Automated Circular Assume-Guarantee Reasoning with N-way Decomposition and Alphabet Refinement

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

[1] Technion – Israel Institute of Technology, Haifa, Israel
skaramt@gmail.com
[2] CMU/NASA Ames Research Center, Mountain View, USA
[3] Tel Aviv University, Tel Aviv, Israel

Abstract. In this work we develop an *automated* circular reasoning framework that is applicable to systems decomposed into *multiple* components. Our framework uses a family of circular assume-guarantee rules for which we give conditions for soundness and completeness. The assumptions used in the rules are initially approximate and their alphabets are automatically refined based on the counterexamples obtained from model checking the rule premises. A key feature of the framework is that the compositional rules that are used change *dynamically* with each iteration of the alphabet refinement, to only use assumptions that are relevant for the current alphabet, resulting in a smaller number of assumptions and smaller state spaces to analyze for each premise. Our preliminary evaluation of the proposed approach shows promising results compared to 2-way and monolithic verification.

1 Introduction

We present an automated assume-guarantee style compositional approach to address the state explosion problem in model checking. Model checking [7] is a well-known technique for automatically checking that software systems satisfy desired properties. Despite its many successes, the technique suffers from the *state explosion problem*, which refers to the worst-case exponential growth of a program's state space with the number of variables and concurrent components. Compositional techniques have shown promise in addressing this problem, by breaking-up the global verification of a system into the local, more manageable, verification of the system's individual components. The environment for each component, consisting of the other components, is replaced by a "small" *assumption*, making each verification task easier. This is often referred to as *assume-guarantee* reasoning [21,24].

Significant progress has been made on automating compositional reasoning using learning and abstraction-refinement techniques [2–6,8,23]. Most of this

© Springer International Publishing Switzerland 2016
S. Chaudhuri and A. Farzan (Eds.): CAV 2016, Part I, LNCS 9779, pp. 329–351, 2016.
DOI: 10.1007/978-3-319-41528-4_18

work has been done in the context of applying a simple compositional assume-guarantee rule, where components of a system are checked using assumptions and properties which are related in an *acyclic* manner.

Another important category of assume-guarantee rules involves *circular reasoning* and uses inductive arguments over time to ensure soundness [1,15,18,19,21]. Circular rules have often been found to be more effective than non-circular rules [13,14,18–20,25]. However, their applicability has been hindered by the manual effort involved in defining assumptions, while automation remained a challenge due to the more involved inductive reasoning and the mutual dependency between assumptions.

Recent work [10] proposed an automated compositional technique for *circular reasoning* which has shown better scalability results compared with an established learning-based method which implements acyclic reasoning [8]. This circular technique is only applicable to a system decomposed into two components (or two subsystems), thus limiting its applicability in practice. Further, the constructed assumptions are defined over the full interface alphabet of the components, thus causing a blowup in the sizes of the assumptions (and the verification tasks) that is in many cases unnecessary.

We propose an automated circular assume-guarantee technique for reasoning about systems decomposed into an arbitrary (but fixed) number of components. We consider a synchronous composition, where components synchronize on the common alphabet (shared actions), and interleave on the remaining actions. We give a *generic* n-component circular rule which can be instantiated into a number of circular rules that can be used for verification. The rule checks that a system composed of M_1, M_2 .. M_n satisfies safety property P based on assumptions g_i ($1 \leq i \leq n$). The first n premises of the rule have the form $M_i \models G_i \triangleright g_i$ and the last premise is $G_n \models P$ (to be formally defined later). The first n premises verify, for each g_i, that it is a correct "guarantee" of M_i, based on a set G_i of "assumptions" representing the guarantees of other components. The last premise ensures that P holds under the assumptions.

We further devise an algorithm that automates the process of building the assumptions g_i based on SAT solving. The algorithm works for any rule that can be derived from the generic rule. The assumption generation algorithm can be viewed as a two layered counterexample-guided search, where the outer layer searches for appropriate assumption alphabets, and the inner layer searches for appropriate assumptions, given a set of alphabets for each of the assumptions.

In the inner layer, the search for assumptions with fixed alphabets is performed similarly to [10]. The algorithm starts with approximate assumptions for all components, and attempts to apply the rule. When the application fails, i.e., at least one of the premises of the rule does not hold, the algorithm uses the counterexamples obtained from checking the failed premises to accumulate *joint* constraints on behaviors (traces) that should or should not be exhibited by the assumptions. These constraints reflect dependencies between assumptions of different components – which is essential for exploiting the circularity of the rule. A SAT solver is then used to determine how to update the assumptions while satisfying all the constraints.

The inner search may return a set of minimal assumptions over the given alphabets that satisfy the rule premises. In this case, we conclude that the property holds and the overall algorithm terminates. Alternatively, the inner search may return a counterexample in the form of a trace of the system over the partial alphabet that violates the property. Since the alphabet is partial, the counterexample is *abstract*: it might turn out to be infeasible (spurious) when considering the full alphabet (which adds synchronization constraints). Feasibility of the abstract counterexample is checked by the outer layer.

The outer layer employs an iterative refinement over the alphabets of the assumptions. It starts with only a subset of the interface alphabet of the components which is sufficient to guarantee soundness of the proof rule and it adds actions to it only as needed. Actions to be added are discovered by analysis of counterexamples obtained from the inner layer. The analysis attempts to extend the counterexample to a trace over the full system alphabet. If the counterexample is successfully extended (indicating a real error in the system), the overall algorithm terminates. Otherwise, new actions are added to the alphabet and a new iteration of the inner search begins. Intuitively these new actions are sufficient for preventing the same counterexample to appear in future iterations. Finally, either a concrete counterexample is found, or the property is verified via a successful application of the rule.

A key feature of our approach is the interplay between the two layers of the search, which manifests itself in two ways. First, the abstraction of the alphabet gives rise to *simplifications* of the assume-guarantee rule. We show that any rule can be simplified based on the inter-dependency induced by the assumption alphabets. Namely, in each premise $M_i \models G_i \triangleright g_i$, assumptions in G_i that contain no common actions with either g_i or with other assumptions in G_i that share actions with g_i can be safely eliminated from G_i, since they do not affect the outcome of checking the premise. The last premise gets simplified as well and the resulting rule becomes easier to check than the original rule. As the alphabets change with each iteration of alphabet refinement, so do the dependencies between assumptions. This leads to different simplifications of the assume-guarantee rule in different iterations. Hence, even though the algorithm searches assumptions for a *fixed* rule, the rules used by the inner search change dynamically between different “outer” iterations.

The second interaction point between the two layers is in the constraints on the assumptions accumulated by the assumption generation algorithm. In each alphabet refinement step, some of these constraints are refined as well according to the new alphabet and are re-used by the inner layer. This makes the search for assumptions *incremental*, resulting in better running time and faster convergence.

Our preliminary evaluation shows that circular compositional reasoning with n-way decomposition, alphabet refinement and dynamic rule simplification can significantly outperform both 2-way compositional and monolithic verification.

Related Work. While the literature on assume-guarantee reasoning is vast, for space reason, we only discuss here some closely related work. Previous work has proposed learning and abstraction-refinement techniques for automating the generation of assumptions [2–6,8,12,23]. This work has been done in the context of applying simple compositional rules, where there is no circular dependency between assumptions and no inductive reasoning involved. Our automatic alphabet refinement uses the heuristics from previous work [11] which were developed for non-circular reasoning. Note however that unlike [11] our approach is *incremental*, i.e. it *re-uses* the results across refinement iterations. Further we employ rule simplification based on the current alphabet which is new. N-way decompositions were handled in e.g. [23] by a recursive application of a non-circular rule. Unlike our approach, that work is sensitive to the order in which the components are analyzed and is not incremental. Our search for minimal assumptions using SAT with an increasing bound is inspired by [12]. However there, a single (separating) assumption is generated while we generate multiple mutually dependent assumptions.

Circular rules have been studied extensively [1,15,18,19,21], however not for full automation as we do here. The approach in [10] addresses assumption generation for 2-way circular reasoning, shown to perform better than non-circular reasoning. We improve on it by providing N-way compositional reasoning and incremental alphabet refinement. An interesting result [22] makes the connection between circular and non-circular rules, using complex (auxiliary) assumptions that use induction over time. Recent work [16] addresses synthesizing circular compositional proofs based on logical abduction. A key difference is that they refer to a decomposition of a sequential program, while we consider concurrent systems. Similar to [16], the approach in [20] also employs a circular compositional approach and uses different abstractions to discharge proof subgoals.

2 Preliminaries

We use Labeled Transition Systems (LTSs) to model the behavior of communicating components in a concurrent system. We briefly present here LTSs and semantics of their operators following the presentation in [10]. Let *Act* be a set of actions and let $\tau \notin Act$ denote a special "local" action.

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

Throughout the paper we use αM to denote the alphabet of an LTS M.

Paths and Traces. A *trace* σ is a sequence of actions, not including τ. We use σ_i to denote the prefix of σ of length i. A *path* in an LTS M is a sequence $p = q_0, a_0, q_1, a_1 \cdots, a_{n-1}, q_n$ of alternating states and 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 of actions along p, obtained by removing from $a_0 \cdots a_{n-1}$ all occurrences of τ. The set of traces obtained from all the paths in M forms the *language* of M, denoted $L(M)$. Note that $L(M)$ is prefix-closed.

An LTS $M = (Q, \alpha M, \delta, q_0)$ is *deterministic* (denoted as DLTS) if it contains no τ transitions and no transitions $(q, a, q'), (q, a, q'') \in \delta$ such that $q' \neq q''$. Any LTS can be converted to a deterministic LTS that recognizes the same language.

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

Parallel Composition. 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 [8]. *For every $t \in (\alpha M_1 \cup \ldots \cup \alpha M_n)^*$, $t \in L(M_1 \| \ldots \| M_n)$ if and only if $t\downarrow_{\alpha M_i} \in L(M_i)$ for every $1 \leq i \leq n$.*

We define the interface alphabet of a component (with respect to the rest of the system) as follows.

Definition 2 (Interface Alphabet). *For $1 \leq i \leq n$, the* interface alphabet of M_i w.r.t. $M_1 \| M_2 \| \cdots \| M_n$, *denoted αJ_i, is: $\alpha J_i = \alpha M_i \cap \left(\bigcup\{\alpha M_j \mid 1 \leq j \leq n, j \neq i\}\right)$.*

Intuitively, the interface alphabet of M_i contains all the actions that are common between M_i and its environment.

Safety Properties. A *safety property* is specified as an LTS P. An LTS M over $\alpha M \supseteq \alpha P$ satisfies P, denoted $M \models P$, if for every $\sigma \in L(M)$, $\sigma\downarrow_{\alpha P} \in L(P)$. A trace $\sigma \in \alpha M^*$ is a *counterexample* for $M \models P$ if $\sigma \in L(M)$ but $\sigma\downarrow_{\alpha P} \notin L(P)$.

To check $M \models P$, the LTS of P is transformed into a deterministic LTS, which is completed with a special "error state" π by adding transitions from every state q in the deterministic LTS to π for all the missing outgoing actions of q; the resulting LTS is called an *error LTS*, denoted by P_{err}. Checking $M \models P$ reduces to checking that π is not reachable in $M \| P_{err}$.

Assume-Guarantee Formulas. In the circular assume-guarantee reasoning paradigm a formula has the form $M \models A \triangleright P$, where M, A and P are LTSs. Intuitively M stands for a component, A for an assumption about M's environment (i.e. the rest of the components in the system), and P for a property that is "guaranteed" by the component.

Definition 3. *Let M, A and P be LTSs such that $\alpha P \subseteq \alpha M$. Then $M \models A \triangleright P$ holds if for every $k \geq 1$ and for every $\sigma \in (\alpha M \cup \alpha A)^*$ of length k such that $\sigma\downarrow_{\alpha M} \in L(M)$, if $\sigma_{k-1}\downarrow_{\alpha A} \in L(A)$ then $\sigma\downarrow_{\alpha P} \in L(P)$.*

Checking $M \models A \triangleright P$ is done by building the LTS $M \| A \| P_{err}$ and labeling its states with (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 labeled by err_a is reachable or not in $M \| A \| P_{err}$ [10].

Notations. For LTSs M and P and a finite nonempty set $G = \{g_1, \ldots, g_k\}$ of LTSs, we use $G \models P$ as a shorthand for $g_1 \| \cdots \| g_n \models P$, and $M \models G \triangleright P$ as a shorthand for $M \models g_1 \| \ldots \| g_k \triangleright P$. Moreover, if $G = \emptyset$, then $M \models G \triangleright P$ denotes $M \models P$. Given a set S of LTSs, we denote by αS their alphabet union, i.e. $\alpha S = \bigcup_{M \in S} \alpha M$.

3 Circular Reasoning with N-way Decomposition

In this section we introduce the family of circular assume-guarantee rules that form the basis of our technique for proving that a system composed of a finite number of components $M_1 \| M_2 \ldots \| M_n$ (n is fixed) satisfies a safety property P. We use a set of n auxiliary properties $G = \{g_1, g_2 \ldots g_n\}$. Components and properties are described by Labeled Transition Systems (LTSs) such that $\alpha P \subseteq \bigcup_{i=1}^{n} \alpha M_i$; further, g_i is the "guarantee" property for M_i, specified as an LTS over alphabet $\alpha g_i \subseteq \alpha M_i$.

Instead of using a particular assume-guarantee rule, we propose to use a generic rule that defines multiple assume-guarantee rules, all following the same general pattern:

$$\begin{array}{ll}
\text{(Premise 1)} & M_1 \models G_1 \triangleright g_1 \\
\text{(Premise 2)} & M_2 \models G_2 \triangleright g_2 \\
\quad\ldots & \\
\text{(Premise n)} & M_n \models G_n \triangleright g_n \\
\text{(Premise n+1)} & G_{n+1} \models P \\
\hline
 & M_1 \| M_2 \| \cdots \| M_n \models P
\end{array}$$

In each premise i, G_i represents the set of left-hand side assumptions to be used in the premise and is defined such that $G_i \subseteq G - \{g_i\}$ for $i < n+1$, i.e. the "guarantees" of the other components are used as "assumptions" when checking M_i. Furthermore $G_{n+1} = G$, i.e. all the assumptions are used in the last premise. There are many rules that can be derived from the above pattern. For example, one rule can be derived by requiring the *maximal* number of assumptions for each premise, i.e. each G_i contains all the assumptions except g_i (we use this rule in our experiments). Another rule can be derived by requiring on the contrary a minimal number of assumptions to be used in each premise. For instance, G_i may contain only the assumptions that share actions with g_i. In fact sets G_1, .. G_n can be chosen arbitrarily (possibly guided by domain knowledge), as long as each G_i is a subset of $G - \{g_i\}$.

As an example, for $n = 2$, the instantiation $G_1 = \{g_2\}, G_2 = \{g_1\}$ and $G_3 = \{g_1, g_2\}$ gives rule **CIRC-AG** from [10].

One can use such rules to check the system in a *compositional* way: instead of checking $M_1 \| M_2 \ldots \| M_n$ directly, which may be too expensive, we can check each premise separately, which is potentially much cheaper. However coming up manually with assumptions g_1, g_2, .. g_n that are sufficient for proving or disproving the property, and at the same time are small enough to enable efficient verification is very difficult. The goal of our work is to derive the assumptions automatically. Furthermore we aim to *simplify* the rule to eliminate unnecessary assumptions and premises, to enable efficient verification.

Soundness and Completeness. We first argue the soundness and completeness of *any* rule that can be obtained from the general rule. Let $\mathrm{CIRC} - \mathrm{AG}_N$ denote a rule derived from the general rule, i.e. we fix the assumption sets G_1, G_2, ..., G_n, G_{n+1} such that $G_i \subseteq G - \{g_i\}$ for $i < n + 1$ and $G_{n+1} = G$.

Similarly to the 2-component rule used in [10] we need additional conditions on the assumption alphabets to ensure soundness and completeness. Soundness is ensured by requiring $\alpha g_i \supseteq \alpha M_i \cap \alpha P$ for every $1 \leq i \leq n$. Completeness is ensured by considering assumptions over alphabets that include the interface alphabets of the components, i.e. we require $\alpha g_i \supseteq (\alpha M_i \cap \alpha P) \cup \alpha J_i$ for every $1 \leq i \leq n$ (we say that this is the *completeness condition*). We use $\alpha_F(g_i)$ to denote $(\alpha M_i \cap \alpha P) \cup \alpha J_i$, i.e. the alphabet *sufficient for completeness.*

Theorem 1. *Rule CIRC-AG$_N$ is sound if* $\alpha g_i \supseteq \alpha M_i \cap \alpha P$ *for every* $1 \leq i \leq n$. *It is complete if* $\alpha g_i \supseteq (\alpha M_i \cap \alpha P) \cup \alpha J_i$ *for every* $1 \leq i \leq n$.

Intuitively, premises $1, \ldots, n$ prove in a compositional and inductive manner that every trace in the language of $M_1 \| M_2 \| \cdots \| M_n$ is also included in the language of $g_1 \| g_2 \| \cdots \| g_n$ while the last premise ensures that every trace in the language of $g_1 \| g_2 \| \cdots \| g_n$ is also in the language of P. Completeness stems from the fact that $M_1, M_2, \ldots, M_n$ (restricted to appropriate alphabets) can be used for $g_1, g_2, \ldots, g_n$ in a successful application of the rule.

Note that we can remove the completeness condition on the alphabets to obtain rules that are sound but not necessarily complete. These rules would still be useful in practice since the alphabet assumptions are smaller, and therefore the assumptions necessary for the proof may be smaller as well and easier to check. Furthermore, this would give us more opportunities to simplify the rules by removing the assumptions that become irrelevant for the proof, due to the smaller alphabets used. This simplification is described in the next section.

4 Alphabet-Based Simplification

At the heart of our automated approach is a method for simplifying the assume-guarantee rules as dictated by the assumption alphabets. Specifically, when we apply our technique, we fix an n-way rule, and an initial alphabet for each assumption. The alphabets induce a simplification of the rule which makes the

Algorithm 1 . Main algorithm for checking $M_1 \| M_2 \| \ldots \| M_n \models P$ using CIRC-AG$_N$

```
1: procedure ACR
2:     Initialize: $\mathcal{A} = \alpha P$, $C = \emptyset$, $k = n$, $IncTr = \emptyset$
3:     $\alpha g_i \triangleq \mathcal{A} \cap \alpha M_i$, $\forall 1 \leq i \leq n$        // Initially $\alpha g_i = \alpha P \cap \alpha M_i$
4:     while (true) do
5:         repeat
6:             $(g_1, g_2, \ldots, g_n) =$ GenAssmp$(C, k, \mathcal{A})$
7:             $(C', Result, IncTr') =$ ApplyAG$(g_1, g_2, \ldots, g_n)$ using S(CIRC-AG$_N$)
8:             $C = C \cup C'$, $k = \sum_{i=1}^{n} |g_i|$, $IncTr = IncTr \cup IncTr'$
9:         until ($Result \neq$ "continue")
10:        if ($Result ==$"false"$(\sigma)$) then        // $\sigma$ is a cex for $M_1 \downarrow_{\alpha g_1} \| \ldots \| M_n \downarrow_{\alpha g_n} \models P$
11:            $(\mathcal{A}, C, k, Result) =$ AlphaRefine$(\sigma, \mathcal{A}, IncTr)$
12:            if ($Result ==$ "false") then return "false"        // else continue to next iteration
13:        else return "true"
```

rule easier to check (premises become simpler, and some premises become redundant). When the alphabets are refined, the simplification of the rule changes accordingly. Hence, essentially, the rule changes dynamically during the compositional verification.

We describe here this rule simplification. Let CIRC-AG$_N$ be an assume-guarantee rule. Suppose that we fix the alphabets over which the assumptions are defined, such that the rule is sound. These alphabets induce a natural simplification of the rule as follows. We define assumptions of G_i that *directly affect* g_i as the assumptions that have common actions with M_i. This provides the basis of an inductive definition, as other assumptions in G_i *indirectly affect* g_i if they have common actions with an assumption in G_i that (directly or indirectly) affects g_i.

The dependency between assumptions can be computed statically, by building, for each premise i, a graph where vertices are the assumptions in $G_i \cup \{g_i\}$, and edges are common actions between them (except that edges of g_i represent common actions with M_i). The assumptions which are not connected to g_i can be safely eliminated from G_i since they cannot influence the outcome of checking the premise.

We also define the set of assumptions from $G_{n+1} = \{g_1, \ldots, g_n\}$ that affect P (directly or indirectly) in a similar way. All the assumptions that do not share actions with P (directly or indirectly) can be eliminated from the last premise. Furthermore the removed assumptions become redundant and their premises are removed altogether.

For a premise i, let $S(G_i)$ denote the set of assumptions used in the premise after simplification. The following lemma states that each simplified premise is equivalent to the original one.

Lemma 2. *$M_i \models S(G_i) \triangleright g_i$ iff $M_i \models G_i \triangleright g_i$ for $i \leq n$. $S(G_{n+1}) \models P$ iff $G_{n+1} \models P$.*

It follows that we can use the simplified rule instead of the original rule, to obtain the same results.

Example 1. Consider the (last) premise of a rule that has the form $g_1||g_2||g_3 \models P$ with $\alpha g_1 = \{a, b\}$, $\alpha g_2 = \{b\}$, $\alpha g_3 = \{c\}$ and $\alpha P = \{a, b\}$. Since assumption g_3 contains neither actions that participate in the parallel composition with g_1 and g_2 nor actions that appear in the property, we can safely remove it resulting in a simplified premise $g_1||g_2 \models P$ which is cheaper to check and results in faster convergence of our algorithms, as explained later in the paper.

If, on the other hand, we consider the same rule with increased alphabets $\alpha g_1 = \{a, b, d\}$ and $\alpha g_3 = \{c, d\}$, then simplification will leave the last premise unchanged.

Thus, for a given rule, and by varying the assumption alphabets, we can obtain a family of sound rules using alphabet-based simplifications. If the alphabets satisfy the completeness condition, the rules are complete as well.

5 Automated Circular Reasoning

In this section we provide an overview of our Automated Circular Reasoning (ACR) algorithm for the compositional verification of a system composed of $M_1, \ldots, M_n$ with respect to property P, where n is fixed. The pseudocode of the algorithm appears in Algorithm 1. ACR can be used with any rule that can be derived from the general pattern described in Sect. 3. Let CIRC-AG$_N$ be such a rule. The assumptions to be used in the rule are derived automatically using a two layered approach which combines *iterative assumption generation* (the inner layer) with *automatic refinement over the assumption alphabets* (the outer layer). The two layers are closely intertwined.

The algorithm maintains in $\mathcal{A}$ the current assumption alphabet, where $\alpha g_i = \mathcal{A} \cap \alpha M_i$, for $1 \leq i \leq n$. In the following, we use $\mathcal{A}$ and $\alpha g_1, \ldots, \alpha g_n$ interchangeably, as the former determines the latter. In addition, the algorithm maintains a set of constraints C on the desired assumptions and a bound k, which is the sum of the number of states in each assumption. The algorithm also maintains a set *IncTr* of *incremental traces*, whose role will become clear in the sequel.

Initially, we allow for each assumption to have a single state, hence $k = n$ for n assumptions. The assumption alphabets over which the assumptions are derived are initialized as follows: $\mathcal{A} = \alpha P$, which means that $\alpha g_i = \alpha P \cap \alpha M_i$ for each $1 \leq i \leq n$. These are the minimal sets needed to guarantee soundness. The sets C and *IncTr* are both initialized to the empty set.

Iterative Assumption Generation. The inner layer of the algorithm builds assumptions over alphabets $\alpha g_1, \ldots, \alpha g_n$ iteratively (in the "repeat ... until" loop, lines 5–9) based on the collected constraints C and the bound k (procedure GenAssmp). For the current assumptions $g_1, \ldots, g_n$, the framework runs model checking to check the premises of the (simplified) assume-guarantee rule (see procedure ApplyAG). The result returned by ApplyAG is either "true", "continue" or "false" (together with a counterexample σ): "true" indicates that all the premises of the rule hold, so ACR finishes returning that the property holds (since the rule is sound); "continue" means that the analysis was inconclusive. Hence the inner loop continues its execution, with a new set of constraints C' that is added to C, resulting in a refinement of the assumptions in the next iteration. The third case ("false") requires further analysis to determine if the counterexample obtained for $M_1{\downarrow_{\alpha g_1}} \| \dots \| M_n{\downarrow_{\alpha g_n}} \models P$ is real or if the assumption alphabets need to be refined. This is explained in more detail below.

Alphabet Refinement. To reduce the complexity of the verification task, we use alphabet refinement over the assumption alphabets. This is the role of the outer layer of our algorithm (in the "while" loop, lines 4–13). Our motivation is that even though completeness of the rule is only guaranteed for the "full" assumption alphabets (i.e. $\alpha_F(g_i)$), there may be smaller alphabets that are enough to prove or disprove the property, and at the same time enable more efficient verification. We aim to discover them iteratively starting with the minimal alphabets that guarantee soundness and only enlarging them as needed.

As long as only subsets of the alphabets are used for the assumptions, counterexamples that are found by the inner loop for $M_1{\downarrow_{\alpha g_1}} \| \dots \| M_n{\downarrow_{\alpha g_n}} \models P$ might not indicate a real error in $M_1 \| \dots \| M_n$, i.e. the counterexamples may be "spurious". Procedure AlphaRefine performs a counterexample analysis to determine if a counterexample is real or not. In the former case, ACR terminates and reports an error. In the latter case AlphaRefine uses heuristics to add actions to $\mathcal{A}$ (and accordingly to αg_i) that are guaranteed to avoid producing the same counterexample in subsequent iterations. AlphaRefine returns "continue" with a new alphabet $\mathcal{A}$, and ACR executes a new iteration.

A key novelty of our approach is that the assume-guarantee rules change dynamically during alphabet refinement. With each update to $\mathcal{A}$, rule CIRC-AG$_N$ is simplified according to the procedure described in Sect. 4 and procedure ApplyAG applies the simplified rule (denoted S(CIRC-AG$_N$)). As the algorithm progresses, the assumption alphabets change resulting in new simplifications applied to the rule.

When alphabet refinement takes place, we need to restart the construction of assumptions (in the inner loop) with a new set of assumptions that are built over the new alphabets. We have optimized this step by re-using some of the constraints from the previous iteration. The set $IncTr$ keeps track of the traces used to derive the constraints. The refined constraints are used to initialize the g_i's in the new iteration. Our *incremental* approach avoids getting and re-analyzing traces that were already removed at previous iterations. Thus, it reduces the number of iterations and improves runtime.

Theorem 2. *(Correctness and Termination) The framework implemented by* ACR *terminates and returns true if* $M_1 \| M_2 .. \| M_n \models P$ *and false otherwise.*

Partial correctness holds since ACR returns true if and only if all premises of CIRC-AG$_N$ hold, in which case correctness follows from the soundness of the rule. Further, if ACR returns false, then this corresponds to σ being a counterexample in $M_1{\downarrow_{\alpha_F(g_1)}} \| .. M_n{\downarrow_{\alpha_F(g_n)}}$ (see AlphaRefine) which corresponds to a real counterexample (Lemma 7).

As for termination, ACR executes a new iteration of the inner loop if the assumptions need to be refined (but the alphabet stays the same). For a given $\mathcal{A}$, there can only be finitely many iterations before "true" or "false" is obtained from ApplyAG. This is shown similarly to [10]. A "true" result makes ACR terminate, while a "false" result might lead to an alphabet refinement step (if the counterexample does not extend to all actions). The result of refinement is that more interface actions are added to $\mathcal{A}$, and a new iteration of the outer loop is executed. In the worst case, all interface actions are added (if no conclusive reply is obtained before) in which case no more spurious counterexamples can be obtained and the algorithm is guaranteed to terminate.

6 Iterative Construction of Assumptions over a Given Alphabet

In this section we describe in detail the inner layer of ACR, which searches for assumptions over a given alphabet. As we explain next, this can be understood as a search for assumptions for an *abstract system*, where the abstraction is defined by the alphabet.

Recall that $\mathcal{A} = \alpha g_1 \cup .. \cup \alpha g_n$. The assumption alphabets induce a natural abstraction over the system by projecting each component M_i to the alphabet αg_i, i.e. $M_1{\downarrow_{\alpha g_1}} \| M_2{\downarrow_{\alpha g_2}} \| .. M_n{\downarrow_{\alpha g_n}}$. This is an abstraction since $M_i{\downarrow_{\alpha g_i}} = M_i{\downarrow_{\mathcal{A}}}$ for every $1 \leq i \leq n$, and $L(M_1 \| M_2 \| .. M_n){\downarrow_{\mathcal{A}}} \subseteq L(M_1{\downarrow_{\mathcal{A}}} \| M_2{\downarrow_{\mathcal{A}}} \| .. M_n{\downarrow_{\mathcal{A}}})$ [8].

Further, note that since $\alpha G_i, \alpha g_i \subseteq \mathcal{A}$, premises of the form $M_i \models G_i \triangleright g_i$ are equivalent to $M_i{\downarrow_{\mathcal{A}}} \models G_i \triangleright g_i$. Intuitively this means that applying CIRC-AG$_N$ with the alphabets restricted to $\mathcal{A}$ can be interpreted as a compositional analysis of the abstracted system $M_1{\downarrow_{\alpha g_1}} \| M_2{\downarrow_{\alpha g_2}} \| .. M_n{\downarrow_{\alpha g_n}}$, which may be smaller and therefore easier to analyze (i.e. may require smaller assumptions to be used in the rule). Furthermore note that the rule is complete for this abstraction, since the alphabets of the abstract components $M_i{\downarrow_{\alpha g_i}}$ are equal to the assumption alphabets, ensuring that the alphabets satisfy the completeness condition in the abstraction.

6.1 Assumption Generation in GenAssmp

Given a set of constraints C, a lower bound k on the total number of states in $\sum_{i=1}^{n} |g_i|$, and an alphabet sequence $\mathcal{A}$, GenAssmp computes assumptions $g_1, g_2, \cdots, g_n$ over $\mathcal{A}$ that satisfy C. Assumptions are built as deterministic LTSs. The implementation of GenAssmp is a natural generalization of previous work [10] where it was used to generate two assumptions. Roughly speaking, for each value of k starting from the given k, GenAssmp creates a SAT instance $SatEnc_k(C)$ that encodes the structure of the desired DLTSs $g_1, g_2, \cdots, g_n$ (e.g., deterministic and prefix closed) with $\sum_{i=1}^{n} |g_i| \leq k$, as well as the requirement that they satisfy the constraints in C. GenAssmp then searches for a satisfying assignment and transforms the satisfying assignment into DLTSs $g_1, g_2, \cdots, g_n$ that satisfy all the constraints in C. The bound k is increased only when $SatEnc_k(C)$ is unsatisfiable, hence minimal DLTSs that satisfy C are obtained.

The key difference in our encoding compared to [10] is the need to handle disjunctive constraints with up to n disjuncts (as opposed to 2 in [10]). While in [10] each disjunctive constraint with 2 disjuncts is handled with a single "selector" variable, we use $\log n$ selector variables to encode a disjunctive constraint with n disjuncts.

6.2 ApplyAG Algorithm

Given assumptions g_1, g_2, …, g_n, ApplyAG (see Algorithm 2) applies assume-guarantee reasoning with the current circular rule simplified under the current alphabets. This is done by model checking all premises of the rule (the order does not matter).[1] If all the premises are satisfied, then, since the rule is sound, it follows that $M_1 \| M_2 \| \cdots \| M_n \models P$ holds (and the result "true" is returned to the user). Otherwise, at least one of the premises does not hold and a counterexample trace is found.

ApplyAG performs an analysis of the counterexample trace as described below. The counterexample analysis is performed with respect to the projections $M_i{\downarrow_{\alpha g_i}}$.

The counterexample analysis may conclude that "$M_1{\downarrow_{\alpha g_1}} \| \ldots \| M_n{\downarrow_{\alpha g_n}} \not\models P$", indicating an error in the abstract system induced by the current alphabet, in which case "false" is returned. Recall, however, that due to the abstraction this is not necessarily an error in $M_1 \| \ldots \| M_n$. If the analyzed trace does not correspond to an error in the abstract system, we conclude that the counterexample is a result of imprecise assumptions. We then compute a set of new constraints C on the assumptions in order to avoid getting the same counterexample in subsequent iterations and return "continue".

Similar to [10] we use constraints to gather information about traces over the current alphabet that need or need not be in the languages of the assumptions. The constraints are of the form: $+(\sigma, i)$ – meaning that σ should be in $L(g_i)$, $-(\sigma, i)$ – meaning that σ should not be in $L(g_i)$, or boolean combinations of them, where $\sigma \in \alpha g_i{}^*$.

The check for an error in the abstract system or for new constraints is performed by ApplyAG directly (if last premise failed) or by UpdateConstraint (if other premises failed). Essentially the counterexample indicates an error if it corresponds to a trace in each $M_i{\downarrow_{\alpha g_i}}$ and furthermore it violates the property. Since all assumptions whose alphabet affects P (directly or indirectly) are in $S(G_{n+1})$, it suffices to check membership in $M_i{\downarrow_{\alpha g_i}}$ for every $g_i \in S(G_{n+1})$ when searching for an error. The formal justification is provided by the following lemma:

Lemma 3. *Let $S(G_{n+1}) = \{g_{i_1}, \ldots, g_{i_k}\}$. Then $M_1{\downarrow_{\alpha g_1}} \| \cdots \| M_n{\downarrow_{\alpha g_n}} \models P$ if and only if $M_{i_i}{\downarrow_{\alpha g_{i_1}}} \| \cdots \| M_{i_k}{\downarrow_{\alpha g_{i_k}}} \models P$.*

[1] The check of the premise i is performed over the full alphabet of M_i in order to maintain the set *IncTr* which enables our incremental approach for alphabet refinement, as explained in Sect. 7. In addition, it helps in detecting errors.

Algorithm 2 . Applying S(CIRC-AG$_N$) with $g_1, g_2, \ldots g_n$, and constraint updating.

```
1: procedure ApplyAG($g_1, g_2, \ldots, g_n$)
2:     for $i \in I_A$ do
3:         if $M_i \not\models S(G_i) \triangleright g_i$ then
4:             Let $\sigma_i a_i$ be a counterexample for $M_i \not\models S(G_i) \triangleright g_i$
5:             return UpdateConstraints($i, \sigma_i a_i$)
6:     if $S(G_{n+1}) \not\models P$ then
7:         Let $\sigma$ be a counterexample
8:         if ( $\bigwedge_{g_i \in S(G_{n+1})} (\sigma{\downarrow}_{\alpha g_i} \in L(M_i{\downarrow}_{\alpha g_i}))$ ) then
9:             return ($\emptyset$, "false", $\emptyset$)      // "$M_1{\downarrow}_{\alpha g_1} \| M_2{\downarrow}_{\alpha g_2} \| \ldots \| M_n{\downarrow}_{\alpha g_n} \not\models P$"
10:        else       // Remove $\sigma$ from one of $g_j \in S(G_{n+1})$
11:            $C = \{ \bigvee_{g_i \in S(G_{n+1})} -(\sigma{\downarrow}_{\alpha g_i}, i) \}$
12:            return ($C$, "continue", $\emptyset$)
13:    return ($\emptyset$, "true", $\emptyset$)      // "$M_1 \| M_2 \| \cdots \| M_n \models P$"
```

For a counterexample obtained for the last premise the checks are done at line 8 in ApplyAG. If one of these checks fails, a new constraint is added to make sure that the same trace will not be in $g_1 \| \cdots \| g_n$ in the next iterations (see line 11 in ApplyAg). However a similar check in UpdateConstraint is more involved, and is described in the next subsection.

6.3 Assumption Refinement in UpdateConstraints

UpdateConstraint (Algorithm 3) gets a counterexample σa for one of the inductive premises i where $1 \leq i \leq n$ and checks whether the trace corresponds to an error (in the abstract system). If it does, then "false" is returned. Otherwise, the counterexample analysis continues in order to decide which constraint(s) need to be added to the set of constraints C in order to refine the assumptions and avoid getting the same counterexample in subsequent iterations.

Trace Extension. The counterexample of premise i is over the alphabet $\alpha M_i \cup \alpha S(G_i)$. However, in order to determine whether the trace corresponds to an error and if not to determine which constraints to add, UpdateConstraint needs to check membership of (projections) of this trace in other components as well as in P. The first step taken by UpdateConstraints therefore calls ExtendTrace to extend the counterexample trace to a trace over the alphabet $\mathcal{A}$ such that its projection to $\alpha M_i \cup \alpha S(G_i)$ remains unchanged. The algorithm works correctly with any such extension, including the one that keeps the trace unchanged. However, more sophisticated extension schemes can contribute to a faster convergence of the algorithm.

Specifically, our implementation of ExtendTrace, presented in Algorithm 4, employs a greedy extension algorithm that considers the LTSs whose alphabet is (potentially) uncovered in an arbitrary order, and iteratively extends the trace by simulating it on these LTSs one by one in that order. Whenever the simulation

Algorithm 3. Computation of constraints based on a counterexample for $M_i \models S(G_i) \triangleright g_i$. We use $\sigma{\downarrow} \in L(B)$ as a shorthand for $\sigma{\downarrow}_{\alpha B} \in L(B)$.

```
1: // $\sigma a$ is a counterexample for $M_i \models S(G_i) \triangleright g_i$, i.e. $\sigma a{\downarrow} \in L(M_i), \sigma{\downarrow} \in L(g_j)$ for every
   $g_j \in S(G_i)$, $\sigma a{\downarrow} \notin L(g_i)$.
2: procedure UPDATECONSTRAINTS($i, \sigma a$)
3:     $\sigma a = \text{EXTENDTRACE}(\sigma a, (\alpha S(G_i) \cup \alpha M_i), \{M_j{\downarrow}_{\alpha g_j} \mid g_j \notin (S(G_i) \cup \{g_i\})\} \cup \{P\})$
4:     if ( $\bigwedge_{g_j \in S(G_{n+1}), j \neq i} (\sigma a{\downarrow} \in L(M_j{\downarrow}_{\alpha g_j}))$ and $\sigma a{\downarrow} \notin L(P)$) then
5:         // $\sigma a{\downarrow} \in L(M_1{\downarrow}_{\alpha g_1} \| \cdots \| M_n{\downarrow}_{\alpha g_n})$ and $\sigma a{\downarrow} \notin L(P)$
6:         return ($\emptyset$, “false”,$\emptyset$)    // “$M_1{\downarrow}_{\alpha g_1} \| M_2{\downarrow}_{\alpha g_2} \| \ldots \| M_n{\downarrow}_{\alpha g_n} \not\models P$”
7:     // Optimized constraints
8:     for each $G \in T(g_i)$ do    // $G$ is a closed set
9:         if ( $\bigwedge_{g_j \in G, j \neq i} (\sigma a{\downarrow} \in L(M_j{\downarrow}_{\alpha g_j}))$ then
10:            // Add $\sigma a$ to all assumptions in $G$ based on Lemma 4(1).
11:            // Adding $\sigma a$ to $g_i \in G$ prevents getting $\sigma a$ as a cex to premise $i$ in the future.
12:            $C = \bigcup_{g_j \in G} \{+(\sigma a{\downarrow}_{\alpha g_j}, j)\}$
13:            return ($C$, “continue”,$\emptyset$)
14:        if ( $\bigwedge_{g_j \in G, j \neq i} (\sigma{\downarrow} \in L(M_j{\downarrow}_{\alpha g_j}))$ then
15:            // Add $\sigma$ to all assumptions in $G$ based on Lemma 4(1).
16:            // Since $S(G_i) \subseteq G$, we cannot remove $\sigma$ from $S(G_i)$, hence add $\sigma a$ to $g_i$.
17:            $C = \bigcup_{g_j \in G} \{+(\sigma{\downarrow}_{\alpha g_j}, j)\} \cup \{+(\sigma a{\downarrow}_{\alpha g_i}, i)\}$
18:            return ($C$, “continue”,$\emptyset$)
19:    if $\sigma a{\downarrow} \notin L(P)$ and $\sigma{\downarrow} \in L(P)$ then
20:        // Remove $\sigma$ from $S(G_i)$ or (add $\sigma a$ to $g_i$ and remove it from $S(G_{n+1}) \setminus \{g_i\}$).
21:        // In the latter case, the removal of $\sigma a$ from $S(G_{n+1}) \setminus \{g_i\}$ is due to Lemma 4(2).
22:        $C = \{(\bigvee_{g_j \in S(G_i)} (-(\sigma{\downarrow}_{\alpha g_j}, j)))\vee$
23:                $(+(\sigma a{\downarrow}_{\alpha g_i}, i) \wedge (\bigvee_{g_j \in S(G_{n+1}) \setminus \{g_i\}} (-(\sigma a{\downarrow}_{\alpha g_j}, j)))))\}$
24:        $IncTr = \{(\sigma a, i, 22)\}$
25:        return ($C$, “continue”,$IncTr$)
26:    if $\sigma a{\downarrow} \notin L(P)$ and $\sigma{\downarrow} \notin L(P)$ then
27:        // Removal of $\sigma$ from $S(G_{n+1}) \setminus (S(G_i) \cup \{g_i\})$ in line 30 is due to Lemma 4(2).
28:        // Since LTSs are prefix-closed, the latter implies removal of $\sigma a$ from $S(G_{n+1}) \setminus \{g_i\}$.
29:        $C = \{(\bigvee_{g_j \in S(G_i)} (-(\sigma{\downarrow}_{\alpha g_j}, j)))\vee$
30:                $(+(\sigma a{\downarrow}_{\alpha g_i}, i) \wedge (\bigvee_{g_j \in S(G_{n+1}) \setminus (S(G_i) \cup \{g_i\})} (-(\sigma{\downarrow}_{\alpha g_j}, j)))))\}$
31:        $IncTr = \{(\sigma a, i, 29)\}$
32:        return ($C$, “continue”,$IncTr$)
33:    // Default constraint
34:    $C = \{(\bigvee_{g_j \in S(G_i)} (-(\sigma{\downarrow}_{\alpha g_j}, j)) \vee +(\sigma a{\downarrow}_{\alpha g_i}, i))\}$
35:    $IncTr = \{(\sigma a, i, 34)\}$
36:    return ($C$, “continue”,$IncTr$)
```

Algorithm 4. Get trace σ over alphabet Σ and extend it to be over the alphabet of all assumptions

```
1: procedure ExtendTrace($\sigma, \Sigma, N_1, N_2, \cdots N_k$)
2:     for $i = 1, \ldots, k$ do
3:         if $(L(LTS(\sigma) \| N_i) \neq \emptyset)$ then
4:             $\sigma$ = trace in $LTS(\sigma) \| N_i$
5:             $\Sigma = \Sigma \cup \alpha N_i$
6:     return $\sigma$ over the alphabet $\Sigma \cup \bigcup_{i=1}^{n} \alpha N_i$
```

succeeds, the trace and its alphabet are extended accordingly. When it fails, the trace remains unchanged. Upon termination, the alphabet of the trace is extended to include the full alphabet, even if the simulation on some of the LTSs failed. The distinguishing feature of Algorithm 4 compared to other extensions (e.g., random extensions) is the fact that it tries to find an extension of the trace that is in the language of the LTSs that it gets as input. It therefore increases the chances of successful checks in UpdateConstraints.

The analysis and computation of constraints are performed on the extended trace.

Default Constraints. If the (extended) counterexample trace corresponds to an error in the abstract system (line 4), then UpdateConstraints returns "false". Otherwise, it computes a new set of constraints. The added constraints are a crucial ingredient as they guide the search for assumptions. They should be strong enough to eliminate the already seen counterexamples and allow progress and convergence of the algorithm, but should not over-constrain the assumptions, in order not to exclude viable assumptions.

Recall that the ith inductive premise in the simplified rule is of the form $M_i \models S(G_i) \triangleright g_i$, and a counterexample for it is a trace σa such that $\sigma a{\downarrow} \in L(M_i), \sigma{\downarrow} \in L(g_j)$ for every $g_j \in S(G_i)$ and $\sigma a{\downarrow} \notin L(g_i)$. The default constraint to eliminate such a counterexample σa is the constraint $+(\sigma a{\downarrow}_{\alpha g_i}, i) \vee \bigvee_{j \in S(G_i)} -(\sigma{\downarrow}_{\alpha g_j}, j)$ stating that the counterexample should be added to g_i or its prefix should be removed from $S(G_i)$ (i.e., from at least one of the assumptions in $S(G_i)$). Such a constraint is added in line 34. This specific constraint is also *incremental*, and the corresponding trace is therefore also added to $IncTr$, along with an identifier of the premise by which it was added and the line number in which the constraint was computed, in order to allow its re-use after alphabet refinement.

Optimized Constraints. A key aspect in the assumptions refinement is our ability to add stronger constraints, without over-constraining the assumptions. This helps the overall algorithm converge faster since a stronger constraint removes more irrelevant assumptions from the assumption space at once. Furthermore, stronger constraints are easier for GenAssmp to solve, thus the overall run-time of the algorithm is reduced.

We come up with several properties of useful assumptions (i.e., assumptions that can be successfully used in the rule) which enable the addition of stronger constraints in several cases. These properties are nontrivial extensions of the properties observed in [10] for the 2-component case.

For example, from the simplified rule we extract *closed sets* of assumptions. A set G of assumptions is *closed* if $g_i \in G$ implies that $S(G_i) \subseteq G$. We show that:

Lemma 4. *Let $g_1, \ldots, g_n$ be LTS assumptions over alphabets $\alpha g_1, \ldots, \alpha g_n$ successfully used in CIRC-AG$_N$, and let $S(\cdot)$ denote the simplification of CIRC-AG$_N$ with respect to the alphabets $\alpha g_1, \ldots, \alpha g_n$. Then*

1. *if $\{g_{i_1}, ..g_{i_m}\}$ is a closed set, then $M_{i_1} \| M_{i_2} \| \cdots \| M_{i_m} \models g_{i_1} \| g_{i_2} \| \cdots \| g_{i_m}$, and*
2. *forall $1 \leq i \leq n$, if $\{g_{i_1}, ..g_{i_m}\} = S(G_{n+1}) \setminus \{g_i\}$, then $M_i \| g_{i_1} \| g_{i_2} \cdots \| g_{i_m} \models P$.*

We carefully select closed sets and scenarios in which it is beneficial to use observation (1) for generation of constraints. Namely, we consider closed sets which are the *closures* of some assumption, defined as follows.

Definition 4 (Closure). *The* closure *of g_k, denoted $Cl(g_k)$, is the smallest set of assumptions G such that $S(G_k) \subseteq G$ and for every $g_j \in G$ it holds that $S(G_j) \subseteq G$ as well.*

The oset $Cl(g_k)$ includes all the assumptions in premise k (after simplification), and for each of them includes all the assumptions in their premises etc. Note that $Cl(g_k)$ is defined based on the simplified rule, and is a closed set. For every assumption g_i, we define the set of closures that it is part of, denoted $T(g_i)$:

Definition 5. *For every $1 \leq i \leq n$, $T(g_i) = \{Cl(g_k) \mid g_i \in Cl(g_k), 1 \leq k \leq n\}$.*

When UpdateConstraints (Algorithm 3) analyzes a counterexample σa for premise i of (the simplified) CIRC-AG$_N$, it considers all the closures $G = \{g_{i_1}, ..g_{i_m}\}$ in $T(g_i)$ and checks whether $\sigma a\downarrow_{\alpha g_j} \in L(M_j\downarrow_{\alpha g_j})$ for every $g_j \in G$. If so, we add a constraint $+(\sigma a\downarrow_{\alpha g_j}, j)$ for every j such that $g_j \in G$ (see line 12) in order to ensure that (the projection of) σa is in $g_{i_1} \| \cdots \| g_{i_m}$, as follows from Lemma 4(1). Since $g_i \in G$, the added constraints imply $+(\sigma a\downarrow_{\alpha g_i}, i) \vee \bigvee_{j \in S(G_i)} -(\sigma\downarrow_{\alpha g_j}, j)$, thus they suffice to eliminate the counterexample and avoid the need for a disjunctive constraint.

Similar reasoning is performed in line 17 using σ. However, in this case, the added constraints $+(\sigma\downarrow_{\alpha g_j}, j)$ for every j such that $g_j \in G$ refer to σ and do *not* imply $+(\sigma a\downarrow_{\alpha g_i}, i) \vee \bigvee_{j \in S(G_i)} -(\sigma\downarrow_{\alpha g_j}, j)$. Still, the fact that $g_i \in G$ and G is a closed set, ensures that $S(G_i) \subseteq G$, and hence $\bigvee_{j \in S(G_i)} -(\sigma\downarrow_{\alpha g_j}, j)$ cannot hold. Therefore, the disjunctive constraint is strengthened into $+(\sigma a\downarrow_{\alpha g_i}, i)$, again avoiding the disjunction (see line 17).

Similarly, we use observation (2) to strengthen the $+(\sigma a{\downarrow_{\alpha g_i}}, i)$ disjunct of the default constraint by adding specialized constraints of the form $\bigvee_{g_j \in S(G_{n+1})\setminus\{g_i\}}(-(\sigma a{\downarrow_{\alpha g_j}}, j))$ in the case where $\sigma a{\downarrow_{\alpha P}} \notin L(P)$ (line 22), with an additional strengthening in line 29 for the case where also $\sigma{\downarrow_{\alpha P}} \notin L(P)$. These specialized constraints are also incremental, hence the corresponding traces are added to $IncTr$.

Progress and Termination of Assumption Refinement. The assumption refinement continues until the assumptions satisfy all premises of the rule, or an error is found (in the abstract system). The progress of the assumption refinement is guaranteed by the following lemmas.

Lemma 5. *Let σ be a counterexample of premise i of CIRC-AG$_N$ and let C be the updated set of constraints. Then any LTSs $g'_1, g'_2, \cdots, g'_n$ that satisfy the set of constraints C will no longer exhibit σ as a counterexample for premise i of CIRC-AG$_N$.*

We conclude that any sequence of LTSs $g'_1, g'_2, \ldots, g'_n$ that satisfies C is different from every previous sequence of LTSs considered by the algorithm.

The following lemma states that the added constraints do not over-constrain the assumptions. It ensures that the "desired" assumptions that enable to verify (1) or falsify (2) the property are always within reach.

Lemma 6. *Let $g_1, \ldots, g_n$ be LTSs over $\alpha g_1, \ldots, \alpha g_n$ s.t. one of the following holds:*

1. $g_1, \ldots, g_n$ satisfy all premises of CIRC-AG$_N$, or
2. $g_i = M_i{\downarrow_{\alpha g_i}}$ for every $1 \leq i \leq n$.

Then $(g_1, \ldots, g_n)$ satisfy every set of constraints C produced by ACR.

Due to the above lemmas, along with the completeness of the rule with respect to the abstraction $M_1{\downarrow_{\alpha g_1}} \| M_2{\downarrow_{\alpha g_2}} \| .. M_n{\downarrow_{\alpha g_n}}$, the iterative construction of the assumptions over $\mathcal{A}$ (lines 5–9 of Algorithm 1) is guaranteed to terminate returning either minimal assumptions over $\mathcal{A}$ that satisfy the rule premises or a counterexample for the abstract system. This is shown similarly to [10].

7 Alphabet Refinement

This section describes the outer layer of ACR, which iteratively searches for an appropriate alphabet for the assumptions. Each iteration defines a different alphabet $\mathcal{A}$ which restricts the alphabet of the assumptions. Initially $\mathcal{A} = \alpha P$, and therefore $\alpha g_i = \alpha P \cap \alpha M_i$. As long as $\mathcal{A}$ is a strict subset of $\alpha P \cup \bigcup_{i=1}^{n} \alpha J_i$ (which means that αg_i is a strict subset of $(\alpha M_i \cap \alpha P) \cup \alpha J_i$), completeness is not guaranteed with respect to $M_1 \| \cdots \| M_n$. This also means that a counterexample obtained by the inner layer might be spurious. Hence, when an abstract

Algorithm 5. Alphabet Refinement

```
 1: procedure AlphaRefine($\sigma, \mathcal{A}, IncTr$)
 2:     if ($\alpha g_i = \alpha_F(g_i)$ for every $1 \leq i \leq n$) then
 3:         return (-,-,-,"false")
 4:     for $1 \leq i \leq n$ do
 5:         let $\sigma_i$ be a trace in $L(LTS(\sigma\downarrow_{\alpha g_i}) \| M_i\downarrow_{\alpha_F(g_i)})$
 6:     if (Match($\sigma_1, \sigma_2, \cdots, \sigma_n$)) then
 7:         return (-,-,-,"false")
 8:     // IncAlpha decides which new interface letters to add based on heuristics from 11.
 9:     $\mathcal{A} = \mathcal{A} \cup$ IncAlpha($\sigma_1, \sigma_2, \cdots, \sigma_n$)
10:     $k = n$, $C = \emptyset$
11:     for each ($\sigma, i, type$) $\in IncTr$ do // Update constraints based on incremental traces
12:         $C = C \cup RC(\sigma, i, type, \mathcal{A})$
13:     return ($\mathcal{A}, C, k$, "continue")
```

counterexample for $M_1\downarrow_{\alpha g_1} \| \cdots \| M_n\downarrow_{\alpha g_n} \models P$ is obtained, AlphaRefine (Algorithm 5) is called to check if the counterexample can be extended to a real counterexample. If it can not, AlphaRefine performs automatic alphabet refinement using heuristics similar to previous work [11] developed for non-circular assume-guarantee reasoning. Note however that a key difference from previous work is that our alphabet refinement enables dynamic simplification of the rule used for verification. Furthermore we improve upon [11] by providing a procedure for *re-using* the results across refinement iterations.

In essence, a counterexample σ is real if $\sigma\downarrow_{\alpha_F(g_i)} \in L(M_i\downarrow_{\alpha_F(g_i)})$ and $\sigma\downarrow_{\alpha P} \notin L(P)$. This is stated by the following lemma, extending the 2-component case [10].

Lemma 7. *If $\sigma\downarrow_{\alpha_F(g_i)} \in L(M_i\downarrow_{\alpha_F(g_i)})$ (for $i = 1..n$) and $\sigma\downarrow_{\alpha P} \notin L(P)$ then $M_1 \| M_2 \| ..M_n \not\models P$. Moreover, σ can be extended into a full counterexample for $M_1 \| M_2 \| ..M_n \models P$.*

AlphaRefine first checks if $\alpha g_i = \alpha_F(g_i)$, where $\alpha_F(g_i)$ is the alphabet sufficient for completeness. If this is not the case, and also if we do not manage to extend the counterexample to this alphabet, AlphaRefine chooses heuristically new interface actions to be added to the alphabet $\mathcal{A}$ (and to αg_i accordingly). The heuristic uses backward refinement shown to work well in previous studies. The counterexample σ is projected on all the components one by one with the full alphabet of completeness. We then perform a backward analysis for every two traces: the traces are scanned backward, from the end of each trace to the beginning looking for the first action where the two traces disagree. The alphabet $\mathcal{A}$ is refined by adding all these actions. The refined alphabet is used in the next iteration of ACR. Procedure Match simply checks that all counterexamples agree on common alphabets.

Once the alphabet changes, the set of constraints C maintained by the algorithm is no longer suitable and has to be emptied. A novel aspect of our approach is that we identify certain constraints that can be refined and moved to the new iteration (as described below).

7.1 Incremental Alphabet Refinement

Recall that constraints are computed based on counterexamples to premises of the form $M_i \models S(G_i) \triangleright g_i$. These are traces over $\alpha M_i \cup \alpha S(G_i)$. While $\alpha S(G_i)$ changes as the alphabet increases, αM_i does not. A naive incremental approach would therefore keep all these traces, and would regenerate constraints based on them by the same counterexample analysis, but with the refined alphabet of the assumptions. However, our goal is to avoid the overhead in analyzing the counterexamples again.

Ideally, we would like to simply derive the same constraints by projecting the counterexample traces on the new alphabet without any further checks. However, this might introduce incorrect constraints that would over-constrain the assumptions. The reason is that the correctness of an existing constraint relies on checks such as $\sigma a{\downarrow_{\alpha g_j}} \in L(M_j{\downarrow_{\alpha g_j}})$ performed with respect to the previous alphabet. The same checks might return different outcomes when conducted with the refined alphabet, in which case the correctness of the (refined) constraint is not guaranteed.

The key challenge to address when trying to re-use constraints is therefore to make sure that the same checks are valid after alphabet refinement. To that end, we identify a subset of the constraints for which this is the case. These are constraints whose correctness relies on checks over $\sigma{\downarrow_{\alpha P}}$, and checks such as $\sigma{\downarrow_{\alpha g_j}} \notin L(M_j{\downarrow_{\alpha g_j}})$, but no checks such as $\sigma{\downarrow_{\alpha g_j}} \in L(M_j{\downarrow_{\alpha g_j}})$. The justification for the re-use of such constraints stems from (i) the fact that αP is always a subset of $\mathcal{A}$, and hence checks over it remain unchanged, and (ii) the following lemma:

Lemma 8. *If $\sigma{\downarrow_{\alpha g_j}} \notin L(M_j{\downarrow_{\alpha g_j}})$ then $\sigma{\downarrow_{\alpha g'_j}} \notin L(M_j{\downarrow_{\alpha g'_j}})$ for any $\alpha g'_j \supseteq \alpha g_j$.*

For example, the constraints created in line 34 in Algorithm 3 are incremental. In order to re-use these constraints, we define an operator, *RC* (for *Refined Constraints*) which receives the full trace over $\alpha M_i \cup \alpha S(G_i)$ and an identifier of the constraints in the form of a pair of the premise index and the line in the algorithm in which the constraint was generated, referred to as the *type* of the constraint.

The *RC* operator then re-constructs the corresponding constraints by conducting projections according to the current alphabet, without re-performing any of the checks. For example, $RC(\sigma, i, 34, \mathcal{A}) = \{(\bigvee_{g_j \in S(G_i)} (-(\sigma{\downarrow_{\alpha g_j}}, j)) \vee +(\sigma a{\downarrow_{\alpha g_i}}, i))\}$.

Table 1. Results of comparison of 2-way compositional verification with and without alphabet refinement (2W-AR and 2W), n-way compositional verification with and without alphabet refinement (NW-AR and NW) and monolithic verification (Mon).

Case	2W [T]	2W+AR [T]	$\lvert g_1 \rvert$	$\lvert g_2 \rvert$	NW [T]	NW+AR [T]	$\lvert g_{max} \rvert$	$\lvert g_{min} \rvert$	Mon [T]	$\lvert Sys \| P_{err} \rvert$
GasSt 3	26	2.187	3	3	40.053	17.434	3	1	0.01	1280
GasSt 4	48	6.87	3	3	72.135	22.274	3	1	0.008	10466
GasSt 5	309	27	3	3	127.802	21.008	3	1	0.128	80368
ClServ 9	248	25.5	10	2	6.08	3.297	2	2	0.09	14335
ClServ 10	815.6	69	11	2	7	5	2	2	0.026	34303
ClServ 11	–	307.6	12	2	11.883	6.675	2	2	0.063	80895
ClServ 12	–	–	–	–	15.617	8.213	2	2	0.31	188415
MER 6 4	–	–	–	–	132.958	9.422	3	2	33.931	5246875
MER 7 4	–	–	–	–	769.61	23.894	3	2	–	–
MER 8 4	–	–	–	–	1611.072	54.568	3	2	–	–
MER 4 5	–	–	–	–	43.4	3.6	3	2	0.235	159192
MER 5 5	–	–	–	–	91.5	5.9	3	2	6.5	2057616
MER 6 5	–	–	–	–	200	11	3	2	500	23685696
MER 7 5	–	–	–	–	1470.85	30.38	3	2	–	–
MER 8 5	–	–	–	–	–	89.278	3	2	–	–

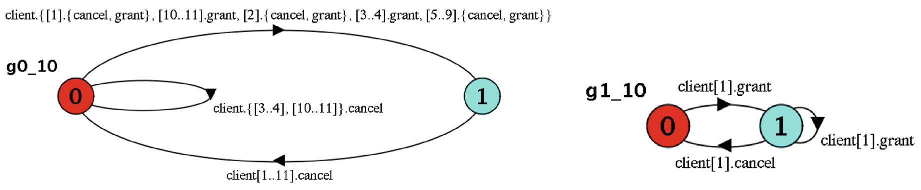
client.{[1].{cancel, grant}, [10..11].grant, [2].{cancel, grant}, [3..4].grant, [5..9].{cancel, grant}}
g0_10
0
1
client.{[3..4], [10..11]}.cancel
client[1..11].cancel
g1_10
client[1].grant
0
1
client[1].grant
client[1].cancel

Fig. 1. Assumptions generated for: server (left) and a client (right) for client-server (11 clients) using n-way compositional verification with alphabet refinement

8 Evaluation

We implemented our approach in the LTSA (Labelled Transition System Analyser) tool [17]. We use MiniSAT [9] for SAT solving. As an optimization we made ACR return (at each iteration) k counterexamples for the $n + 1$ premises where, k is $n \times \sum_{i=0}^{n} |g_i|$.

We evaluated our approach on the following examples [10,23]: Gas Station (3 to 5 customers), Client Server (9 to 12 clients), and a NASA rover model: MER (4 to 8 users competing for 4–5 resources). 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. We compared n-way verification using ACR with both 2-way ACR and monolithic verification.

Table 1 summarizes our results. We report the run time for: 2W (2-way ACR without alphabet refinement), 2W+AR (2-way ACR with alphabet refinement), NW (n-way ACR without alphabet refinement), NW+AR (n-way ACR with alphabet refinement); $|g_1|$ and $|g_2|$ are the assumption sizes produced by 2W, $|g_{max}|$ and $|g_{min}|$ are the sizes of the largest and smallest assumptions produced by NW. For 2W, each system was decomposed into two sub-systems, according to some "best decomposition" obtained before [10,23]. Mon is the run time of the monolithic classical algorithm and $|Sys\|P_{err}|$ is the number of states for the verification task, where Sys is the system $M_1\|M_1..\|M_n$. We put a limit of 1800 s for each experiment; "–" indicates that the time for that case exceeds this limit.

The results show that NW is better than 2W, and generates smaller assumptions. For example, Fig. 1 illustrates the small assumptions generated for the client-server example. Note that computing the "best" 2–way decomposition is expensive (its cost is not reported here). In contrast NW simply uses the natural decomposition of the system into its multiple components. Our results also show that alphabet refinement with rule simplification always improves circular reasoning, both in terms of analysis time and assumption sizes. Furthermore Mon performs better for small systems, but as the systems get larger n-way compositional verification significantly outperforms it. These lead to cases such as MER where, for large parameter values, Mon runs out of resources while NW+AR succeeds in under 2 min.

9 Conclusion and Future Work

We presented an automatic technique for the compositional analysis of systems decomposed into n components. The technique uses iterative assumption generation with incremental alphabet refinement and dynamic rule simplification. Preliminary results show its promise in practice. In the future we plan to check the rule premises in parallel to speed-up our approach. Further we plan to explore abstraction-refinement and learning as alternatives to our SAT-based assumption discovery.

Acknowledgments. We thank the reviewers for their detailed and helpful comments. This work was funded in part by the National Science Foundation (NSF Grant No. CSF-1329278) and the Binational Science Foundation (BSF Grant No. 2012259). Shoham was supported by the European Research Council under the European Union's Seventh Framework Program (FP7/2007–2013)/ERC grant agreement no. [321174-VSSC].

References

1. Alur, R., Henzinger, T.A.: Reactive modules. Formal Methods Syst. Des. **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. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
8. 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)
9. Een, N., Sörensson, N.: The minisat. http://minisat.se
10. Elkader, K.A., Grumberg, O., Păsăreanu, C.S., Shoham, S.: Automated circular assume-guarantee reasoning. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 23–39. Springer, Heidelberg (2015)
11. Gheorghiu, M.D., Giannakopoulou, D., Păsăreanu, C.S.: Refining interface alphabets for compositional verification. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 292–307. Springer, Heidelberg (2007)
12. Gupta, A., McMillan, K.L., Fu, Z.: Automated assumption generation for compositional verification. Formal Methods Syst. Des. **32**(3), 285–301 (2008)
13. 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)
14. 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)
15. Henzinger, T.A., Qadeer, S., Rajamani, S.K., Taşıran, S.: An assume-guarantee rule for checking simulation. In: Gopalakrishnan, G.C., Windley, P. (eds.) FMCAD 1998. LNCS, vol. 1522, pp. 421–431. Springer, Heidelberg (1998)
16. Li, B., Dillig, I., Dillig, T., McMillan, K., Sagiv, M.: Synthesis of circular compositional program proofs via abduction. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 370–384. Springer, Heidelberg (2013)
17. Magee, J., Kramer, J.: Concurrency: State Models and Java Programs. Wiley, New York (1999)
18. 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)
19. 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)
20. 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)
21. Misra, J., Chandy, K.M.: Proofs of networks of processes. IEEE Trans. Softw. Eng. **7**(4), 417–426 (1981)
22. Namjoshi, K.S., Trefler, R.J.: On the completeness of compositional reasoning. In: Allen Emerson, E., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 139–153. Springer, Heidelberg (2000)

23. 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 Syst. Des. **32**(3), 175–205 (2008)
24. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Logics and Models of Concurrent Systems. NATO ASI Series (1985)
25. Rushby, J.: Formal verification of mcmillan's compositional assume-guarantee rule. CSL Technical report, SRI (2001)