Assume-Guarantee Reasoning for Deadlock

Sagar Chaki, Software Engineering Institute
Nishant Sinha, Carnegie Mellon University
chaki@sei.cmu.edu  nishants@cs.cmu.edu

Abstract—We extend the learning-based automated assume guarantee paradigm to perform compositional deadlock detection. We define Failure Automata, a generalization of finite automata that accept regular failure sets. We develop a learning algorithm \( L^F \) that constructs the minimal deterministic failure automaton accepting any unknown regular failure set using a minimally adequate teacher. We show how \( L^F \) can be used for compositional regular failure language containment, and deadlock detection, using non-circular and circular assume guarantee rules. We present an implementation of our techniques and encouraging experimental results on several non-trivial benchmarks.

I. INTRODUCTION

Ensuring deadlock freedom is one of the most critical requirements in the design and validation of systems. The biggest challenge toward the development of effective deadlock detection schemes remains the statespace explosion problem. Compositional reasoning [1], [2], [3] is recognized to be one of the most promising approaches for alleviating statespace explosion. This paper presents an automated compositional deadlock detection procedure based on assume-guarantee (AG) reasoning.

In general, AG reasoning revolves around a proof rule that relates system components and assumptions about them to global system properties. In order to apply the proof rule, one is normally required to construct manually appropriate assumptions that can discharge the premises of the rule. In most realistic situations however, suitable assumptions are quite complicated and the absence of automated assumption generation techniques has been a major stumbling block toward the wider practical adoption of AG reasoning.

An important breakthrough in this respect has been the use of learning algorithms for assumption construction [5]. The general idea is to learn an automaton corresponding to the weakest assumption [6] that can discharge the AG premises. The learning process is embedded in the overall verification procedure in a way that guarantees termination with the correct result. The choice of the learning algorithm is dictated by the kind of automaton that can represent the weakest assumption, which in turn depends on the verification goal. For example, in the case of trace containment [5], weakest assumptions are naturally represented as deterministic finite automata, and this leads to the use of the \( L^* \) [7] learning algorithm. Similarly, in the case of simulation [8], the corresponding choices are deterministic tree automata and the \( L^F \) learning algorithm.

However, neither of the above two options are appropriate for deadlock detection. Intuitively, word (as well as tree) automata are unable to capture failures [9], a critical concept for understanding, and detecting, deadlocks. Note that it is possible to devise schemes for transforming any deadlock detection problem to one of ordinary trace containment. However, such schemes invariably introduce new components and an exponential number of actions, and are thus not scalable. Our work, therefore, was initiated by the search for an appropriate automata-theoretic formalism that can handle failures directly. Our overall contribution is a deadlock detection algorithm that uses learning-based automated AG reasoning, and does not require the introduction of additional actions or components.

As we shall see, two key ingredients of our solution are: (i) a new type of acceptors for regular failure languages with a non-standard accepting condition, and (ii) a notion of parallel composition between these acceptors that is consistent with the parallel composition of the languages accepted by them. The accepting condition we use is novel, and employs a notion of maximality to crucially avoid the introduction of an exponential number of new actions. To the best of our knowledge, such acceptors and their composition have not been discussed before. In addition, we believe that this paper presents the first use of learning in the context of automated AG reasoning for deadlock detection. More specifically, we make the following contributions.

First, we present the theory of regular failure languages (RFLs) which are downward-closed, and define failure automata that exactly accept the set of regular failure languages. Although RFLs are closed under union and intersection, they are not closed under complementation, an acceptable price we pay for using the notion of maximality. Further, we show a Myhill-Nerode-like theorem for RFLs and failure automata. Second, we show that the failure language of an LTS \( M \) is regular and checking deadlock-freedom for \( M \) is a particular instance of the problem of checking containment of RFLs. We present an algorithm for checking containment of RFLs. Note that checking containment of a failure language \( L_1 \) by a failure language \( L_2 \) is not possible in the usual way by complementing \( L_2 \) and intersecting with \( L_1 \) since RFLs are not closed under complementation. Third, we present a sound and complete non-circular AG rule, called AG-NC, on failure languages for checking failure language specifications. Given failure languages \( L_1 \) and \( L_S \), we define the weakest assumption failure language \( L_W \): for every \( L_A \), if \( L_1 \parallel L_A \subseteq L_S \), then \( L_A \subseteq L_W \). We then show, constructively, that if failure languages \( L_1 \) and \( L_2 \) are regular, then \( L_W \) uniquely exists, is also regular, and hence is accepted by a minimum failure automaton \( A_W \). Fourth, we develop an algorithm \( L^F \) (pronounced “el-ef”) to learn the minimum deterministic failure automaton that accepts an unknown regular failure language \( U \) using a minimally adequate teacher that can
answer membership and candidate queries pertaining to $U$. We show how the teacher can be implemented using the RFL containment algorithm mentioned above. Fifth, we develop an automated and compositional deadlock detection algorithm that employs $AG$-$NC$ and $LF$. We also define a circular AG proof rule $AG$-$Circ$ for deadlock detection and show how it can be used for automated and compositional deadlock detection. Finally, we have implemented our approach in the COMFORT [10] reasoning framework. We present encouraging results on several non-trivial benchmarks, including an embedded OS, and Linux device drivers.

II. RELATED WORK

Machine learning techniques have been used in several contexts related to verification [11], [12], [13], [14], [15]. We follow the approach of Cobleigh et al. [5] (respectively Chaki et al. [8]) to automate assume-guarantee reasoning for trace-containment (respectively simulation) between finite state systems (Alur et al. [16] have also investigated symbolic learning in this context). However, we apply this general paradigm for deadlock detection. Further, the $LF$ algorithm that we present may be of independent interest. The use of circular AG rules was also investigated in the context of trace containment by Barringer et al. [17].

Overkamp has explored synthesis of supervisory controller for discrete-event systems [18] based on failure semantics [9]. His notion of the least restrictive supervisor that guarantees deadlock-free behavior is similar to the weakest failure assumption in our case. However, our approach differs from his as follows: (i) we use failure automata to represent failure traces, (ii) we use learning to compute the weakest failure assumption automatically, and (iii) our focus is on checking deadlocks in software modules. Williams et al. [19] investigate an approach based on static analysis for detecting deadlocks in software modules. Williams et al. [24] have presented a sound and complete assume-guarantee method in the context of an abstract process composition framework. However, they do not discuss deadlock detection, nor explore the use of learning.

In the rest of this paper we omit proofs for the sake of brevity. Detailed proofs can be found in an extended version of this paper [25].

III. FAILURE LANGUAGES AND AUTOMATA

In this section we present the theory of failure languages and failure automata. We consider a subclass of regular failure languages and provide a lemma relating regular failure languages and failure automata, analogous to Myhill-Nerode theorem for ordinary regular languages. We begin with a few standard [26] definitions.

Definition 1 (Labeled Transition System): A labeled transition system (LTS) is a quadruple $(S, Init, \Sigma, \delta)$ where: (i) $S$ is a set of states, (ii) $Init \subseteq S$ is a set of initial states, (iii) $\Sigma$ is a set of actions (alphabet), and (iv) $\delta \subseteq S \times \Sigma \times S$ is a transition relation.

We only consider LTSs such that both $S$ and $\Sigma$ are finite. We write $s \overset{\alpha}{\rightarrow} s'$ to mean $(s, \alpha, s') \in \delta$. A trace is any finite (possibly empty) sequence of actions, i.e., the set of all traces is $\Sigma^*$. We denote an empty trace by $\epsilon$, a singleton trace $\{\alpha\}$ by $\alpha$, and the concatenation of two traces $t_1$ and $t_2$ by $t_1 \cdot t_2$. We extend the relation $\delta$ to a function $\delta$ on a set of states in the usual way. We also employ the usual definitions of determinism and completeness for LTSs.

Definition 2 (Finite Automaton): A finite automaton is a pair $(M, F)$ such that $M = (S, Init, \Sigma, \delta)$ is an LTS and $F \subseteq S$ is a set of final states. Let $G = (M, F)$ be a finite automaton. Then $G$ is said to be deterministic (complete) iff the underlying LTS $M$ is deterministic (complete).

Definition 3 (Refusal): Let $M = (S, Init, \Sigma, \delta)$ be an LTS and $s \in S$ be any state of $M$. We say that $s$ refuses an action $\alpha$ iff $\forall s' \in S, (s, \alpha, s') \notin \delta$. We say that $s$ refuses a set of actions $R$, and denote this by $Ref(s, R)$, iff $s$ refuses every element of $R$. Note that the following holds: (i) $\forall s, Ref(s, \emptyset)$, and (ii) $\forall s, R, R' \cdot Ref(s, R) \land R' \subseteq R \Rightarrow Ref(s, R')$, i.e., $\text{refusals are downward-closed}$.

Definition 4 (Failure): Let $M = (S, Init, \Sigma, \delta)$ be an LTS. A pair $(t, R) \in \Sigma^* \times 2^\Sigma$ is said to be a failure of $M$ iff there exists some $s \in \delta(Init, t)$ such that $Ref(s, R)$. The set of all failures of an LTS $M$ is denoted by $\mathcal{F}(M)$.

Note that a failure consists of both, a trace, and a refusal set. A (possibly infinite) set of failures $L$ is said to be a failure language. Let us denote $2^\Sigma$ by $\Sigma$. Note that $L \subseteq \Sigma^* \times \Sigma$. Union and intersection of failure languages is defined in the usual way. The complement of $L$, denoted by $\overline{L}$, is defined to be $(\Sigma^* \times \Sigma) \setminus L$. A failure language is said to be downward-closed iff $\forall t \in \Sigma^*, \forall R \in \Sigma, (t, R) \in L \implies \forall R' \subseteq R, (t, R') \in L$. Note that in general, failure languages may not be downward closed. However, as we show later, failure languages generated from LTSs are always downward closed.
because the refusal sets at each state of an LTS are downward-closed. In this article, we focus on downward-closed failure languages, in particular, regular failure languages.

**Definition 5 (Deadlock):** An LTS $M$ is said to deadlock iff the following holds: $\mathcal{F}(M) \cap (\Sigma^* \times \{\delta\}) \neq \emptyset$. In other words, $M$ deadlocks iff it has a reachable state that refutes every action in its alphabet.

Let us denote the failure language $\Sigma^* \times \{\delta\}$ by $L_{\Delta \delta \delta \delta}$. Then, it follows that $M$ is deadlock-free iff $\mathcal{F}(M) \subseteq L_{\Delta \delta \delta \delta}$.

**Maximality.** Let $P$ be any subset of $\Sigma$. Then the set of maximal elements of $P$ is denoted by $Max(P)$ and defined as follows: $Max(P) = \{R \in P \mid \forall R’ \in P, R \subseteq R’ \}$

For example, if $P = \{\{a\}, \{b\}, \{a,b\}, \{a,c\}\}$, then $Max(P) = \{\{a\}, \{a,b\}\}$. A subset $P$ of $\Sigma$ is said to be maximal iff it is non-empty and $Max(P) = P$. Intuitively, a failure automaton is finite automata whose final states are labeled with maximal refusal sets. Thus, a failure $(t,R)$ is accepted by a failure automaton $M$ iff upon receiving input $t$, $M$ reaches a final state labeled with a refusal $R’$ such that $R \subseteq R’$. Note that the notion of maximality allows us to concisely represent downward-closed failure languages by using only the upper bounds of a set (according to subset partial order) to represent the complete set.

**Definition 6 (Failure Automaton):** A failure automaton (FLA) is a triple $(M,F,\mu)$ such that $M = (S, Init, \Sigma, \delta)$ is an LTS, $F \subseteq S$ is a set of final states, and $\mu : F \to 2^\Sigma$ is a mapping from the final states to $2^\Sigma$ such that: $\forall s \in F, \mu(s) \neq \emptyset \land \mu(s) = Max(\mu(s))$.

Let $A = (M,F,\mu)$ be a FLA. Then $A$ is said to be deterministic (respectively complete) iff the underlying LTS $M$ is deterministic (respectively complete).

**Definition 7 (Language of a FLA):** Let $A = (M,F,\mu)$ be a FLA such that $M = (S, Init, \Sigma, \delta)$. Then a failure $(t,R)$ is accepted by $A$ iff $\exists s \in F, \exists R’ \in \mu(s), s \in \delta(Init,t) \cap R \subseteq R’$. The language of $A$, denoted by $L(A)$, is the set of all failures accepted by $A$.

Every deterministic FLA $A$ can be extended to a complete deterministic FLA $A’$ such that $L(A’’) = L(A)$ by adding a non-final sink state. In the rest of this article we consider FLA and languages over a fixed alphabet $\Sigma$.

**Lemma 1:** A language is accepted by a FLA iff it is accepted by a deterministic FLA, i.e., deterministic FLA have the same accepting power as FLA in general.

**Proof:** (Sketch) By subset construction and properties of downward-closed sets.

**Regular Failure Languages (RFLs).** A failure language is said to be regular iff it is accepted by some FLA. It follows from the definition of FLAs that RFLs are downward closed. Hence the set of RFLs is closed under union and intersection but not under complementation. In addition, every regular failure language is accepted by an unique minimal deterministic FLA. The following Lemma is analogous to the Myhill-Nerode theorem for regular languages and ordinary finite automata.

**Lemma 2:** Every regular failure language (RFL) is accepted by a unique (up to isomorphism) minimal deterministic finite failure automaton.

Note that for any LTS $M$, $\mathcal{F}(M)$ is regular. Indeed, the failure automaton corresponding to $M = (S, Init, \Sigma, \delta)$ is $A = (M, S, \mu)$ such that $\forall s \in S, \mu(s) = Max(\{R \mid Ref(s,R)\})$.

IV. ASSUME-GUARANTEE REASONING FOR DEADLOCK

We now present an assume-guarantee style [4] proof rule for deadlock detection for systems composed of two components. We use the notion of parallel composition proposed in the theory of CSP [9] and define it formally.

**Definition 8 (LTS Parallel Composition):** Consider LTSs $M_1 = (S_1, Init_1, \Sigma_1, \delta_1)$ and $M_2 = (S_2, Init_2, \Sigma_2, \delta_2)$. Then the parallel composition of $M_1$ and $M_2$, denoted by $M_1 \parallel M_2$, is the LTS $(S_1 \times S_2, Init_1 \times Init_2, \Sigma_1 \cup \Sigma_2, \delta_2)$, such that $\delta_2((s_1,s_2), \alpha, (s_1’,s_2’)) \in \delta$ iff the following holds: $\forall i \in \{1,2\}, \alpha \in \Sigma_i \land s_i’ \in \delta_i \lor (\alpha \not\in \Sigma_i \land s_i = s_i’)$.

Without loss of generality, we assume that both $M_1$ and $M_2$ have the same alphabet $\Sigma$. Indeed, any system with two components having different alphabets, say $\Sigma_1$ and $\Sigma_2$, can be converted to a bisimilar (and hence deadlock equivalent) system [8] with two components each having the same alphabet $\Sigma_1 \cup \Sigma_2$. Thus, all languages and automata we consider in the rest of this article will also be over the same alphabet $\Sigma$. We now extend the notion of parallel composition to failure languages. Observe that the composition involves set-intersection on the trace part and set-union on the refusal part of failures. Proofs of all the lemmas are in the full version [25] of the paper.

**Definition 9 (Failure Language Composition):** The parallel composition of any two failure languages $L_1$ and $L_2$, denoted by $L_1 \parallel L_2$, is defined as follows: $L_1 \parallel L_2 = \{(t,R_1 \cup R_2) \mid (t,R_1) \in L_1 \land (t,R_2) \in L_2\}$.

**Lemma 3:** For any failure languages $L_1$, $L_2$, $L_1’$, and $L_2’$, the following holds: $(L_1 \subseteq L_1’ \land L_2 \subseteq L_2’) \implies (L_1 \parallel L_2) \subseteq (L_1' \parallel L_2’)$.

**Definition 10 (FLA Parallel Composition):** Consider two FLAs $A_1 = (M_1, F_1, \mu_1)$ and $A_2 = (M_2, F_2, \mu_2)$. The parallel composition of $A_1$ and $A_2$, denoted by $A_1 \parallel A_2$, is defined as the FLA $M_1 \parallel M_2$, $F_1 \times F_2, \mu_2$ such that $\mu_2(s_1,s_2) = Max(\{R_1 \cup R_2 \mid (R_1 \in \mu_1(s_1) \land R_2 \in \mu_2(s_2)\})$.

Note that we have used different notation (II and || respectively) to denote the parallel composition of automata and languages. Let $M_1, M_2$ be LTSs and $A_1, A_2$ be FLAs. Then the following two lemmas bridge the concepts of composition between automata and languages.

1FLA are closely related to automata on guarded strings [27], which contain arbitrary transition labels drawn from a partially-ordered set. In contrast, the state labels (refusals) in FLA are only maximal elements from such a set. Further, since it suffices to consider refusals at the end of a trace for checking deadlock freedom, we only label the final states of a FLA.

2For example, consider $\Sigma = \{\alpha\}$ and the RFL $L = \Sigma^* \times \emptyset$. Then $L = \Sigma^* \times \{\alpha\}$ is downward closed and hence is not an RFL.

3However, there exists RFLs that do not correspond to any LTS. In particular, any failure language $L$ corresponding to some LTS must satisfy the following condition: $\exists R \subseteq \Sigma_1, (\epsilon, R) \in L$. Thus, the RFL $\{(\alpha, \emptyset)\}$ does not correspond to any LTS.

4We overload the operator $\parallel$ to denote parallel composition in the context of both LTSs and FLAs. The actual meaning of the operator will be clear from the context.
Lemma 4: $F(M_1 \sqcup M_2) = F(M_1) \parallel F(M_2)$.  

Lemma 5: $L(A_1 \sqcup A_2) = L(A_1) \parallel L(A_2)$.  

Regular Failure Language Containment (RFLC). We develop a general compositional framework for checking regular failure language containment. This framework is also applicable to deadlock detection since, as we illustrate later, deadlock freedom is a form of RFLC. Recall that regular failure languages are not closed under complementation and hence, given RFLs $L_1$ and $L_2$, it is not possible to verify $L_1 \subseteq L_2$ in the usual manner, by checking if $L_1 \cap \overline{L_2} = \emptyset$. However, as is shown by the following crucial lemma, it is possible to check containment between RFLs using their representations in terms of deterministic FLA, without having to complement the automaton corresponding to $L_2$.

Lemma 6: Consider any FLA $A_1$ and $A_2$. Let $A'_1 = (M_1, F_1, \mu_1)$ and $A'_2 = (M_2, F_2, \mu_2)$ be the FLA obtained by determinizing $A_1$ and $A_2$ respectively, and let $M_1 = (S_1, init_1, \Sigma, \delta_1)$ and $M_2 = (S_2, init_2, \Sigma, \delta_2)$. Then $L(A_1) \subseteq L(A_2)$ iff for every reachable state $(s_1, s_2)$ of $M_1 \sqcup M_2$ the following condition holds: $s_1 \in F_1 \implies (s_2 \in F_2 \land (\forall t_1 \in \mu_1(s_1), \exists t_2 \in \mu_2(s_2), s_1 \delta t_1 \Rightarrow s_2 \delta t_2))$.

In other words, we can check if $L(A_1) \subseteq L(A_2)$ by determining $A_1$ and $A_2$, constructing the product of the underlying LTSs and checking if the condition in Lemma 6 holds on every reachable state of the product. The condition essentially says that for every reachable state $(s_1, s_2)$, if $s_1$ is final, then $s_2$ is also final and each refusal $R_1$ labeling $s_1$ is contained in some refusal $R_2$ labeling $s_2$.

Now suppose that $L(A_1)$ is obtained by composing two RFLs $L_1$ and $L_2$, i.e., $L(A_1) = L_1 \parallel L_2$ and let $L(A_2) = L_2$, the specification language. In order to check RFLC between $L_1 \parallel L_2$ and $L_2$, the approach presented in Lemma 6 will require us to directly compose $L_1$, $L_2$ and $L_2$, a potentially expensive computation. In the following, we first show that checking deadlock-freedom is a particular case of RFLC and then present a compositional technique to check RFLC (and hence deadlock-freedom) that avoids composing $L_1$ and $L_2$ (or their FLA-representations) directly.

Deadlock as Regular Failure Language Containment. Given three RFLs $L_1$, $L_2$ and $L_2$, we can use our regular language containment algorithm to verify whether $(L_1 \parallel L_2) \subseteq L_2$. If this is the case, then our algorithm returns TRUE. Otherwise it returns FALSE along with a counterexample $CE \in (L_1 \parallel L_2) \setminus L_2$. Also, we assume that $L_1$, $L_2$ and $L_2$ are represented as FLA. To use our algorithm for deadlock detection, recall that for any two LTSs $M_1$ and $M_2$, $M_1 \sqcup M_2$ is deadlock free iff $F(M_1 \sqcup M_2) \subseteq L_{\text{Dlk}}$. Let $L_1 = F(M_1)$, $L_2 = F(M_2)$ and $L_{\text{Dlk}} = L_{\text{Dlk}}$. Using Lemma 4, the above deadlock check reduces to verifying if $L_1 \parallel L_2 \subseteq L_2$. Observe that we can use our RFLC algorithm provided $L_1$, $L_2$ and $L_2$ are regular. Recall that since $M_1$ and $M_2$ are LTSs, $L_1$ and $L_2$ are regular. Also, $L_{\text{Dlk}}$ is regular since it is accepted by the failure automaton $A = (M, F, \mu)$ such that: (i) $M = \{\{s\}, \{s\}, \Sigma, \delta\}$, (ii) $\delta = \{s \xrightarrow{a} s \mid a \in \Sigma\}$, (iii) $F = \{s\}$, and (iv) $\mu(s) = \text{Max}(R \mid R \subseteq \Sigma)$. For instance, if $\Sigma = \{a, b, c\}$ then $\mu(s) = \{\{a, b\}, \{b, c\}, \{c, a\}\}$. Thus, deadlock detection is just a specific instance of RFLC.

Suppose we are given three RFLs $L_1$, $L_2$ and $L_2$ in the form of their accepting FLA $A_1$, $A_2$ and $A_3$. To check $L_1 \parallel L_2 \subseteq L_3$, we can construct the FLA $A_1 \sqcup A_2$ (cf. Lemma 10) and then check if $L(A_1 \sqcup A_2) \subseteq L(A_3)$ (cf. Lemma 5 and 6). The problem with this naive approach is state-space explosion. In order to alleviate this problem, we present a compositional language containment scheme based on AG-style reasoning.

A Non-circular AG Rule. Consider RFLs $L_1$, $L_2$ and $L_3$. We are interested in checking whether $L_1 \parallel L_2 \subseteq L_3$. In this context, the following non-circular AG proof rule, which we call AG-NC, is both sound and complete:

$$\frac{L_1 \parallel L_2 \subseteq L_3}{L_1 \parallel L_2 \subseteq L_3}$$

In principle, AG-NC enables us to prove $L_1 \parallel L_2 \subseteq L_3$ by discovering an assumption $L_A$ that discharges its two premises. In practice, it leaves us with two critical problems. First, it provides no effective method for constructing an appropriate assumption $L_A$. Second, if no appropriate assumption exists, i.e., if the conclusion of AG-NC does not hold, then AG-NC does not help in obtaining a counterexample to $L_1 \parallel L_2 \subseteq L_3$. In this paper we develop and employ a learning algorithm that solves both the above problems. More specifically, our algorithm learns automatically, and incrementally, the weakest assumption $L_A$ that can discharge the first premise of AG-NC. During this process, it is guaranteed to reach, in a finite number of steps, one of the following two situations, and thus always terminate with the correct result:

1. It discovers an assumption that can discharge both premises of AG-NC, and terminates with TRUE.
2. It discovers a counterexample $CE$ to $L_1 \parallel L_2 \subseteq L_3$, and returns FALSE along with $CE$.

Weakest Assumption. Consider the proof rule AG-NC. For any $L_1$ and $L_3$, let $\hat{L}$ be the set of all languages that can discharge the first premise of AG-NC. In other words, $\hat{L} = \{L_A \mid (L_1 \parallel L_2) \subseteq L_3\}$. The following central theorem asserts that $\hat{L}$ contains a unique weakest (maximal) element $L_W$ that is also regular. This result is crucial for showing the termination of our approach.

Theorem 1: Let $L_1$ and $L_2$ be any RFLs and $f$ is a failure. Let us define a language $L_W$ as follows: $L_W = \{f \mid (L_1 \parallel \{f\}) \subseteq L_3\}$. Then the following holds: (i) $L_1 \parallel L_2 \subseteq L_3$, (ii) $\forall L \cdot (L_1 \parallel L_2 \subseteq L_3) \implies L \subseteq L_W$, and (iii) $L_W$ is regular.

Proof: (Sketch) Parts (i) and (ii) can be proved from the definition of $L_W$. For (iii) we assume that $L_1$ and $L_2$ are represented as failure automata $A_1$ and $A_2$, and use them to construct a failure automata $A_W$ for $L_W$. The LTS for $A_W$ is the product of the LTSs of $A_1$ and $A_2$. For every state $(s_1, s_2)$, where $s_1$ and $s_2$ are final in their respective FLAs, we first compute a label $X$ as follows: we add a refusal $R$ to $X$ iff for each refusal $R_1$ labeling $s_1$ there exists a refusal $R_2$ labeling $s_2$ such that $R_1 \cup R \subseteq R_2$. Finally, if $X \neq \emptyset$, we make $(s_1, s_2)$ final and set $\mu(s_1, s_2) = \text{Max}(X)$.

Now that we have proved that the weakest environment $L_W$ is regular, we can apply a learning algorithm to iteratively construct a FLA assumption that accepts $L_W$. In particular, we develop a learning algorithm $L^\mathcal{P}$ that iteratively
learns the minimal DFLA corresponding to \( L_W \) by asking queries about \( L_W \) to a minimally adequate teacher (MAT) and learning from them. In the next section, we present \( L^F \). Subsequently, in Section VI, we describe how \( L^F \) is used in our compositional language containment procedure. A reader who is interested in the overall compositional deadlock detection algorithm more than the intricacies of \( L^F \) may skip directly to Section VI at this point.

V. LEARNING FLA

In this section we present an algorithm \( L^F \) to learn the minimal FLA that accepts an unknown RFL \( U \). Our algorithm will use a minimally adequate teacher (MAT) that can answer two kinds of queries regarding \( U \): (1) Membership query: Given a failure \( e \), the MAT returns \( \text{TRUE} \) if \( e \in U \) and \( \text{FALSE} \) otherwise. (2) Candidate query: Given a deterministic FLA \( C \), the MAT returns \( \text{TRUE} \) if \( E(C) = U \). Otherwise it returns \( \text{FALSE} \) along with a counterexample failure \( CE \in (E(C) \setminus U) \cup (U \setminus E(C)) \).

Observation Table. \( L^F \) uses an observation table to record the information it obtains by querying the MAT. The rows and columns of the table correspond to specific traces and failures respectively. Formally, a table is a triple \((T, E, R)\) where: (i) \( T \subseteq \Sigma^* \) is a set of traces, (ii) \( E \subseteq \Sigma^* \times \Sigma \) is a set of failures or experiments, and (iii) \( R \) is a function from \( T \times E \) to \( \{0, 1\} \) where \( T = T \cup (T \times \Sigma) \).

For any table \( T = (T, E, R) \), the function \( R \) is defined as follows: \( \forall t \in T, \forall e \in E, (t', R) \in R(t, e) = 1 \iff (t \cdot t', R) \in U \). Thus, given \( T \) and \( E \), algorithm \( L^F \) can compute \( R \) via membership queries to the MAT. For any \( t \in T \), we write \( R(t) \) to mean the function from \( E \) to \( \{0, 1\} \) defined as follows: \( \forall e \in E, R(t)(e) = R(t, e) \).

An observation table \( T = (T, E, R) \) is said to be well-formed iff: \( \forall t_1 \in T, \forall t_2 \in T, t_1 \neq t_2 \implies R(t_1) \neq R(t_2) \). Essentially, this means that any two distinct rows \( t_1 \) and \( t_2 \) of a well-formed table can be distinguished by some experiment \( e \in E \). This also imposes an upper-bound on the number of rows of any well-formed table, as expressed by the following lemma.

Lemma 7: Let \( n \) be the number of states of the minimal DFLA accepting \( U \) and let \( T = (T, E, R) \) be any well-formed observation table. Then \( |T| \leq n \).

Closed observation table. An observation table \( T = (T, E, R) \) is said to be closed iff it satisfies the following: \( \forall t \in T, \forall 0 \in \Sigma, \exists t' \in T, R(t \cdot \alpha) = R(t') \). Intuitively, this means that if we extend any trace \( t \in T \) by any action \( \alpha \) then the result is indistinguishable from an existing trace \( t' \in T \) by the current set of experiments \( E \). Note that any well-formed table can be extended so that it is both well-formed and closed. This can be achieved by the algorithm \textbf{MakeClosed} shown in Figure 1. Observe that at every step of \textbf{MakeClosed}, the table \( T \) remains well-formed and hence, by Lemma 7, cannot grow infinitely. Also note that restricting the occurrence of refusals to \( E \) allows us to avoid considering the exponential possible refusal extensions of a trace while closing the table. Exponential number of membership queries will only be required if all possible refusals occur in \( E \).

Input: Well-formed observation table \( T = (T, E, R) \) while \( T \) is not closed do
pick \( t \in T \) and \( \alpha \in \Sigma \) such that \( \forall t' \in T, R(t \cdot \alpha) \neq R(t') \) add \( t \cdot \alpha \) to \( T \) and update \( R \) accordingly return \( T \)

Fig. 1. Algorithm \textbf{MakeClosed} extends an input well-formed table \( T \) so that the resulting table is both well-formed and closed.

Overall \( L^F \) algorithm. Algorithm \( L^F \) is iterative. It initially starts with a table \( T = (T, E, R) \) such that \( T = \{\} \) and \( E = \emptyset \). Note that the initial table is well-formed. Subsequently, in each iteration \( L^F \) performs the following steps:

1) Make \( T \) closed by invoking \textbf{MakeClosed}.
2) Construct candidate DFLA \( C \) from \( T \) and make candidate query with \( C \).
3) If the answer is \( \text{TRUE} \), \( L^F \) terminates with \( C \) as the final answer.
4) Otherwise \( L^F \) uses the counterexample \( CE \) to the candidate query to add a single new failure to \( E \) and repeats from step 1.

In each iteration, \( L^F \) either terminates with the correct answer (step 3) or adds a new failure to \( E \) (step 4). In the latter scenario, the new failure to be added is constructed in a way that guarantees an upper bound on the total number of iterations of \( L^F \). This, in turn, ensures its ultimate termination.

We now present the procedures for: (i) constructing a candidate DFLA \( C \) from a closed and well-formed table \( T \) (used in step 2 above), and (ii) adding a new failure to \( E \) based on a counterexample to a candidate query (step 4).

Candidate construction. Let \( T = (T, E, R) \) be a closed and well-formed observation table. The candidate DFLA \( C \) is constructed from \( T \) as follows: \( C = (M, \text{Init}, \Sigma, \delta) \) such that: (i) \( S = T \), (ii) \( \text{Init} = \{t\} \), (iii) \( \delta = \{t \cdot t' \mid R(t \cdot \alpha) = R(t')\} \), (iv) \( F = \{t \mid \exists e \in E, R(t, e) = 1\} \), and (v) \( \mu(t) = \max\{R \mid R(t, e), R(e) = 1\} \).

Adding new failures. Let \( C = (M, \text{Init}, \Sigma, \delta) \) be a candidate DFLA such that \( M = (S, \text{Init}, \Sigma, \delta) \). Let \( CE = (t, R) \) be a counterexample to a candidate query made with \( C \). In other words, \( CE \in L(C) \iff CE \notin U \). The algorithm \textbf{NewExp} adds a single new failure to \( T \) as follows. Let \( t = \alpha_1 \ldots \alpha_k \). For \( 0 \leq i < k \), let \( t_i \) be the prefix of \( t \) of length \( i \) and \( t^I \) be the suffix of \( t \) of length \( k - i \). In other words, for \( 0 \leq i \leq k \), we have \( t_i \cdot t^I = t \).

Additionally, for \( 0 \leq i \leq k \), let \( s_i \) be the state of \( C \) reached by executing \( t_i \). In other words, \( s_i = \delta(t_i) \). Since the candidate \( C \) was constructed from an observation table \( T \), it corresponds to a row of \( T \), which in turn corresponds to a trace. Let us also refer to this trace as \( s_i \). Finally, let \( b_i = 1 \) if the failure \( (s_i \cdot t^I, R) \in U \) and \( 0 \) otherwise. Note that we can compute \( b_i \) by evaluating \( s_i \) and then making a membership query with \( (s_i \cdot t^I, R) \). In particular, \( s_0 = c \), and hence \( b_0 = 1 \) if \( CE \in U \) and \( 0 \) otherwise. We now consider two cases.

Case 1: \( b_0 = 0 \). In this case, there exists an index \( j \in \)
such that \( b_j = 0 \) and \( b_{j+1} = 1 \). \( L^F \) finds such an index \( j \) and adds the failure \((\ell^{j+1}, R)\) to \( E \). As a result, the table \( T \) becomes non-closed and therefore, the next candidate DFLA has strictly more states than the current candidate \( C \). Complete details can be found in the full version of this paper.

**Case 2:** \( h_0 = 1 \) In this case, \( L^F \) adds a new failure to \( E \) that leads to the next candidate differing from the current candidate \( C \) in at least one of the following three ways: (i) it has strictly more states, (ii) it has a new final state, and (iii) the labeling of one of the current final states gets augmented.

**Correctness of \( L^F \).** Algorithm \( L^F \) always returns the correct answer in step 3 since it always does so after a successful candidate query. To see that \( L^F \) always terminates, observe that in every iteration, the candidate \( C \) computed by \( L^F \) undergoes at least one of the following three changes:

- \( \text{(Ch1)} \) The number of states of \( C \), and hence the number of rows of the observation table \( T \), increases.
- \( \text{(Ch2)} \) The states and transitions of \( C \) remain unchanged but a state of \( C \) that was previously non-final becomes final.
- \( \text{(Ch3)} \) The states, transitions and final states of \( C \) remain unchanged but for some final state \( s \) of \( C \), the size of \( \mu(s) \) increases.

Of the above changes, \( \text{Ch1} \) can happen at most \( n \) times where \( n \) is the number of states of the minimal DFLA accepting \( U \). Between any two consecutive occurrences of \( \text{Ch1} \), there can only be a finite number of occurrences of \( \text{Ch2} \) and \( \text{Ch3} \). Hence there can only be a finite number of iterations of \( L^F \). Therefore, \( L^F \) always terminates.

**Number of iterations.** To analyze the complexity of \( L^F \) we have to impose a tighter bound on the number of iterations. We already know that \( \text{Ch1} \) can happen at most \( n \) times. Since a final state can never become non-final, \( \text{Ch2} \) can also occur at most \( n \) times. Now let the minimal DFLA accepting \( U \) be \( A = (M, F, \mu) \) such that \( M = (S, \text{init}, \Sigma, \delta) \). Consider the set \( P = \bigcup_{s \in F} \mu(s) \) and let \( n' = |P| \). Since each \( \text{Ch3} \) adds an element to \( \mu(s) \) for some \( s \in F \), the total number of occurrences of \( \text{Ch3} \) is at most \( n' \). Therefore the maximum number of iterations of \( L^F \) is \( 2n + n' = O(n + n') \).

**Time complexity.** Let us make the standard assumption that each MAT query takes \( O(1) \) time. From the above discussion, we see that the number of columns of the observation table is at most \( O(n + n') \). The number of rows is at most \( O(n) \). Let us assume that the size of \( \Sigma \) is a constant. Then the number of membership queries, and hence time, needed to fill up the table is \( O(n(n + n')) \).

Let \( m \) be the length of the longest counterexample returned by a candidate query. Then to add each new failure, we have to make \( O(\log(m)) \) membership queries to find the appropriate index \( j \). Also, let the time required to find the maximal element \( R_{\max} \) be \( O(m') \). Then total time required for constructing each new failure is \( O((n + n')(\log(m) + m')) \). Finally, the number of candidate queries equals the number of iterations and hence is \( O(n + n') \). Thus, in summary, we find that the time complexity of \( L^F \) is \( O((n + n')(n + \log(m) + m')) \), which is polynomial in \( n, n', m \) and \( m' \).

**Space complexity.** Let us again make the standard assumption that each MAT query takes \( O(1) \) space. Since the queries are made sequentially, total space requirement for all of them is still \( O(1) \). Also, the procedure for constructing a new failure can be performed in \( O(1) \) space. A trace corresponding to a table row can be \( O(n) \) long and there are \( O(n) \) of them. A failure corresponding to a table column can be \( O(n) \) long and there are \( O(n + n') \) of them. Space required to store the table elements is \( O(n(n + n')) \). Hence total space required for the observation table is \( O((n + m)(n + n')) \). Space required to store computed candidates is \( O(n^2) \). Therefore, the total space complexity is \( O((n + m)(n + n')) \) which is also polynomial in \( n, n' \) and \( m \).

**VI. COMPOSITIONAL LANGUAGE CONTAINMENT**

Given RFLs \( L_1, L_2 \) and \( L_S \) (in the form of FLAs that accept them) we want to check whether \( L_1 \parallel L_2 \subseteq L_S \). If not, we also want to generate a counterexample \( CE \in (L_1 \parallel L_2) \setminus L_S \). To this end, we invoke the \( L^F \) algorithm to learn the weakest environment corresponding to \( L_1 \) and \( L_S \). We present an implementation strategy for the MAT to answer the membership and candidate queries posed by \( L^F \). In the following we assume that \( A_1, A_2 \) and \( A_S \) are the given FLAs such that \( L(A_1) = L_1, L(A_2) = L_2 \) and \( L(A_S) = L_S \).

**Membership Query.** The answer to a membership query with failure \( e = (t, R) \) is \( TRUE \) if the following condition (which can be effectively decided) holds and \( FALSE \) otherwise: \( \forall (t, R_1) \in L_1, (t, R_1 \cup R) \in L_S \).

**Candidate Query.** A candidate query with a failure automaton \( C \) is answered step-wise as follows:

1. Check if \( L(A_1 \parallel C) \subseteq L(A_S) \). If not, let \( (t, R_1 \cup R) \) be the counterexample obtained. Note that \( (t, R) \in L(C) \setminus U \). We return \( FALSE \) to \( L^F \) along with the counterexample \( (t, R) \). If \( L(A_1 \parallel C) \subseteq L(A_S) \), we proceed to step 2.

2. Check if \( L(A_2) \subseteq L(C) \). If so, we have obtained an assumption, viz., \( L(C) \), that discharges both premises of AG-NC. In this case, the overall language containment algorithm terminates with TRUE. Otherwise let \( (t', R') \) be the counterexample obtained. We proceed to step 3.

3. We check if there exists \( (t', R'_1) \in L(A_1) \) such that \( (t', R'_1 \cup R') \not\in L(A_S) \). If so, then \( (t', R'_1 \cup R') \in L(A_1 \parallel A_2) \setminus L(A_S) \) and the overall language containment algorithm terminates with FALSE and the counterexample \( (t', R'_1 \cup R') \). Otherwise \( (t', R') \in U \setminus L(C) \) and we return \( FALSE \) to \( L^F \) along with the counterexample \( (t', R') \).

Note that in the above we are never required to compose \( A_1 \) with \( A_2 \). In practice, the candidate \( C \) (that we compose with \( A_1 \) in step 1 of the candidate query) is much smaller than \( A_2 \). Thus we are able to alleviate the state-space explosion problem. Also, note that our procedure will ultimately terminate with the correct result from either step 2 or 3 of the candidate query. This follows from the correctness of \( L^F \) algorithm: in the worst case, the candidate query will be made with a FLA \( C \) such that \( L(C) = L_W \). In this scenario, termination is guaranteed to occur due to Theorem 1.
VII. ARBITRARY COMPONENTS AND CIRCULARITY

We investigated two approaches for handling more than two components. First, we applied AG-NC recursively. This can be demonstrated for languages $L_1$, $L_2$, $L_3$ and $L_S$ by the following proof-rule.

\[
\begin{align*}
L_1 \parallel L_3 & \subseteq L_S \\
L_2 \parallel L_3 & \subseteq L_A \\
L_2 \parallel L_3 & \subseteq L_A
\end{align*}
\]

At the top-level, we apply AG-NC to the two languages $L_1$ and $L_2$. Now the second premise becomes $L_2 \parallel L_3 \subseteq L_A$ and we can again apply AG-NC. In terms of the implementation of the MAT, the only difference is in step 2 of the candidate query (cf. Section VI). More specifically, we now invoke the language containment procedure recursively with $L(A_2)$, $L(A_3)$, and $L(C)$ instead of checking directly for $L(A_2) \subseteq L(C)$. This technique can be extended to any finite number of components.

Circular AG Rule. We also explored a circular AG rule. Unlike AG-NC however, the circular rule is specific to deadlock detection and not applicable to language containment in general. For any RFL $L$ let us write $W(L)$ to denote the weakest assumption against which $L$ does not deadlock. In other words, $\forall L' \cdot L_1 \parallel L' \subseteq L_{Dlk} \iff L' \subseteq W(L)$. It can be shown that: (PROP) $\forall t \in \Sigma^* \cdot \forall R \in L_\Sigma \cdot (t, R) \in L \iff (t, \Sigma \setminus R) \not\in W(L)$. The following theorem provides a circular AG rule for deadlock detection.

Theorem 2: Consider any two RFLs $L_1$ and $L_2$. Then the following proof rule, which we call AG-Circ, is both sound and complete.

\[
\begin{align*}
L_1 \parallel L_3 & \subseteq L_{Dlk} \\
W(L_1) & \parallel W(L_3) \subseteq L_{Dlk}
\end{align*}
\]

TABLE 1

Experimental results. $C = \# \text{ of components}; S_t = \# \text{ states of largest component}; T = \text{time (seconds)}; M = \text{memory (MB)}; A = \# \text{ states of largest assumption}; * = \text{resource exhaustion}; - = \text{data unavailable}; \alpha = 1247; \beta = 1708. Best figures are highlighted.

<table>
<thead>
<tr>
<th>Exp</th>
<th>LOC</th>
<th>C</th>
<th>#</th>
<th>AG-NC</th>
<th>AG-Circ</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>7272</td>
<td>2</td>
<td>28/7</td>
<td>$*$</td>
<td>308/903/5</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>7432</td>
<td>2</td>
<td>28/4</td>
<td>$*$</td>
<td>766/1135/11</td>
</tr>
<tr>
<td>$\beta$</td>
<td>7272</td>
<td>2</td>
<td>28/4</td>
<td>$*$</td>
<td>1455/11</td>
</tr>
<tr>
<td>$\beta$</td>
<td>7432</td>
<td>2</td>
<td>28/4</td>
<td>$*$</td>
<td>1455/11</td>
</tr>
</tbody>
</table>

Implementation. To use this rule for deadlock detection for two components $L_1$ and $L_2$ we use the following iterative procedure:

1) Using the first premise, construct a candidate $C_1$ similar to Step 1 of the candidate query in AG-NC (cf. Section VI). Similarly, using the second premise, construct another candidate $C_2$. Construction of $C_1$ and $C_2$ proceeds exactly as in the case of AG-NC.

2) Check if $W(L(C_1)) \parallel W(L(C_2)) \subseteq L_{Dlk}$. This is done either directly or via a compositional language containment using AG-NC. We compute the automata for $W(L(C_1))$ and $W(L(C_2))$ using the procedure described in the proof of Theorem 1. If the check succeeds then there is no deadlock in $L_1 \parallel L_2$ we exit successfully. Otherwise, we proceed to Step 3.

3) From the counterexample obtained above construct $t \in \Sigma^*$ and $R \in \Sigma$ be such that $t \not\in W(L(C_1))$ and $t \not\in W(L(C_2))$. Check if $t \not\in W(L_1)$ and $t \not\in W(L_2)$. If both these checks pass then we have a counterexample $t$ to the overall deadlock detection problem and therefore we terminate unsuccessfully. Otherwise, without loss of generality, suppose $t \not\in L_1$. But then, from PROP, $(t, \Sigma_1 \setminus R \in W(L_1)$. Again from PROP, since $t \not\in W(L(C_1))$, $(t, \Sigma \setminus R \not\in L(C_1)$. This is equivalent to a failed candidate query for $C_1$ with counterexample $(t, \Sigma \setminus R)$, and we repeat from Step 1 above.

Note that even though we have presented AG-Circ in the context of only two components, it generalizes to an arbitrary, but finite, number of components.

VIII. EXPERIMENTAL VALIDATION AND CONCLUSION

We implemented our algorithms in the COMFORT [10] reasoning framework and experimented with a set of real-life examples. All our experiments were done on a 2.4 GHz
Pentium 4 machine running RedHat 9 and with time limit of 1 hour and a memory limit of 2 GB. Our results are summarized in Table I. The MC benchmarks are derived from MicroC version 2.70, a lightweight OS for real-time embedded applications. The IPC benchmark is based on an inter-process communication library used by an industrial robot controller software. The ide, syn, mx and tg3 examples are based on Linux device drivers. Finally, DP is a synthetic benchmark based on the well-known dining philosophers example.

For each example, we obtained a set of benchmarks by increasing the number of components. For each such benchmark, we tested a version without deadlock, and another with an artificially introduced deadlock. In all cases, deadlock was caused by incorrect synchronization between components – the only difference was in the synchronization mechanism. Specifically, the dining philosophers synchronized using “forks". In all other examples, synchronization was achieved via a shared “lock”.

For each benchmark, a finite LTS model was constructed via a predicate abstraction [10] that transformed the synchronization behavior into appropriate actions. For example, in the case of the ide benchmark, calls to the spin_lock and spin_unlock functions were transformed into lock and unlock actions respectively. Note that this makes sense because, for instance, multiple threads executing the driver for a specific device will acquire and release a common lock specific to that device by invoking spin_lock and spin_unlock respectively.

For each abstraction, appropriate predicates were supplied externally so that the resulting models would be precise enough to display the presence or absence of deadlock. In addition, care was taken to ensure that the abstractions were sound with respect to deadlocks, i.e., the extra behavior introduced did not eliminate any deadlock in the concrete system. Each benchmark was verified using explicit brute-force statespace exploration (referred to in Table I as “Plain”), the non-circular AG rule (referred as AG-NC), and the circular AG rule (referred as AG-Circ). When using AG-Circ, Step 2 (i.e., checking if \( W(C_1) \) \( \subseteq W(C_2) \)) was done via compositional language containment using AG-NC.

We observe that the AG-based methods outperform the naive approach for most of the benchmarks. More importantly, for each benchmark, the growth in memory consumption with increasing number of components is benign for both AG-based approaches. This indicates that AG reasoning is effective in combating statespace explosion even for deadlock detection. We also note that larger assumptions (and hence time and memory) are required for detecting deadlocks as opposed to detecting deadlock freedom. Among the AG-based approaches, AG-Circ is in general faster than AG-NC but (on a few occasions) consumes negligible extra memory. In several cases, AG-NC runs out of time while AG-Circ is able to terminate successfully. Overall, whenever AG-NC and AG-Circ differ significantly in any real-life example, AG-Circ is superior.

Conclusion. We have extended the learning-based automated assume guarantee paradigm to deadlock detection. We have defined a new kind of automata that are similar to finite automata but accept failures instead of traces. We have also developed an algorithm, \( L^F \), that learns the minimal failure automata accepting an unknown regular failure language using a minimally adequate teacher. We have shown how \( L^F \) can be used for compositional deadlock detection using both circular and non-circular assume-guarantee rules. Finally, we have implemented our technique and have obtained encouraging experimental results on several non-trivial benchmarks.

REFERENCES


