Automated Circular Assume-Guarantee Reasoning

Karam Abd Elkader[1], Orna Grumberg[1], Corina S. Păsăreanu[2], and Sharon Shoham[3(✉)]

[1] Technion – Israel Institute of Technology, Haifa, Israel
[2] CMU/NASA Ames Research Center, USA
[3] The Academic College of Tel aviv Yaffo, Tel Aviv, Israel
sharon.shoham@gmail.com

Abstract. Compositional verification techniques aim to decompose the verification of a large system into the more manageable verification of its components. In recent years, compositional techniques have gained significant successes following a breakthrough in the ability to automate assume-guarantee reasoning. However, automation is still restricted to simple acyclic assume-guarantee rules.

In this work, we focus on automating *circular* assume-guarantee reasoning in which the verification of individual components mutually depends on each other. We use a sound and complete circular assume-guarantee rule and we describe how to automatically build the assumptions needed for using the rule. Our algorithm accumulates *joint* constraints on the assumptions based on (spurious) counterexamples obtained from checking the premises of the rule, and uses a SAT solver to synthesize minimal assumptions that satisfy these constraints.

We implemented our approach and compared it with an established learning-based method that uses an acyclic rule. In all cases, the assumptions generated for the circular rule were significantly smaller, leading to smaller verification problems. Further, on larger examples, we obtained a significant speedup as well.

1 Introduction

Compositional verification techniques aim to break up the global verification of a program into local, more manageable, verification of its individual components. The environment for each component, consisting of the other program's components, is replaced by a "small" *assumption*, making each verification task easier. This style of reasoning is often referred to as *Assume-Guarantee* (AG) reasoning [17,20].

Progress has been made on automating compositional reasoning using learning and abstraction-refinement techniques for iterative building of the necessary assumptions [7,19,3,4,2,5,6]. This work has been done mostly in the context of applying a simple compositional assume-guarantee rule, where assumptions and properties are related in an *acyclic* manner. For example, in a two component program, suppose component M_1 guarantees property P under assumption A on its environment. Further suppose that M_2 unconditionally guarantees A. Then it follows that the composition $M_1||M_2$ also satisfies P (denoted here as rule **NonCIRC-AG**).

However, there is another important category of rules that involve *circular reasoning*. These rules use inductive arguments, over time, formulas to be checked, or both, e.g. [17,14,15,1], which makes automation challenging. Circular assume-guarantee rules have been successfully used in scaling model checking, and have often been found to

N. Bjørner and F. de Boer (Eds.): FM 2015, LNCS 9109, pp. 23–39, 2015.
DOI: 10.1007/978-3-319-19249-9_3

be more effective than non-circular rules [14,15,16,21,12,11]. Further, they could naturally exploit the inherent circular dependency exhibited by the verified systems, but their applicability has been hindered by the manual effort involved in defining assumptions.

In this work we propose a novel *circular* compositional verification technique that is fully *automated*. The technique uses the following assume-guarantee circular rule **CIRC-AG**, for proving that $M_1||M_2 \models P$, based on assumptions g_1 and g_2. Components, properties and assumptions are Labeled Transition Systems (LTSs).

$$\frac{\begin{array}{ll}\text{(Premise 1)} & M_1 \models g_2 \triangleright g_1 \\ \text{(Premise 2)} & M_2 \models g_1 \triangleright g_2 \\ \text{(Premise 3)} & g_1||g_2 \models P\end{array}}{M_1||M_2 \models P}$$

Similar rules have been studied before [15,18,9]. The rule is both *sound* and *complete*. Premises 1 and 2 of the rule use inductive arguments to ensure soundness and have the form $M \models A \triangleright P$, which means that for every trace σ of size k, if σ is in the language of M, and its prefix of size $k-1$ is in the language of A then σ is also in the language of P. Intuitively, premises 1 and 2 prove, in a *compositional and inductive* manner, that every trace in the language of $M_1||M_2$ is also included in the language of $g_1||g_2$. Premise 3 ensures that every trace in the language of $g_1||g_2$ is also included in the language of P, thus the consequence of the rule is obtained. Completeness of the rule stems from the fact that M_1 and M_2 (restricted to appropriate alphabets) can be used for g_1 and g_2 in a successful application of the rule.

Coming up manually with assumptions g_1 and g_2 that are small and also satisfy the premises of the rule is difficult. We propose an algorithm, Automated Circular Reasoning (ACR), for the automated generation of the assumptions. In ACR the assumptions are initially approximate and are iteratively refined based on counterexamples obtained from checking the rule premises and found to be spurious (i.e. do not indicate real errors). Refinement is performed using a SAT solver over a set of constraints that determine how the assumptions should be refined in order to avoid producing the same counterexample in subsequent iterations. The algorithm is guaranteed to terminate, returning either minimal assumptions that satisfy the rule premises (meaning that the property holds) or a real counterexample (indicating a property violation).

Our search for minimal assumptions using SAT is inspired by [10]. However, in [10] a single (separating) assumption is generated, with the goal of automating non-circular reasoning. ACR, on the other hand, searches for two mutually dependent assumptions to be used with circular reasoning. Finding such assumptions poses unusual challenges since they need to be generated in a tightly related manner. We achieve this by constraining the assumption refinement with boolean combinations of requirements that certain traces must or must not be included in the language of the updated assumptions. For example, we may require "trace σ_1 must not be in g_1 **or** trace σ_2 must be in g_2". The SAT encoding of this constraint makes sure that at least one of its disjuncts will be satisfied. Solving the constraints for increasing number of states in $|g_1|+|g_2|$, yields the minimal candidate assumptions to be used in the next iteration of ACR. We establish the correctness of our ACR algorithm (proofs are omitted due to space constraints).

To the best of our knowledge, our work is the first to fully automate circular assume-guarantee reasoning. We implemented our algorithm and compared it with an established

learning-based method that uses the acyclic rule NonCIRC-AG [7]. Our experiments indicate that the assumptions generated using the circular rule can be much smaller, leading to smaller verification problems, both in the number of explored states and the analysis time.

2 Preliminaries

Let Act be the universal set of observable actions and let τ denote a local action, unobservable to a component's environment.

Definition 1. *A* Labeled Transition System *(LTS) M is a quadruple $(Q, \alpha M, \delta, q_0)$ where Q is a finite set of states, $\alpha M \subseteq Act$ is a finite set of observable actions called the alphabet of M, $\delta \subseteq Q \times (\alpha M \cup \tau) \times Q$ is a transition relation, and $q_0 \in Q$ is the initial state.*

M is *nondeterministic* if it contains a τ transition or if there exist $(q, a, q'), (q, a, q'') \in \delta$ such that $q' \neq q''$. Otherwise, M is *deterministic* (denoted as DLTS). We write $\delta(q, a) = \perp$ if there is no q' such that $(q, a, q') \in \delta$. For a DLTS, we write $\delta(q, a) = q'$ to denote that $(q, a, q') \in \delta$.

Note. A non-deterministic LTS can be converted to a deterministic LTS that accepts the same language. However the deterministic LTS might have exponentially many more states than the non-deterministic LTS.

Paths and Traces. A *trace* σ is a sequence of observable actions. We use σ_i to denote the prefix of σ of length i. A *path* in an LTS $M = (Q, \alpha M, \delta, q_0)$ is a sequence $p = q_0, a_0, q_1, a_1 \cdots, a_{n-1}, q_n$ of alternating states and observable or unobservable actions of M, such that for every $k \in \{0, \ldots, n-1\}$ we have $(q_k, a_k, q_{k+1}) \in \delta$. The *trace* of p is the sequence $b_0 b_1 \cdots b_l$ of actions along p, obtained by removing from $a_0 \cdots a_{n-1}$ all occurrences of τ. The set of all traces of paths in M is called the *language* of M, denoted $L(M)$. A trace σ is *accepted* by M if $\sigma \in L(M)$. Note that $L(M)$ is prefix-closed and that the empty trace, denoted by ϵ, is accepted by any LTS.

Projections. For $\Sigma \subseteq Act$, we use $\sigma{\downarrow_\Sigma}$ to denote the trace obtained by removing from σ all occurrences of actions $a \notin \Sigma$. $M{\downarrow_\Sigma}$ is defined to be the LTS over alphabet Σ obtained by renaming to τ all the transitions labeled with actions that are not in Σ. Note that $L(M{\downarrow_\Sigma}) = \{\sigma{\downarrow_\Sigma} \mid \sigma \in L(M)\}$.

Parallel Composition. Given two LTSs M_1 and M_2 over alphabet αM_1 and αM_2, respectively, their *interface alphabet* αI consists of their common alphabet. That is, $\alpha I = \alpha M_1 \cap \alpha M_2$. The parallel composition operator $||$ is a commutative and associative operator that combines the behavior of two components by synchronizing on the actions in their interface and interleaving the remaining actions.

Let $M_1 = (Q_1, \alpha M_1, \delta_1, q_{0_1})$ and $M_2 = (Q_2, \alpha M_2, \delta_2, q_{0_2})$ be two LTSs. Then $M_1 || M_2$ is an LTS $M = (Q, \alpha M, \delta, q_0)$, where $Q = Q_1 \times Q_2$, $q_0 = (q_{0_1}, q_{0_2})$, $\alpha M = \alpha M_1 \cup \alpha M_2$, and δ is defined as follows where $a \in \alpha M \cup \{\tau\}$:

- if $(q_1, a, q_1') \in \delta_1$ for $a \notin \alpha M_2$, then $((q_1, q_2), a, (q_1', q_2)) \in \delta$ for every $q_2 \in Q_2$,
- if $(q_2, a, q_2') \in \delta_2$ for $a \notin \alpha M_1$, then $((q_1, q_2), a, (q_1, q_2')) \in \delta$ for every $q_1 \in Q_1$,
- if $(q_1, a, q_1') \in \delta_1$ and $(q_2, a, q_2') \in \delta_2$ for $a \neq \tau$, then $((q_1, q_2), a, (q_1', q_2')) \in \delta$.

Lemma 1. *[7] For every* $t \in (\alpha M_1 \cup \alpha M_2)^*$, $t \in L(M_1||M_2)$ *if and only if* $t{\downarrow_{\alpha M_1}} \in L(M_1)$ *and* $t{\downarrow_{\alpha M_2}} \in L(M_2)$.

Example 1. Consider the example in Figure 1. This is a variation of the example of [7] modified to illustrate circular dependencies. LTSs In and Out have interface alphabet $\{send, ack\}$. Their composition $In||Out$ is an LTS where the transition from state 0 to 1 in component In (labeled with *ack*) never takes place, since there is no corresponding matching transition in component Out. Similarly the transition from state 2 to 3 in component Out (labeled with *send*) never takes place. As a result, $In||Out$ simply repeats the trace $\langle in, send, out, ack \rangle$.

Properties and Satisfiability. A *safety property* is defined as an LTS P, whose language $L(P)$ defines the set of acceptable behaviors over the alphabet αP of P. An LTS M over $\alpha M \supseteq \alpha P$ satisfies P, denoted $M \models P$, if $\forall \sigma \in L(M).\sigma{\downarrow_{\alpha P}} \in L(P)$. To check a safety property P, its LTS is transformed into a deterministic LTS, which is also completed by adding an error state π and adding transitions from every state q in the deterministic LTS into π for all the missing outgoing actions of q; the resulting LTS is called an *error LTS*, denoted by P_{err}. Checking that $M \models P$ is done by checking that π is not reachable in $M||P_{err}$.

A trace $\sigma \in \alpha M^*$ is a *counterexample* for $M \models P$ if $\sigma \in L(M)$ but $\sigma{\downarrow_{\alpha P}} \notin L(P)$.

The $Order$ LTS from Figure 1 depicts a safety property satisfied by $In||Out$. Note that neither In, nor Out, satisfy this property individually. For example, the trace $\langle in, send, ack, ack \rangle$ of In is a counterexample for $In \models Order$.

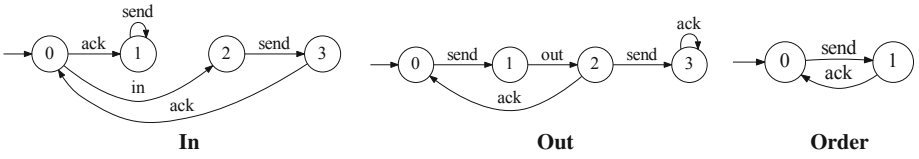

Fig. 1. LTSs describing the In and Out components and the $Order$ property

3 Circular Assume-Guarantee Reasoning

In this section we formally establish the soundness and completeness of the circular rule CIRC-AG introduced in Section 1 (proofs are omitted due to space constraints). We start by defining inductive properties. CIRC-AG uses formulas of the form $M \models A \triangleright P$, where M is a component, P is a property, and A is an assumption about M's environment. To ensure soundness of the circular rule the assume-guarantee formula is defined using induction over finite traces.

Definition 2. *Let M, A and P be LTSs over $\alpha M, \alpha A$ and αP respectively, such that $\alpha P \subseteq \alpha M$. We say that $M \models A \triangleright P$ holds if $\forall k \geq 1$ $\forall \sigma \in (\alpha M \cup \alpha A)^*$ of length k such that $\sigma{\downarrow_{\alpha M}} \in L(M)$, if $\sigma_{k-1}{\downarrow_{\alpha A}} \in L(A)$ then $\sigma{\downarrow_{\alpha P}} \in L(P)$.*

Intuitively, the formula states that if a trace in M satisfies the assumption A up to step $k-1$, it should guarantee P up to step k. As an example consider the LTSs In from Figure 1 and g_1 and g_2 from Figure 2. Then $In \models g_2 \triangleright g_1$. On the other hand, $In \not\models g_1 \triangleright g_2$ since the trace $\sigma = \langle in, send, ack, ack\rangle \in L(In)$ is such that $\sigma_{k-1}{\downarrow_{\alpha g_1}} = \langle send, ack\rangle \in L(g_1)$, but $\sigma{\downarrow_{\alpha g_2}} = \langle send, ack, ack\rangle \notin L(g_2)$. σ is therefore a counterexample for $In \models g_1 \triangleright g_2$.

Definition 3. *A trace $\sigma \in (\alpha M \cup \alpha A)^*$ of length k is a* counterexample *for $M \models A \triangleright P$ if $\sigma{\downarrow_{\alpha M}} \in L(M)$ and $\sigma_{k-1}{\downarrow_{\alpha A}} \in L(A)$ but $\sigma{\downarrow_{\alpha P}} \notin L(P)$.*

Soundness and Completeness of Rule CIRC-AG. To establish the soundness of rule CIRC-AG we have the following requirements. M_1, M_2 and P are LTSs where $\alpha P \subseteq \alpha M_1 \cup \alpha M_2$. Moreover, g_1, g_2 are LTSs, used as *assumptions* in the rule, such that $\alpha M_1 \cap \alpha P \subseteq \alpha g_1$ and $\alpha M_2 \cap \alpha P \subseteq \alpha g_2$.

The following lemmas include several observations that are useful both in the soundness and completeness proofs and in the algorithm for automatic generation of assumptions g_1 and g_2, needed for the rule.

Lemma 2. *Let g_1 and g_2 be LTS assumptions successfully used in CIRC-AG, such that $\alpha M_i \cap \alpha P \subseteq \alpha g_i$. Then* **(1)** *$M_1||M_2 \models g_1||g_2$.* **(2)** *$M_1||g_2 \models P$ and $M_2||g_1 \models P$.*

Lemma 3. *Let M_1, M_2, P be LTSs over $\alpha M_1, \alpha M_2, \alpha P$ respectively. Let $\alpha g_1 \supseteq \alpha I \cup (\alpha M_1 \cap \alpha P)$ and $\alpha g_2 \supseteq \alpha I \cup (\alpha M_2 \cap \alpha P)$. Then $M_1||M_2 \models P$ if and only if $M_1{\downarrow_{\alpha g_1}}||M_2{\downarrow_{\alpha g_2}} \models P$.*

Theorem 1. *The Rule CIRC-AG is sound and complete.*

Soundness states that if there exist LTS assumptions g_1 and g_2 that satisfy all premises of CIRC-AG, then $M_1||M_2 \models P$. This result follows from Lemma 2, Item (1). Completeness states that if $M_1||M_2 \models P$ holds we can always find g_1 and g_2 such that the premises of the rule hold. Indeed completeness is established by showing that if $M_1||M_2 \models P$, then $g_1 = M_1{\downarrow_{\alpha g_1}}$ and $g_2 = M_2{\downarrow_{\alpha g_2}}$ where $\alpha g_1 = \alpha M_1 \cap (\alpha M_2 \cup \alpha P)$ and $\alpha g_2 = \alpha M_2 \cap (\alpha M_1 \cup \alpha P)$, satisfy the premises of the rule.

Example 2. Consider our running example (Figure 1), and consider the assumptions g_1 and g_2 depicted in Figure 2, over alphabet $\alpha g_1 = \alpha In \cap (\alpha Out \cup \alpha Order)$ and $\alpha g_2 = \alpha Out \cap (\alpha In \cup \alpha Order)$. In both cases $\alpha g_i = \{send, ack\}$. As stated above, $In \models g_2 \triangleright g_1$. Similarly, $Out \models g_1 \triangleright g_2$. Moreover, $g_1||g_2 \models Order$. It follows that $In||Out \models Order$ can be verified using CIRC-AG with g_1 and g_2 as assumptions.

4 Automatic Reasoning with CIRC-AG

We describe an iterative algorithm to automate the application of rule CIRC-AG by automating the assumption generation.

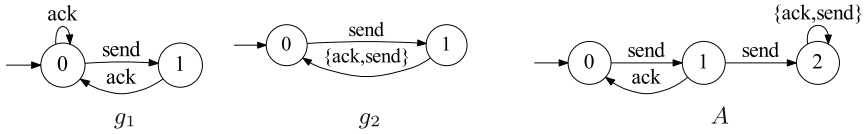

Fig. 2. LTSs describing the assumptions g_1 and g_2 generated by ACR, and the assumption A generated with L*. $\alpha g_1 = \alpha g_2 = \alpha A = \{send, ack\}$

Checking Inductive Properties. We first introduce a simple algorithm that checks if an inductive property of the form $M \models A \triangleright P$, where $\alpha P \subseteq \alpha M$, holds and if it does not, it returns a counterexample. To do so, we consider the LTS $M||A||P_{err}$. We label its states by (parameterized) propositions err_a, where $a \in \alpha P$. (s_M, s_A, s_P) is labeled by err_a if s_M has an outgoing transition in M labeled by a, but the corresponding transition (labelled by a) leads to π in P_{err}. We then check if a state q labeled by err_a is reachable in $M||A||P_{err}$. If so, then the algorithm returns the trace of a path from q_0 to q extended with action a as a counterexample. Intuitively, such a path to q represents a trace in M that satisfies A (because it is a trace in $M||A$) such that if we extend it by a we get a trace in M violating P.

Overview of the Main Algorithm. We propose an iterative algorithm to automate the application of the rule CIRC-AG by automating the assumption generation. Previous work used approximate iterative techniques based on automata learning or abstraction refinement to automate the assumption generation in the context of *acyclic* rules [7,19,3,4,2,5,6]. A different approach [10] used a SAT solver over a set of constraints encoding how the assumptions should be updated to find minimal assumptions; the method was shown to work well in practice, in the context of the same *acyclic* rule. We follow the latter approach here and we adapt it to reasoning for *cyclic* rules and checking inductive assume-guarantee properties. As mentioned, this is challenging due to the mutual dependencies between the two assumptions that we need to generate. We achieve this by constraining the assumptions with boolean combinations of requirements that certain traces must or must not be included in the language of the updated assumptions.

Algorithm 1 describes our Automated Circular Reasoning (ACR) algorithm for checking $M_1||M_2 \models P$ using the rule CIRC-AG.

We fix the alphabet of the assumptions g_1 and g_2 to be $\alpha g_1 = \alpha M_1 \cap (\alpha M_2 \cup \alpha P)$ and $\alpha g_2 = \alpha M_2 \cap (\alpha M_1 \cup \alpha P)$. By the completeness proof of the rule, this suffices.

ACR maintains a set C of *membership* constraints on g_1 and g_2. At each iteration it calls GENASSMP (described in Section 6) to synthesize, using a SAT solver, new minimal assumptions g_1 and g_2 that satisfy all the constraints in C. GENASSMP also receives as input a parameter k which provides a lower bound on the total number of states in the assumptions we look for. This avoids searching for smaller assumptions that cannot satisfy C. The algorithm then invokes APPLYAG (described in Section 5) to check the three premises of rule CIRC-AG using the obtained assumptions g_1 and g_2. APPLYAG may return a conclusive result: either "$M_1||M_2 \models P$" or "$M_1||M_2 \not\models P$",

Algorithm 1. Main algorithm for automating rule CIRC-AG for checking $M_1||M_2 \models P$

```
1: procedure ACR(M_1, M_2, P)
2:     Initialize: C = ∅, k = 2
3:     repeat
4:         (g_1, g_2) = GENASSMP(C, k)
5:         (C', Result) = APPLYAG(M_1, M_2, P, g_1, g_2)
6:         C = C ∪ C', k = |g_1| + |g_2|
7:     until (Result ≠ "continue")
8:     return Result
9: end procedure
```

in which case ACR terminates. If no conclusive result is obtained, it means that g_1 and g_2 do not satisfy the premises of the rule. Further, the counterexamples demonstrating the falsification of the premises are not suitable for concluding $M_1||M_2 \not\models P$, i.e. they are spurious. In this case APPLYAG returns "continue" together with new membership constraints that determine how the assumptions should be refined. The new constraints are added to C. Note that since the set C of constraints is monotonically increasing, any new pair (g_1', g_2') that satisfies it also satisfies previous sets of constraints. The previous set was satisfied by assumptions whose total size is $|g_1| + |g_2|$ but not smaller. Thus, we should start our search for new (g_1', g_2') from $k = |g_1| + |g_2|$ number of states. k is updated accordingly (line 6).

Example 3. The assumptions g_1 and g_2 from Figure 2 used to verify $In||Out \models Order$ with CIRC-AG were obtained by ACR in the 7th iteration. The LTS A from Figure 2 describes the assumption obtained with the algorithm of [7], which is based on acyclic rule NonCIRC-AG and uses L* for assumption generation. Notice that both g_1 and g_2 are smaller than A (and our experiments show that they can be much smaller in practice). The reason is that, after a successful application of CIRC-AG, $g_1||g_2$ overapproximates $M_1||M_2$. This means that each g_i overapproximates the part of M_i restricted to the composition with the other component. For example g_1 does not include the traces leading to state 1 from *In* since they do not participate in the composition. Similarly g_2 does not include the traces leading to state 3 in *Out*. In contrast, for the acyclic rule, the assumption A has to overapproximate M_2 (*Out*) as a whole. Therefore, CIRC-AG can result in substantially smaller assumptions, as also demonstrated by our experiments.

Membership Constraints. Membership constraints are used by our algorithm to gather information about traces that need to be in $L(g_i)$ or must not be in $L(g_i)$, for $i = 1, 2$. Thus they allow us to encode dependencies between the languages of the two assumptions $L(g_1)$ and $L(g_2)$. The constraints are defined by formulas with a special syntax and semantics, as defined below.

Definition 4. Membership constraints formulas *over* $(\alpha g_1, \alpha g_2)$ *are defined inductively as follows: For every* $\sigma_1 \in \alpha g_1^*$ *and* $\sigma_2 \in \alpha g_2^*$ *the formulas* $+(\sigma_1, 1)$, $-(\sigma_1, 1)$, $+(\sigma_2, 2)$, $-(\sigma_2, 2)$ *are* atomic *membership constraints formulas. Further, if* c_1 *and* c_2 *are membership constraints formulas, then so are* $(c_1 \wedge c_2)$ *and* $(c_1 \vee c_2)$.

Given a membership constraints formula c, $Strings(c, i)$ is the set of prefixes of all $\sigma \in \alpha g_i{}^*$ such that $+(\sigma, i)$ or $-(\sigma, i)$ is an atomic formula in c.

Definition 5. *Let c be a membership constraints formula over $(\alpha g_1, \alpha g_2)$, and let A_1 and A_2 be two LTSs. The satisfaction of c by (A_1, A_2) is defined inductively. $(A_1, A_2) \models c$ if and only if $\alpha A_1 = \alpha g_1$ and $\alpha A_2 = \alpha g_2$, and:*

- *if c is an atomic formula of the form $+(\sigma, i)$ then $\sigma \in L(A_i)$.*
- *if c is an atomic formula of the form $-(\sigma, i)$ then $\sigma \notin L(A_i)$.*
- *if c is of the form $(c_1 \wedge c_2)$ then $(A_1, A_2) \models c_1$ and $(A_1, A_2) \models c_2$.*
- *if c is of the form $(c_1 \vee c_2)$ then $(A_1, A_2) \models c_1$ or $(A_1, A_2) \models c_2$.*

For a set C of membership constraints formulas over $(\alpha g_1, \alpha g_2)$, we say that A_1 and A_2 satisfy C if and only if for every $c \in C$, $(A_1, A_2) \models c$.

For example, a membership constraint of the form $+(\sigma_1, 1) \vee -(\sigma_2, 2)$ requires that $\sigma_1 \in L(g_1)$ or $\sigma_2 \notin L(g_2)$ (or both).

5 APPLYAG Algorithm

Given assumptions g_1,g_2, APPLYAG (see Algorithm 2) applies assume-guarantee reasoning by checking the three premises of rule CIRC-AG using g_1 and g_2. In the algorithm we check premises 1, 2, 3 in this order but in fact the order of the checks does not matter and the checks can be done in parallel. If all three premises are satisfied, then, since the rule is sound, it follows that $M_1||M_2 \models P$ holds (and this is returned to the user). Otherwise, at least one of the premises does not hold. Hence a counterexample σ for (at least) one of the premises is found. APPLYAG then checks if the counterexample indicates a real violation for $M_1||M_2 \models P$, as described below. If this is the case, then APPLYAG returns $M_1||M_2 \not\models P$. Otherwise APPLYAG uses the counterexample to compute a set of new membership constraints C and returns "continue" (note that in the first two cases an empty constraint set is returned).

Notation. For readability, in APPLYAG (and UPDATECONSTRAINTS) we use $\sigma\downarrow \in L(A)$ and $\sigma\downarrow \notin L(A)$ as a shorthand for $\sigma\downarrow_{\alpha A} \in L(A)$ and $\sigma\downarrow_{\alpha A} \notin L(A)$, respectively.

Checking Validity of a Counterexample. Given a counterexample σ for one of the premises of the CIRC-AG rule, APPLYAG checks if σ can be extended into a trace in $L(M_1||M_2)$ which does not satisfy P. This check is performed either by APPLYAG directly (if premise 3 fails: in lines 9-16 of APPLYAG) or by algorithm UPDATECONSTRAINTS (if one of the first two premises fails). In essence, a counterexample σ is real if $\sigma\downarrow_{\alpha g_1} \in L(M_1\downarrow_{\alpha g_1})$, $\sigma\downarrow_{\alpha g_2} \in L(M_2\downarrow_{\alpha g_2})$ and $\sigma\downarrow_{\alpha P} \notin L(P)$. This is also stated by the following lemma, which follows from Lemma 1 and Lemma 3.

Lemma 4. *If $\sigma\downarrow_{\alpha g_1} \in L(M_1\downarrow_{\alpha g_1})$, $\sigma\downarrow_{\alpha g_2} \in L(M_2\downarrow_{\alpha g_2})$ and $\sigma\downarrow_{\alpha P} \notin L(P)$, then $M_1||M_2 \not\models P$. Moreover, σ can be extended into a counterexample for $M_1||M_2 \models P$.*

For example, in line 9 of Algorithm 2, $\sigma \in (\alpha g_1 \cup \alpha g_2)^*$ is a counterexample for premise 3, hence $\sigma{\downarrow_{\alpha P}} \notin L(P)$. It therefore suffices to check if $\sigma{\downarrow_{\alpha g_1}} \in L(M_1{\downarrow_{\alpha g_1}})$ and $\sigma{\downarrow_{\alpha g_2}} \in L(M_2{\downarrow_{\alpha g_2}})$ in order to conclude that a real counterexample exists (line 11). Similarly, in Algorithm 3, $\sigma a \in (\alpha M_i \cup \alpha g_j)^*$ is a counterexample for premise i for $i \in \{1, 2\}$, hence $\sigma a{\downarrow_{\alpha M_i}} \in L(M_i)$, and since $\alpha g_i \subseteq \alpha M_i$, also $\sigma a{\downarrow_{\alpha g_i}} \in L(M_i{\downarrow_{\alpha g_i}})$. In line 3, the algorithm then checks if, in addition, $\sigma a{\downarrow_{\alpha g_j}} \in L(M_j{\downarrow_{\alpha g_j}})$ and $\sigma a{\downarrow_{\alpha P}} \notin L(P)$. If these conditions hold then by Lemma 4 the counterexample is real (line 5).

Computation of New Membership Constraints based on Counterexamples. When the counterexample found for one of the premises does not produce a real counterexample for $M_1||M_2 \models P$, then ApplyAG (or UpdateConstraints) analyzes the counterexample and computes new membership constraints to *refine* the assumptions. In essence, these constraints encode whether the counterexample trace (or a restriction of it) should be added to or removed from the languages of the two assumptions such that future checks will not produce the same counterexample again.

If premise 3 does not hold, i.e. $g_1||g_2 \not\models P$ and the reported counterexample σ is found not to be real then it should be removed from $L(g_1)$ or from $L(g_2)$ (in this way the trace will no longer be present in the composition $g_1||g_2$ for the assumptions computed in subsequent iterations). Therefore in line 14, ApplyAG adds the corresponding constraint $(-(\sigma{\downarrow_{\alpha g_1}}, 1) \vee -(\sigma{\downarrow_{\alpha g_2}}, 2))$ to C.

If either premise 1 or 2 does not hold, i.e. $M_i \not\models g_j \triangleright g_i$, then the analysis of the counterexample $\sigma_i a_i$ (for i=1 or 2) and the addition of constraints (if needed) are performed by UpdateConstraints (see Algorithm 3). Specifically, in this case $\sigma_i a_i$ should be added to $L(g_i)$ or its prefix σ_i should be removed from $L(g_j)$ (where $j \neq i$). In both cases, this ensures that checking $M_i \not\models g_j \triangleright g_i$ in subsequent iterations will no longer produce the same counterexample (see Definition 2).

We add this constraint in line 11 of Algorithm 3, where C is updated with $(-(\sigma{\downarrow_{\alpha g_j}}, j) \vee +(\sigma a{\downarrow_{\alpha g_i}}, i)$. Although this simple refinement would work for all cases, note that Algorithm 3, uses a more involved refinement. The reason is that we exploit the properties stated in Lemma 2, Items (1) and (2), to detect more elaborate constraints; using the lemma and analyzing both σ and σa allows us to *accelerate* the refinement process.

For example, in line 18, the subconstraint $+(\sigma a{\downarrow_{\alpha g_i}}, i)$ is conjoined with $-(\sigma a{\downarrow_{\alpha g_j}}, j)$. This is because Lemma 2, Item (2) establishes that $M_i||g_j \models P$ is a necessary condition for a successful application of CIRC-AG. Therefore since $\sigma a{\downarrow_{\alpha g_i}} \in L(M_i{\downarrow_{\alpha g_i}})$ and $\sigma a{\downarrow_{\alpha P}} \notin L(P)$, then $\sigma a{\downarrow_{\alpha g_j}}$ must not be in $L(g_j)$. Explanations of other cases appear as comments in the pseudocode. Note that there are more cases that we do not show in order to simplify the presentation.

Example 4. Consider the LTSs from Figure 3, produced in the 6th iteration of ACR. When trying to apply CIRC-AG with these assumptions, ApplyAG obtains the trace $\langle send, out, send\rangle$ as a counterexample for $Out \models g_1^{(6)} \triangleright g_2^{(6)}$ (premise 2). Since $\langle send, out, send\rangle{\downarrow_{\alpha g_1}} \notin L(In{\downarrow_{\alpha g_1}})$, the counterexample turns out to be spurious, and after checking the additional conditions in UpdateConstraints, $-(\langle send\rangle, 1) \vee (+(\langle send, send\rangle, 2) \wedge -(\langle send, send\rangle, 1))$ is produced in line 18 as a membership constraint in order to eliminate it in the following iterations.

Algorithm 2. Applying CIRC-AG with g_1 and g_2, and constraint updating

procedure APPLYAG(M_1, M_2, P, g_1, g_2)
  if $M_1 \not\models g_2 \triangleright g_1$ **then**
    Let $\sigma_1 a_1$ be a counterexample for $M_1 \not\models g_2 \triangleright g_1$
    return UPDATECONSTRAINTS($1, 2, M_1, M_2, P, \sigma_1 a_1$)
  else if $M_2 \not\models g_1 \triangleright g_2$ **then**
    Let $\sigma_2 a_2$ be a counterexample for $M_2 \not\models g_1 \triangleright g_2$
    return UPDATECONSTRAINTS($2, 1, M_2, M_1, P, \sigma_2 a_2$)
  else if $g_1 || g_2 \not\models P$ **then**
    Let σ be a counterexample for $g_1 || g_2 \not\models P$
    if ($\sigma\downarrow \in L(M_1 \downarrow \alpha g_1)$ && $\sigma \downarrow \in L(M_2 \downarrow \alpha g_2)$) **then**
      return ($\emptyset$, "$M_1 || M_2 \not\models P$")
    else // $\sigma \notin L(M_1{\downarrow_{\alpha g_1}} || M_2{\downarrow_{\alpha g_2}}), \sigma\downarrow \notin L(P)$
      // Remove σ from g_1 or remove σ from g_2
      $C = \{(-(\sigma{\downarrow_{\alpha g_1}}, 1) \vee -(\sigma{\downarrow_{\alpha g_2}}, 2))\}$
      return (C, "continue")
    end if
  else
    return ($\emptyset$, "$M_1 || M_2 \models P$")
  end if
end procedure

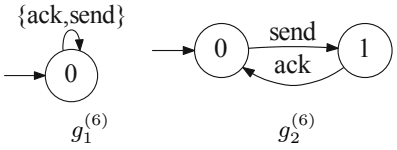

Fig. 3. LTSs produced in the 6th iteration of ACR

In the following we state the progress of assumption refinement based on spurious counterexamples.

Lemma 5. *Let σ be a spurious counterexample obtained for premise $i \in \{1, 2, 3\}$ of CIRC-AG with respect to assumptions g_1, g_2 and let C be the updated set of constraints. Then any pair of LTSs g_1' and g_2' such that $(g_1', g_2') \models C$ will no longer exhibit σ as a counterexample for premise i of CIRC-AG.*

Corollary 1. *Any pair of LTSs g_1' and g_2' such that $(g_1', g_2') \models C$ is different from every previous pair of LTSs considered by the algorithm.*

The following two lemmas state that the added membership constraints do not over-constrain the assumptions. They ensure that the "desired" assumptions that enable to verify (Lemma 6) or falsify (Lemma 7) the property are always within reach.

Lemma 6. *Suppose $M_1 || M_2 \models P$ and let g_1 and g_2 be LTSs that satisfy the premises of rule CIRC-AG. Then (g_1, g_2) satisfy every set of constraints C produced by* APPLYAG.

Lemma 7. *Let $g_1 = M_1{\downarrow_{\alpha g_1}}$ and $g_2 = M_2{\downarrow_{\alpha g_2}}$. Then (g_1, g_2) satisfy every set of constraints C produced by* APPLYAG.

Algorithm 3. Computation of constraints based on a counterexample for $M_i \models g_j \triangleright g_i$

1: // σa is a counterexample for $M_i \models g_j \triangleright g_i$, i.e. $\sigma a{\downarrow} \in L(M_i), \sigma{\downarrow} \in L(g_j), \sigma a{\downarrow} \notin L(g_i)$
2: **procedure** UPDATECONSTRAINTS($i, j, M_i, M_j, P, \sigma a$)
3: **if** $\sigma a{\downarrow} \in L(M_j{\downarrow_{\alpha g_j}})$ and $\sigma a{\downarrow} \notin L(P)$ **then**
4: // $\sigma a{\downarrow} \in L(M_i{\downarrow_{\alpha g_i}} || M_j{\downarrow_{\alpha g_j}})$ and $\sigma a{\downarrow} \notin L(P)$
5: **return** ($\emptyset$, "$M_i || M_j \not\models P$")
6: **if** $\sigma a{\downarrow} \in L(M_j{\downarrow_{\alpha g_j}})$ and $\sigma a{\downarrow} \in L(P)$ **then**
7: // Add σa to both g_i and g_j to ensure $M_1{\downarrow_{\alpha g_1}} || M_2{\downarrow_{\alpha g_2}} \models g_1 || g_2$ (Lemma 2 (1))
8: $C = \{+(\sigma a{\downarrow_{\alpha g_i}}, i), +(\sigma a{\downarrow_{\alpha g_j}}, j)\}$
9: **if** $\sigma a{\downarrow} \notin L(M_j{\downarrow_{\alpha g_j}})$ and $\sigma{\downarrow} \notin L(M_j{\downarrow_{\alpha g_j}})$ and $\sigma a{\downarrow} \in L(P)$ **then**
10: // Remove σ from g_j or add σa to g_i
11: $C = \{(-(\sigma{\downarrow_{\alpha g_j}}, j) \vee +(\sigma a{\downarrow_{\alpha g_i}}, i)\}$
12: **if** $\sigma a{\downarrow} \notin L(M_j{\downarrow_{\alpha g_j}})$ and $\sigma{\downarrow} \notin L(M_j{\downarrow_{\alpha g_j}})$ and $\sigma a{\downarrow} \notin L(P)$ and $\sigma{\downarrow} \notin L(P)$ **then**
13: // Remove σ from g_j (Because of Lemma 2 (2))
14: $C = \{-(\sigma{\downarrow_{\alpha g_j}}, j)\}$
15: **if** $\sigma a{\downarrow} \notin L(M_j{\downarrow_{\alpha g_j}})$ and $\sigma{\downarrow} \notin L(M_j{\downarrow_{\alpha g_j}})$ and $\sigma a{\downarrow} \notin L(P)$ and $\sigma{\downarrow} \in L(P)$ **then**
16: // Remove σ from g_j or (add σa to g_i and remove it from g_j)
17: // In the latter case removal of σa from g_j is due to Lemma 2 (2)
18: $C = \{(-(\sigma{\downarrow_{\alpha g_j}}, j) \vee (+(\sigma a{\downarrow_{\alpha g_i}}, i) \wedge -(\sigma a{\downarrow_{\alpha g_j}}, j)))\}$
19: **return** (C, "continue")
20: **end procedure**

6 GENASSMP Algorithm

Given a set of membership constraints C, and a lower bound k on the total number of states in $|g_1| + |g_2|$, we compute assumptions g_1 and g_2 that satisfy C. Similarly to previous work [10] we build assumptions as deterministic LTSs (even though APPLYAG is not restricted to deterministic LTSs). Technically, for each value of k starting from the given k, GENASSMP encodes the structure of the desired DLTSs g_1 and g_2 with $|g_1| + |g_2| \leq k$, as well as the membership constraints, as a SAT instance $SatEnc_k(C)$. It then searches for a satisfying assignment and obtains DLTSs g_1 and g_2 based on this assignment. k is increased only when $SatEnc_k(C)$ is unsatisfiable, hence minimal DLTSs that satisfy C are obtained.

We use the following encoding of the problem of finding whether there are DLTSs g_1 and g_2 with k states in total such that $(g_1, g_2) \models C$.

Variables used for Encoding the LTSs Structure. Let $n = \lceil \log_2(k+2) \rceil$. We use boolean vectors of length n to encode the states of g_1 and g_2, where for each of them we add a special "error" state. For each $0 \leq m \leq k+1$ we use $\overline{m}$ to denote the n-bit vector that represents the number m. We fix the vector $\overline{0}$ to represent the error state of g_1, and the vector $\overline{k+1}$ to represent the error state of g_2. We explicitly add the error states in order to distinguish between traces that are rejected by the DLTS and traces for which the behavior is unspecified. For every $i \in \{1, 2\}$:

- Let S_i include the prefixes of all traces over αg_i which are constrained in C with respect to i. That is, $S_i = \bigcup_{c \in C} Strings(c, i)$.
- For every $\sigma \in S_i$, we introduce a set of boolean variables $Var(\sigma, i) = \{v^j_{(\sigma,i)} \mid 0 \leq j \leq n-1\}$. We denote by $\overline{v}_{(\sigma,i)}$ the vector $(v^0_{(\sigma,i)} \cdots v^{n-1}_{(\sigma,i)})$ of boolean variables. $\overline{v}_{(\sigma,i)}$ represents the state of g_i reached when traversing σ.

We define $V_{g_i} = \bigcup_{\sigma \in S_i} Var(\sigma, i)$. In addition to V_{g_1} and V_{g_2}, we introduce a set V_{aux} of boolean variables which consist of the following variables:

- To guarantee that the LTSs we produce are indeed deterministic, we add a set of boolean variables which are used to enumerate the (non error) states in the DLTSs. For this we use $k \times |\alpha g_1 \cup \alpha g_2|$ vectors of boolean variables, each of size n: For every $1 \leq m \leq k$ and $a \in (\alpha g_1 \cup \alpha g_2)$, we introduce a set of boolean variables $Var(m, a) = \{u^j_{(m,a)} \mid 0 \leq j \leq n-1\}$. We denote by $\overline{u}_{(m,a)}$ the vector $(u^0_{(m,a)} \cdots u^{n-1}_{(m,a)})$ of boolean variables. $\overline{u}_{(m,a)}$ represents the state (of either g_1 or g_2) reached from state m after seeing action a.
- To guarantee that the states of the DLTSs are disjoint, we introduce another vector $\overline{u} = (u^0 \cdots u^{n-1})$ of boolean variables, used to represent the number l such that all states of g_1 are smaller or equal l and all states of g_2 are larger than l.

Variables used for Encoding Membership Constraints. For every disjunctive membership constraint formula $c \in C$ we introduce a boolean "selector" variable en_c that determines which of the disjuncts of *c must* be satisfied (the other disjunct might be satisfied as well). Technically, let $En_c = \{en_c \mid c \in C\}$, and let $A = En_c \cup \{\neg en_c \mid en_c \in En_c\} \cup \{true\}$. We define $\theta^{add}_{g_1}, \theta^{rem}_{g_1} : S_1 \to 2^A$ and $\theta^{add}_{g_2}, \theta^{rem}_{g_2} : S_2 \to 2^A$ such that for every $\sigma \in S_i$, $\theta^{add}_{g_i}(\sigma)$ and $\theta^{rem}_{g_i}(\sigma)$ are the smallest sets such that $true \in \theta^{add}_{g_1}(\epsilon)$ and $true \in \theta^{add}_{g_2}(\epsilon)$, and for every $c \in C$:

- if $c = (-(\sigma{\downarrow_{\alpha g_i}}, i) \vee -(\sigma{\downarrow_{\alpha g_j}}, j))$ then $en_c \in \theta^{rem}_{g_i}(\sigma{\downarrow_{\alpha g_i}})$ and $\neg en_c \in \theta^{rem}_{g_j}(\sigma{\downarrow_{\alpha g_j}})$.
- if $c = +(\sigma{\downarrow_{\alpha g_i}}, i)$ then $true \in \theta^{add}_{g_i}(\sigma{\downarrow_{\alpha g_i}})$.
- if $c = -(\sigma{\downarrow_{\alpha g_i}}, i)$ then $true \in \theta^{rem}_{g_i}(\sigma{\downarrow_{\alpha g_i}})$.
- if $c = (-(\sigma{\downarrow_{\alpha g_j}}, j) \vee +(\sigma a{\downarrow_{\alpha g_i}}, i))$ then $en_c \in \theta^{rem}_{g_j}(\sigma{\downarrow_{\alpha g_j}})$ and $\neg en_c \in \theta^{add}_{g_i}(\sigma a{\downarrow_{\alpha g_i}})$.
- if $c = (-(\sigma{\downarrow_{\alpha g_j}}, j) \vee (+(\sigma a{\downarrow_{\alpha g_i}}, i) \wedge -(\sigma a{\downarrow_{\alpha g_j}}, j)))$ then $en_c \in \theta^{rem}_{g_j}(\sigma{\downarrow_{\alpha g_j}})$, $\neg en_c \in \theta^{add}_{g_i}(\sigma a{\downarrow_{\alpha g_i}})$ and $\neg en_c \in \theta^{rem}_{g_j}(\sigma a{\downarrow_{\alpha g_j}})$.

Intuitively, if at least one of the literals in $\theta^{add}_{g_i}(\sigma)$ is satisfied then σ must be added to the language of g_i, and similarly for $\theta^{rem}_{g_i}(\sigma)$ with removal. These sets are therefore interpreted as disjunctions. Formally, let $Bool(A)$ be the set of boolean formulas over A. For $\theta^{ac}_{g_i} : S_i \to 2^A$ (where $ac \in \{rem, add\}$), we define $\tilde{\theta}^{ac}_{g_i} : S_i \to Bool(A)$ as follows: $\tilde{\theta}^{ac}_{g_i}(\sigma) = \begin{cases} false & \theta^{ac}_{g_i}(\sigma) = \emptyset \\ \bigvee \theta^{ac}_{g_i}(\sigma) & \text{otherwise} \end{cases}$

SAT Constraints. $SatEnc_k(C)$ is a set of constraints (with the meaning of conjunction) over the variables $En_c \cup V_{g_1} \cup V_{g_2} \cup V_{aux}$ defined as follows:

– Encoding the LTSs structures into SAT constraints:
  1. For every trace $\sigma_1 \in S_1$ we add the constraint $\overline{v}_{(\sigma_1,1)} \leq \overline{u}$, and for every trace $\sigma_2 \in S_2$ we add the constraint $\overline{u} < \overline{v}_{(\sigma_2,2)}$ (separating states of the DLTSs). We also add a constraint $\overline{1} \leq \overline{u} \leq \overline{k-1}$ to restrict the range of $\overline{u}$.
  2. For every $\sigma \in S_2$ we add the following constraint $\overline{v}_{(\sigma,2)} \leq \overline{k+1}$ (every trace is mapped to a valid state in the DLTSs).
  3. For every $i \in \{1,2\}$, every trace $\sigma \in S_i$, every action $a \in \alpha g_i$ such that $\sigma a \in S_i$, and for every $1 \leq m \leq k$, we add the following constraint: $(\overline{v}_{(\sigma,i)} = \overline{m} \Rightarrow \overline{v}_{(\sigma a,i)} = \overline{u}_{(m,a)}$ (the DLTSs are deterministic).
  4. For every trace $\sigma \in S_1$ and action $a \in \alpha g_1$, if $\sigma a \in S_1$ then we add the following constraint: $\overline{v}_{(\sigma,1)} = \overline{0} \Rightarrow \overline{v}_{(\sigma a,1)} = \overline{0}$ (the error state of g_1 is a sink state; DLTSs are prefix closed).
  5. For every string $\sigma \in S_2$ and action $a \in \alpha g_2$, if $\sigma a \in S_2$ then we add the following constraint: $\overline{v}_{(\sigma,2)} = \overline{k+1} \Rightarrow \overline{v}_{(\sigma a,2)} = \overline{k+1}$ (the error state of g_2 is a sink state; DLTSs are prefix closed).
– Encoding the membership constraints formulas into SAT constraints:
  6. For every trace $\sigma \in S_1$ we add the constraint: $\tilde{\theta}^{rem}_{g_1}(\sigma) \Rightarrow \overline{v}_{(\sigma,1)} = \overline{0}$.
  7. For every trace $\sigma \in S_2$ we add the constraint: $\tilde{\theta}^{rem}_{g_2}(\sigma) \Rightarrow \overline{v}_{(\sigma,2)} = \overline{k+1}$.
  8. For every trace $\sigma \in S_1$ we add the constraint: $\tilde{\theta}^{add}_{g_1}(\sigma) \Rightarrow \overline{v}_{(\sigma,1)} \neq \overline{0}$.
  9. For every trace $\sigma \in S_2$ we add the constraint: $\tilde{\theta}^{add}_{g_2}(\sigma) \Rightarrow \overline{v}_{(\sigma,2)} \neq \overline{k+1}$.

Note that the implications in constraints 6-9 guarantee that a trace is accepted by g_i (leads to a non-error state) whenever it is required to be added to g_i (as encoded by $\theta^{add}_{g_i}(\sigma{\downarrow_{\alpha g_i}})$). However, it may be accepted also in other cases, provided it is not required to be removed by other constraints. The same holds for removal of traces.

Lemma 8. *$SatEnc_k(C)$ is satisfiable if and only if there exist DLTSs g_1 and g_2 that satisfy C such that $|g_1| + |g_2| = k$.*

Due to Lemma 7 which ensures that (the nondeterministic) LTSs $M_1{\downarrow_{\alpha g_1}}$ and $M_2{\downarrow_{\alpha g_2}}$ satisfy C, we get the following corollary, which ensures termination of GENASSMP:

Corollary 2. *At every iteration of* ACR*, there exists $k \leq O(2^{|M_1|} + 2^{|M_2|})$ where $SatEnc_k(C)$ is satisfiable.*

In fact, since the minimal k is found, minimal assumptions that satisfy C are obtained. In particular, together with Lemma 6, this ensures that when $M_1||M_2 \models P$, then minimal assumptions for which CIRC-AG is applicable are eventually obtained.

From SAT Assignment to LTS Assumptions. Given a satisfying assignment ψ to $SatEnc_k(C)$, we use ψ to generate assumptions g_1 and g_2 that satisfy C.

First, we extract DLTSs $A_1(\psi)$ and $A_2(\psi)$ extended with error states: $A_i(\psi) = (Q_i, \alpha g_i, \delta_i, q^i_0, \pi_i)$ where $Q_i = \{\overline{m} \in \{0,1\}^n \mid \exists \sigma \in S_i$ such that $\psi(\overline{v}_{(\sigma,i)}) = \overline{m}\}$, $q^i_0 = \psi(\overline{v}_{(\epsilon,i)})$, $\pi_1 = \overline{0}$, $\pi_2 = \overline{k+1}$, and $\delta_i(\overline{m}, a) = \overline{m}'$ if there exists $\sigma \in S_i$ such that $\psi(\overline{v}_{(\sigma,i)}) = \overline{m} \wedge \sigma a \in S_i \wedge \psi(\overline{v}_{(\sigma a,i)}) = \overline{m}'$, and otherwise $\delta_i(\overline{m}, a) = \perp$ (undefined).

Note that δ_i is deterministic and it is well defined, since constraint 3 of $SatEnc_k(C)$ ensures that if there exist $\sigma, \sigma' \in S_i$ such that $\psi(\overline{v}_{(\sigma,i)}) = \psi(\overline{v}_{(\sigma',i)})$ and both σa and $\sigma' a$ are in S_i, then also $\psi(\overline{v}_{(\sigma a,i)}) = \psi(\overline{v}_{(\sigma' a,i)})$. Further, by constraint 1, $Q_1 \cap Q_2 = \emptyset$.

$A_1(\psi)$ and $A_2(\psi)$ can be thought of as error LTSs, except that they might be incomplete: δ_i is a partial function. As in an error LTS, traces leading to an error state in $A_i(\psi)$ are rejected. Traces for which δ_i is undefined are unspecified (recall that such traces do not exist in an error LTS, which is complete, and in a DLTS, in contrast, such traces are rejected). The latter represent traces that do not affect the satisfaction of C.

We transform $A_1(\psi)$ and $A_2(\psi)$ into (complete) error LTSs by extending δ_i to total functions. Since unspecified traces do not affect satisfaction of C, any completion results in DLTSs that satisfy C. In practice, if $\delta_i(\overline{m}, a) = \bot$, we define $\delta_i(\overline{m}, a) = \overline{m}$.

To obtain DLTSs, we remove the error states. We denote the result by $LTS(A_i(\psi))$.

Lemma 9. *Let $g_1 = LTS(A_1(\psi))$ and $g_2 = LTS(A_2(\psi))$, where ψ satisfies $SatEnc_k(C)$. Then g_1 and g_2 are DLTSs such that (1) $(g_1, g_2) \models C$ and (2) $|g_1| + |g_2| \leq k$.*

Example 5. Consider the 7th (and final) iteration of ACR. Since the assumptions from the 6th iteration (Figure 3) have a total of 3 states, the search performed by GENASSMP at the 7th iteration starts with $k = 3$, and since $SatEnc_3(C)$ is unsatisfiable, k is increased to 4, yielding (the final) g_1 and g_2 with a total of 4 states (Figure 2). Note that (g_1, g_2) indeed satisfy the membership constraint $-(\langle send \rangle, 1) \vee (+(\langle send, send \rangle, 2) \wedge -(\langle send, send \rangle, 1)) \in C$ from the previous iteration (due to the right disjunct). In particular, they do not exhibit the counterexample from Example 4.

7 Correctness, Termination and Minimality

In this section we argue that our main algorithm ACR is correct, it terminates and produces minimal assumptions.

Theorem 2 (Correctness and Termination). *Given components M_1 and M_2, and property P,* ACR *terminates and returns "$M_1||M_2 \models P$" if P holds on $M_1||M_2$ and "$M_1||M_2 \not\models P$", otherwise.*

Proof (sketch). ACR returns "$M_1||M_2 \models P$" if and only if all premises of CIRC-AG hold, in which case correctness follows from the soundness of CIRC-AG. On the other hand, if ACR returns "$M_1||M_2 \not\models P$", then correctness is ensured by Lemma 4. It remains to prove that ACR terminates. First, Corollary 2 ensures that at every iteration of ACR, $SatEnc_k(C)$ is satisfiable for some $k = O(2^{|M_1|} + 2^{|M_2|})$. Therefore, each iteration terminates. Moreover, by Corollary 1, the pair of DLTSs generated at each iteration is different from all pairs considered in previous iterations, which ensures progress of ACR. Finally, by Lemma 7, $g_1 = M_1{\downarrow_{\alpha g_1}}$ and $g_2 = M_2{\downarrow_{\alpha g_2}}$ always satisfy C. Therefore ACR terminates at the latest when $g_1 = M_1{\downarrow_{\alpha g_1}}$ and $g_2 = M_2{\downarrow_{\alpha g_2}}$, in which case premises 1 and 2 of CIRC-AG necessarily hold and premise 3 amounts to $M_1{\downarrow_{\alpha g_1}}||M_2{\downarrow_{\alpha g_2}} \models P$, hence either all premises hold or a real counterexample is obtained. □

Theorem 3 (Minimality). *If* $M_1||M_2 \models P$ *then* ACR *terminates with DLTSs* g_1 *and* g_2 *whose total number of states is minimal among all pairs of DLTSs that satisfy the CIRC-AG rule.*

Proof (sketch). Termination follows from Theorem 2. Let n be the minimal total number of states of DLTSs that satisfy rule CIRC-AG. By Lemma 6, the corresponding DLTSs satisfy C at any iteration of ACR. Therefore by Lemma 8, $SatEnc_n(C)$ is satisfiable at any iteration and in particular in the last one, where Lemma 9 ensures that the obtained DLTSs $g_1 = LTS(A_1(\psi))$, $g_2 = LTS(A_2(\psi))$ are such that $|g_1| + |g_2| \leq n$. □

8 Evaluation and Concluding Remarks

We implemented ACR in the LTSA (Labelled Transition System Analyser) tool [13]; we use MiniSAT [8] for SAT solving. We optimized our implementation to perform incremental SAT encoding using the ability of MiniSAT to solve CNF formulas under a set of unit clause assumptions. We also made ACR return (at each iteration) k counterexamples for the three premises where, k is $|g_1| + |g_2|$.

We compared ACR with learning-based assume guarantee reasoning (based on rule NonCIRC-AG), on the following examples [19]: *Gas Station* (3 to 5 customers), *Chiron* – a model of a GUI (2 to 5 event handlers), *Client Server* – a client-server application (6 to 9 clients), and a NASA rover model: *MER* (2 to 4 users competing for two common resources). We used the same two-way decompositions reported in previous experiments. Experiments were performed on a MacBook Pro with a 2.3 GHz Intel Core i7 CPU and with 16 GB RAM running OS X 10.9.4 and a Suns JDK version 7.

Table 1 summarizes our results. For both approaches, we report the analysis time (in seconds) and the assumption sizes. Measuring memory is unreliable due to the garbage collection and the interfacing with MiniSAT via native method calls (our measurements indicate that memory consumption is stable and does not increase dramatically for larger cases). We instead report the maximum numbers of states observed for checking the premises of the two rules. We put a limit of 1800 seconds for each experiment; "–" indicates that the time for that case exceeds this limit.

In all the experiments ACR generates smaller assumptions and in the majority of cases this results in smaller analysis time and state space explored. For larger cases the assumptions generated by ACR are *significantly* smaller. For the Gas Station, ACR significantly outperforms learning in terms of analysis time and states explored, while for all other cases the two approaches are comparable, at smaller sizes. However at larger configurations (Client Server 8 and 9, MER 4) ACR again significantly outperforms the learning-based approach. In all but one case (Chiron 5) the smaller assumptions generated with ACR lead to smaller state spaces for checking the rule premises. Case Chiron 5 is still comparable in terms of running time but it may indicate that the two-way decomposition that we used (found to be optimal for learning in previous studies) may not be optimal for ACR. We plan to investigate this further in future work.

Future Work. ACR can be optimized in many ways. Currently we are checking the three premises one after the other at each iteration and get k different counterexamples for each one of them. We can check them in parallel on different machines. We

Table 1. Comparison of ACR (rule CIRC-AG) and learning (rule NonCIRC-AG). Best results are shown in bold.

Case	ACR Time	$\|g_1\|$	$\|g_2\|$	Premise1	Premise2	Premise3	L^* Time	$\|A\|$	Premise1	Premise2
GasSt 3	**26**	**3**	**3**	2588	1093	6	–	>351	>8243	>4045
GasSt 4	**48**	**3**	**3**	19503	2196	4	–	>381	>165836	>47360
GasSt 5	**309**	**3**	**3**	132608	6995	6	–	>207	>560000	>61058
Chiron 2	1.257	**2**	**2**	134	204	5	**0.5**	9	256	198
Chiron 3	**2.013**	**2**	**2**	341	2244	5	2.121	25	492	2736
Chiron 4	**3.149**	**2**	**2**	449	6681	5	6.341	45	860	18370
Chiron 5	34	**2**	**2**	1152	258456	5	**33**	122	2101	138537
ClServ 6	11	**7**	**2**	256	16	10	**8**	256	256	2505
ClServ 7	33	**8**	**2**	576	17	10	33	576	576	6455
ClServ 8	**53**	**9**	**2**	1280	17	9	138	1280	1280	16199
ClServ 9	**249.839**	**10**	**2**	2816	23	14	725	2816	2816	39769
MER 2	**4.397**	**5**	**2**	30	147	6	4.54	46	313	79
MER 3	**35**	**7**	**2**	83	1198	13	50	274	3146	250
MER 4	**1220.649**	**9**	**2**	97	7109	9	–	>1210	>128883	>549

further plan to investigate alphabet refinement and generalization to n-way decompositions (for $n > 2$) – both these techniques significantly enhanced the performance of compositional acyclic techniques [19]. For the n-way decompositions we can either consider a recursive application of our current approach to the system decomposed in two components, each decomposed in two sub-components etc. or a more involved approach that synthesizes directly n assumptions, one for each component. We leave this for future work. We also plan to explore learning and abstraction-refinement for discovering suitable assumptions. Although these techniques might not guarantee minimal assumptions, they can be less computationally demanding than our current approach.

Acknowledgements. This research was partially supported by BSF grant no. 2012259 and NSF grant no. 1329278.

References

1. Alur, R., Henzinger, T.A.: Reactive modules. Formal Methods in System Design 15(1), 7–48 (1999)
2. Alur, R., Madhusudan, P., Nam, W.: Symbolic compositional verification by learning assumptions. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 548–562. Springer, Heidelberg (2005)
3. Gheorghiu Bobaru, M., Păsăreanu, C.S., Giannakopoulou, D.: Automated assume-guarantee reasoning by abstraction refinement. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 135–148. Springer, Heidelberg (2008)
4. Chaki, S., Clarke, E., Sinha, N., Thati, P.: Automated assume-guarantee reasoning for simulation conformance. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 534–547. Springer, Heidelberg (2005)
5. Chen, Y.-F., Clarke, E.M., Farzan, A., Tsai, M.-H., Tsay, Y.-K., Wang, B.-Y.: Automated assume-guarantee reasoning through implicit learning. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 511–526. Springer, Heidelberg (2010)

6. Chen, Y.-F., Farzan, A., Clarke, E.M., Tsay, Y.-K., Wang, B.-Y.: Learning minimal separating DFA's for compositional verification. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 31–45. Springer, Heidelberg (2009)
7. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
8. Een, N., Šorensson, N.: The minisat, http://minisat.se
9. Graf, S., Passerone, R., Quinton, S.: Contract-based reasoning for component systems with rich interactions. In: Sangiovanni-Vincentelli, A., Zeng, H., Di Natale, M., Marwedel, P. (eds.) Embedded Systems Development. Embedded Systems, vol. 20, pp. 139–154. Springer, New York (2014)
10. Gupta, A., McMillan, K.L., Fu, Z.: Automated assumption generation for compositional verification. Formal Methods in System Design 32(3), 285–301 (2008)
11. Henzinger, T.A., Liu, X., Qadeer, S., Rajamani, S.K.: Formal specification and verification of a dataflow processor array. In: ICCAD, pp. 494–499 (1999)
12. Henzinger, T.A., Qadeer, S., Rajamani, S.K.: You assume, we guarantee: Methodology and case studies. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 440–451. Springer, Heidelberg (1998)
13. Magee, J., Kramer, J.: Concurrency: State Models and Java Programs. John Wiley & Sons (1999)
14. McMillan, K.L.: Verification of an implementation of Tomasulo's algorithm by compositional model checking. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 110–121. Springer, Heidelberg (1998)
15. McMillan, K.L.: Circular compositional reasoning about liveness. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 342–346. Springer, Heidelberg (1999)
16. McMillan, K.L.: Verification of infinite state systems by compositional model checking. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 219–237. Springer, Heidelberg (1999)
17. Misra, J., Chandy, K.M.: Proofs of networks of processes. IEEE Trans. Software Eng. 7(4), 417–426 (1981)
18. Namjoshi, K.S., Trefler, R.J.: On the competeness of compositional reasoning. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 139–153. Springer, Heidelberg (2000)
19. Pasareanu, C.S., Giannakopoulou, D., Bobaru, M.G., Cobleigh, J.M., Barringer, H.: Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. Formal Methods in System Design 32(3), 175–205 (2008)
20. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Logics and Models of Concurrent Systems. NATO ASI Series (1985)
21. Rushby, J.: Formal verification of mcmillan's compositional assume-guarantee rule. In: CSL Technical Report, SRI (2001)