mathcal{P}$ described above can be performed using logarithmic space in $|U| + |T|$. Finally, $|\Sigma| = |U|$, which completes our proof. □

As we just proved, finding the minimal eliminating alphabet is NP-hard in $|\Sigma|$. Yet, since $|\Sigma|$ is relatively small, this problem can still be feasible in practice (as our experiments have shown: see Section 6). We propose a solution based on a reduction to Pseudo-Boolean constraints. Pseudo-Boolean constraints have the same modeling power as 0-1 ILP, but solvers for this logic are typically based on adapting SAT engines for linear constraints over Boolean variables, and geared towards problems with relatively few linear constraints (and a linear objective function) and constraints in CNF.

Optimal Refinement. Let Π be the set of all (positive and negative) counterexamples seen so far. We wish to find a minimal Σ_C such that: $\forall \pi \in \Pi\textbf{.}\ \mathit{Elim}(\pi, \Sigma_C)$. To this end, we formulate and solve a Pseudo-Boolean constraint problem with an objective function stating that we seek a solution which minimizes the chosen set of actions. The set of constraints of the problem is $\Phi = \bigcup_{\pi \in \Pi} \Phi(\pi)$. In essence, if $M^{[\pi]}$ is the minimal DFA accepting $\{\pi\}$, then $\Phi(\pi)$ represents symbolically the states reachable in $M^{[\pi]} \times M^{(\pi)}$, taking into account all possible values of Σ_C. Henceforth, we continue to use square brackets when referring to elements of $M^{[\pi]}$, and regular parenthesis when referring to elements of $M^{(\pi)}$.

We now define $\Phi(\pi)$ formally. Let $M^{[\pi]} = (S^{[\pi]}, \mathit{Init}^{[\pi]}, \Sigma^{[\pi]}, T^{[\pi]}, F^{[\pi]})$ and $M^{(\pi)} = (S^{(\pi)}, \mathit{Init}^{(\pi)}, \Sigma^{(\pi)}, T^{(\pi)}, F^{(\pi)})$. Let $\delta^{[\pi]}$ and $\delta^{(\pi)}$ be the δ functions of $M^{[\pi]}$ and $M^{(\pi)}$ respectively. We define a state variable of the form (s, t) for each $s \in S^{[\pi]}$ and $t \in S^{(\pi)}$. Intuitively, the variable (s, t) indicates whether the product state (s, t) is reachable in $M^{[\pi]} \times M^{(\pi)}$. We also define a choice variable $s(\alpha)$ for each action $\alpha \in \Sigma$, indicating whether α is selected to be included in Σ_C. Now, $\Phi(\pi)$ consists of the following clauses:

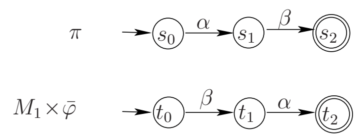

Fig. 6 A positive counterexample π and $M^{(\pi)} = M_1 \times \bar{\varphi}$.

Initialization and Acceptance: Every initial and no accepting state is reachable:

$$\forall s \in Init^{[\pi]} \centerdot \; \forall t \in Init^{(\pi)} \centerdot \; (s,t) \qquad\qquad \forall s \in F^{[\pi]} \centerdot \; \forall t \in F^{(\pi)} \centerdot \; \neg(s,t)$$

Shared Actions: Successors depend on whether an action is selected or not:

$$\forall \alpha \in \Sigma \centerdot \forall s \in S^{[\pi]} \centerdot \forall s' \in \delta^{[\pi]}(\alpha, s) \centerdot \forall t \in S^{(\pi)} \centerdot \forall t' \in \delta^{(\pi)}(\alpha, t) \centerdot (s,t) \Rightarrow (s', t')$$

$$\forall \alpha \in \Sigma \centerdot \forall s \in S^{[\pi]} \centerdot \forall s' \in \delta^{[\pi]}(\alpha, s) \centerdot \forall t \in S^{(\pi)} \centerdot \neg s(\alpha) \wedge (s,t) \Rightarrow (s', t)$$

$$\forall \alpha \in \Sigma \centerdot \forall s \in S^{[\pi]} \centerdot \forall t \in S^{(\pi)} \centerdot \forall t' \in \delta^{(\pi)}(\alpha, t) \centerdot \neg s(\alpha) \wedge (s,t) \Rightarrow (s, t')$$

Local Actions: Asynchronous interleaving:

$$\forall \alpha \in \Sigma^{[\pi]} - \Sigma \centerdot \forall s \in S^{[\pi]} \centerdot \forall s' \in \delta^{[\pi]}(\alpha, s) \centerdot \forall t \in S^{(\pi)} \centerdot (s,t) \Rightarrow (s', t)$$

$$\forall \alpha \in \Sigma^{(\pi)} - \Sigma \centerdot \forall s \in S^{[\pi]} \centerdot \forall t \in S^{(\pi)} \centerdot \forall t' \in \delta^{(\pi)}(\alpha, t) \centerdot (s,t) \Rightarrow (s, t')$$

As mentioned before, the global set of constraints Φ is obtained by collecting together the constraints in each $\Phi(\pi)$. Observe that any solution ν to Φ has the following property. Let $\Sigma_C = \{\alpha \mid \nu(s(\alpha)) = 1\}$. Then we have $\forall \pi \in \Pi \centerdot \; \mathcal{L}((M^{[\pi]}\downharpoonright_{\Sigma_C}) \times (M^{(\pi)}\downharpoonright_{\Sigma_C})) = \emptyset$. But since $\mathcal{L}(M^{[\pi]}) = \{\pi\}$, the above statement is equivalent to $\forall \pi \in \Pi \centerdot \; (\pi\downharpoonright_{\Sigma_C}) \notin \mathcal{L}(M^{(\pi)}\downharpoonright_{\Sigma_C})$, which is further equivalent to $\forall \pi \in \Pi \centerdot \; \mathit{Elim}(\pi, \Sigma_C)$. Thus, Σ_C eliminates all counterexamples. Finally, since we want the minimal such Σ_C, we minimize the number of chosen actions via the following objective function:

$$\min \sum_{\alpha \in \Sigma} s(\alpha). \tag{7}$$

Example 2 Consider Fig. 6, in which there is one counterexample π, and an FA $M^{(\pi)} = M_1 \times \bar{\varphi}$ on which π can be simulated if $\Sigma_c = \emptyset$. The state variables are (s_i, t_j) for $i, j \in [0..2]$ and the choice variables are

$s(\alpha), s(\beta)$. The constraints are:

$$\begin{array}{lll}
\mathit{Initialization}: & (s_0,t_0) & \\
\mathit{Acceptance}: & \neg(s_2,t_2) & \\
\mathit{SharedActions}: & (s_0,t_1) \rightarrow (s_1,t_2) & (s_1,t_0) \rightarrow (s_2,t_1) \\
& (s_0,t_0) \wedge \neg s(\alpha) \rightarrow (s_1,t_0) & (s_1,t_0) \wedge \neg s(\beta) \rightarrow (s_2,t_0) \\
& (s_0,t_1) \wedge \neg s(\alpha) \rightarrow (s_1,t_1) & (s_1,t_1) \wedge \neg s(\beta) \rightarrow (s_2,t_1) \\
& (s_0,t_2) \wedge \neg s(\alpha) \rightarrow (s_1,t_2) & (s_1,t_2) \wedge \neg s(\beta) \rightarrow (s_2,t_2) \\
& (s_0,t_0) \wedge \neg s(\beta) \rightarrow (s_0,t_1) & (s_0,t_1) \wedge \neg s(\alpha) \rightarrow (s_0,t_2) \\
& (s_1,t_0) \wedge \neg s(\beta) \rightarrow (s_1,t_1) & (s_1,t_1) \wedge \neg s(\alpha) \rightarrow (s_1,t_2) \\
& (s_2,t_0) \wedge \neg s(\beta) \rightarrow (s_2,t_1) & (s_2,t_1) \wedge \neg s(\alpha) \rightarrow (s_2,t_2)
\end{array}$$

Since there are no local actions, these are all the constraints. The objective is to minimize $s(\alpha) + s(\beta)$. The optimal solution is $s(\alpha) = s(\beta) = 1$, corresponding to the fact that both actions need to be in Σ_C in order to eliminate π. $\square$

6 Experiments

We implemented our technique in ComFoRT [8] and experimented with a set of benchmarks derived from real-life source code. All our experiments were carried out on quad 2.4 GHz machine with 4 GB RAM running RedHat Linux 9. We used PBS version 2.1 [1] to solve the Pseudo-Boolean constraints. The benchmarks were derived from the source code of OpenSSL version 0.9.6c. Specifically, we used the code that implements the handshake between a client and a server at the beginning of an SSL session. We designed a suite of 10 examples, each focusing on a different property (involving a sequence of message-passing events) that a correct handshake should exhibit. For instance, the first example (SSL-1) was aimed at verifying that a handshake is always initiated by a client and never by a server.

The first set of experiments were aimed at evaluating our proposed improvements separately, and in conjunction with each other in the context of AG-NC. We also performed a second set of experiments

	Cand.		Memb.				(Time) $\neg T_1$				(Time) T_1			
Name	queries		queries		$\|\Sigma\|$		$\neg T_2$		T_2		$\neg T_2$		T_2	
	$\neg T_1$	T_1	$\neg T_2$	T_2	$\neg T_3$	T_3	$\neg T_3$	T_3	$\neg T_3$	T_3	$\neg T_3$	T_3	$\neg T_3$	T_3
SSL-1	2.2	**2.0**	37.5	**4.5**	12	**1**	25.4	19.7	12.3	20.0	23.8	20.1	**10.5**	20.5
SSL-2	**5.0**	5.2	101.5	**11.5**	12	**4**	31.5	40.0	**12.6**	30.0	32.4	44.6	13.7	30.2
SSL-3	8.5	**7.5**	163.0	**28.0**	12	**4**	43.8	49.1	**14.5**	35.3	45.6	48.9	15.6	35.5
SSL-4	13.0	**10.5**	248.0	**56.5**	12	**4**	63.0	67.5	**17.4**	58.1	61.5	67.7	18.6	48.4
SSL-5	3.2	**3.0**	73.0	**9.5**	12	**1**	33.8	22.3	**13.6**	24.1	36.2	22.2	13.8	22.2
SSL-6	**6.8**	7.2	252.0	**36.5**	12	**2**	102.8	30.6	24.2	29.0	102.2	43.3	**23.1**	29.8
SSL-7	9.8	**8.0**	328.8	**52.5**	12	**2**	139.9	44.4	**27.8**	43.9	138.2	38.6	28.2	40.6
SSL-8	15.0	**13.0**	443.0	**77.5**	12	**3**	183.3	73.6	37.1	67.9	184.0	73.2	**35.8**	64.2
SSL-9	23.5	**18.2**	568.0	**109.5**	12	**3**	234.1	120.5	44.1	133.7	236.2	133.4	**41.0**	109.3
SSL-10	25.5	**22.0**	689.5	**128.5**	12	**3**	293.9	188.6	48.4	168.1	297.0	179.9	**45.9**	169.7
Avg.	10.8	**9.2**	290.0	**51.0**	12	**2**	115.1	65.6	25.2	61.0	115.7	67.2	**24.6**	57.1

Fig. 7 Experimental Results for the Non-Circular Rule AG-NC (see Eq. 1)

using the same improvements and benchmarks set, but in the context of the circular AG rule AG-C (Eq. 2). The results for the two rules are described in Figures 7 and 8 respectively. The columns labeled with T_i and $\neg T_i$ contain results with/without the i^{th} improvement for $i \in \{1, 2, 3\}$. The row labeled "Avg." contains the arithmetic mean for the rest of the column. Best figures are boldfaced. Note that entries under the Membership queries and Candidate queries are fractional since they represent the average over the four possible values of the remaining two improvements. Specifically, these are improvements 2 and 3 for Candidate queries, and improvements 1 and 3 for Membership queries.

We observe that the improvements lead to the expected results in terms of reducing the number of queries and the size of assumption alphabets. The first improvement entails somewhat fewer candidate queries on average, but seems to have only a negligible effect on run-times. The second and third improvements, on the other hand, lead to significant reductions in overall verification time, by a factor of over 4.7 (AG-NC)

Name	Cand. queries		Memb. queries		$\|\Sigma\|$		(Time) $\neg T_1$				(Time) T_1			
							$\neg T_2$		T_2		$\neg T_2$		T_2	
	$\neg T_1$	T_1	$\neg T_2$	T_2	$\neg T_3$	T_3	$\neg T_3$	T_3	$\neg T_3$	T_3	$\neg T_3$	T_3	$\neg T_3$	T_3
SSL-1	3.5	**3.0**	84.0	**7**	12	**1**	17.9	10.0	12.5	10.1	16.7	**10.0**	10.7	11.2
SSL-2	3.2	**3.0**	90.0	**9**	12	**4**	25.3	12.6	14.4	11.5	26.1	12.4	14.0	**11.1**
SSL-3	3.8	**3.5**	143.0	**15**	12	**4**	39.6	13.2	14.7	**11.5**	41.2	13.2	15.5	11.6
SSL-4	4.2	**4.0**	209.0	**23**	12	**4**	60.1	12.3	17.8	14.1	61.1	**12.2**	16.8	12.2
SSL-5	4.5	**4.0**	164.0	**15**	12	**1**	32.1	**10.4**	12.9	11.6	32.0	11.7	13.7	11.6
SSL-6	7.8	**7.2**	560.0	**63**	12	**2**	127.2	13.2	25.7	**12.5**	126.6	12.9	24.0	14.2
SSL-7	10.8	**9.5**	954.0	**115**	12	**2**	242.3	14.5	36.8	15.6	240.6	14.0	34.9	**14.0**
SSL-8	12.2	**10.5**	1190.0	**147**	12	**3**	319.2	17.6	48.8	17.5	322.5	**16.3**	45.5	17.2
SSL-9	13.8	**11.5**	1452.0	**183**	12	**3**	408.7	**17.3**	57.5	19.4	409.3	17.3	53.7	19.6
SSL-10	8.8	**7.2**	878.0	**113**	12	**3**	407.5	18.9	55.7	18.8	405.3	19.1	51.8	**18.2**
Avg.	7.0	**6.2**	572.0	**69**	12	**2**	168.0	14.0	29.7	14.3	168.1	**13.9**	28.1	14.1

Fig. 8 Experimental Results for the Circular Rule AG-C (see Eq. 2)

and 12 (AG-C) on average. Interestingly, the two improvements have different degrees of effectiveness for the two rules. While selective membership queries is more beneficial for AG-NC, it is outclassed by assumption alphabet minimization in the case of AG-C.

References

1. F. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: A Backtrack Search Pseudo-Boolean Solver and Optimizer. In *Proceedings of the 5th International Symposium on the Theory and Applications of Satisfiability Testing (SAT '02)*, pages 346–353, Cincinnati, OH, May 6–9, 2002. Cincinnati, OH, May 2002. University of Cincinnati. http://gauss.ececs.uc.edu/Conferences/SAT2002/sat2002list.html.
2. R. Alur, P. Cerny, G. Gupta, P. Madhusudan, W. Nam, and A. Srivastava. Synthesis of Interface Specifications for Java Classes. In Jens Palsberg and Martin Abadi, editors, *Popl05*, pages 98–109, Long Beach, CA, January 12–14, 2005. New York, NY, January 2005. Association for Computing Machinery.

3. Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic compositional verification by learning assumptions. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 548–562, Edinburgh, Scotland, July 6–10, 2005. New York, NY, July 2005. Springer-Verlag.
4. Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, November 1987.
5. 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.
6. Howard Barringer, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Proof rules for automated compositional verification. In *Proceedings of the 2nd Workshop on Specification and Verification of Component Based Systems (SAVCBS '03)*, pages 14–21, Helsinki, Finland, September 1–2, 2003. Ames, Iowa: Iowa State University, September 2003.
7. Sagar Chaki, Edmund M. Clarke, Nishant Sinha, and Prasanna Thati. Automated assume-guarantee reasoning for simulation conformance. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 534–547, Edinburgh, Scotland, July 2005. Springer-Verlag.
8. Sagar Chaki, James Ivers, Natasha Sharygina, and Kurt Wallnau. The ComFoRT Reasoning Framework. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 164–169, Edinburgh, Scotland, July 6–10, 2005. New York, NY, July 2005. Springer-Verlag.
9. Sagar Chaki and Nishant Sinha. Assume-guarantee reasoning for deadlock. In *Proc. of FMCAD*, 2006.
10. Sagar Chaki and Ofer Strichman. Optimized L* for assume-guarantee reasoning. March 2007. (To be published).
11. 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.
12. J. M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In Hubert Garavel and John Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346, Warsaw, Poland, April 7–11, 2003. New York, NY, April 2003. Springer-Verlag.

13. Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, pages 213–224, Los Angeles, CA, May 1999. IEEE Computer Society.

14. Mihaela Gheorghiu, Dimitra Giannakopoulou, and Corina Păsăreanu. Refining Interface Alphabets for Compositional Verification. In *Proc. of TACAS*, 2007.

15. D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption Generation for Software Component Verification. In *Proceedings of the 17th International Conference on Automated Software Engineering (ASE '02)*, pages 3–12, Edinburgh, Scotland, September 23–27, 2002. Los Alamitos, CA, September 2002. IEEE Computer Society.

16. A. Groce, D. Peled, and M. Yannakakis. Adaptive Model Checking. In Joost-Pieter Katoen and Perdita Stevens, editors, *Proceedings of the Eighth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 357–370, Grenoble, France, April 8-12, 2002. New York, NY, April 2002. Springer-Verlag.

17. Peter Habermehl and Tomas Vojnar. Regular model checking using inference of regular languages. In *Proceedings of the 6th International Workshop on Verification of Infinite-State Systems (INFINITY '04)*, volume 138(3) of *Electronic Notes in Theoretical Computer Science*, pages 21–36, December 2005.

18. D. Johnson. Approximation algorithms for combinatorial problems. *J. Computer and System Sciences*, (9):256–278, 1974.

19. Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, Princeton, NJ, 1994.

20. Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. In Jianping Wu, Samuel T. Chanson, and Qiang Gao, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE '99)*, volume 156 of *IFIP Conference Proceedings*, pages 225–240, Beijing, China, October 1999. Kluwer Academic Publishers.

21. A. Pnueli. In Transition from Global to Modular Temporal Reasoning About Programs. *Logics and Models of Concurrent Systems*, 13:123–144, 1985.

22. R. L. Rivest and R. E. Schapire. Inference of Finite Automata Using Homing Sequences. *Information and Computation*, 103(2):299–347, 1993.