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) [4] 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^T 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 L^F. 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 L^F 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 that can be caused by incorrect lock manipulation by Java libraries, and also provide an excellent survey of related research. The problem of detecting deadlocks for pushdown programs communicating only via nested locking has been investigated by Kahlon et al. [20]. In contrast, we present a model checking based framework to compositionally verify deadlock freedom for non-recursive programs with arbitrary lock-based or rendezvous communication. Other non-compositional techniques for detecting deadlock have been investigated in context of partial-order reduction [21] and for checking refinement of CCS processes, using a more discriminative (than failure trace refinement) notion called stuck-free conformance [22].

Brookes and Roscoe [23] use the failure model to show the absence of deadlock in undirectional networks. They also generalize the approach to the class of conflict-free networks via decomposition and local deadlock analysis. In contrast, we provide a completely automated framework for detecting deadlocks in arbitrary networks of asynchronous systems using rendezvous communication. Our formalism is based on an automata-theoretic representation of failure traces. Moreover, in order to analyze the deadlock-freedom of a set of concurrent programs compositionally, we use both circular and non-circular assume-guarantee [4], [1], [17] rules. Amla 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) Σ 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 Σ are finite. We write $s \xrightarrow{\alpha} s'$ 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 Σ^*. We denote an empty trace by ϵ, a singleton trace $\langle\alpha\rangle$ by α, and the concatenation of two traces t_1 and t_2 by $t_1 \bullet t_2$. We extend the relation δ to a function $\widehat{\delta}$ 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 α 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) $\forall s, R, R' \,.\, Ref(s, R) \wedge R' \subseteq R \implies Ref(s, R')$, i.e., 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 $s \in \widehat{\delta}(Init, t)$ 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^Σ by $\widehat{\Sigma}$. Note that $L \subseteq \Sigma^* \times \widehat{\Sigma}$. Union and intersection of failure languages is defined in the usual way. The complement of L, denoted by $\overline{L}$, is defined to be $(\Sigma^* \times \widehat{\Sigma}) \setminus L$. A failure language is said to be *downward-closed* iff $\forall t \in \Sigma^* \,.\, \forall R \in \widehat{\Sigma} \,.\, (t, R) \in L \implies \forall R' \subseteq R.\ (t, R') \in L$. 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: $\mathcal{F}(M) \cap (\Sigma^* \times \{\Sigma\}) \neq \emptyset$. In other words, M deadlocks iff it has a reachable state that refuses every action in its alphabet.

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

Maximality. Let P be any subset of $\widehat{\Sigma}$. Then the set of maximal elements of P is denoted by $Max(P)$ and defined as follows: $Max(P) = \{R \in P \mid \forall R' \in P \centerdot R \not\subset R'\}$

For example, if $P = \{\{a\}, \{b\}, \{a, b\}, \{a, c\}\}$, then $Max(P) = \{\{a, b\}, \{a, c\}\}$. A subset P of $\widehat{\Sigma}$ is said to be *maximal* iff it is non-empty and $Max(P) = P$. Intuitively, failure automata are 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, μ) such that $M = (S, Init, \Sigma, \delta)$ is an LTS, $F \subseteq S$ is a set of final states, and $\mu : F \rightarrow 2^{\widehat{\Sigma}}$ is a mapping from the final states to $2^{\widehat{\Sigma}}$ such that: $\forall s \in F \centerdot \mu(s) \neq \emptyset \wedge \mu(s) = Max(\mu(s))$.

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 $\exists s \in F \centerdot \exists R' \in \mu(s) \centerdot s \in \widehat{\delta}(Init, t) \wedge R \subseteq R'$. The language of A, denoted by $\mathcal{L}(A)$, is the set of all failures accepted by A.

Every deterministic FLA A can be extended to a complete deterministic FLA A' such that $\mathcal{L}(A') = \mathcal{L}(A)$ by adding a non-final *sink* state. In the rest of this article we consider FLA and languages over a fixed alphabet Σ. [1]

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[2]. 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[3]. Indeed, the failure automaton corresponding to $M = (S, Init, \Sigma, \delta)$ is $A = (M, S, \mu)$ such that $\forall s \in S \centerdot \mu(s) = Max(\{R \mid Ref(s, R)\})$.

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 $M_1 \amalg M_2$, is the LTS $(S_1 \times S_2, Init_1 \times Init_2, \Sigma_1 \cup \Sigma_2, \delta)$, such that $((s_1, s_2), \alpha, (s_1', s_2')) \in \delta$ iff the following holds: $\forall i \in \{1, 2\} \centerdot (\alpha \in \Sigma_i \wedge (s_i, \alpha, s_i') \in \delta_i) \vee (\alpha \notin \Sigma_i \wedge s_i = s_i')$.

Without loss of generality, we assume that both M_1 and M_2 have the same alphabet Σ. Indeed, any system with two components having different alphabets, say Σ_1 and Σ_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 Σ. 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: $L_1 \parallel L_2 = \{(t, R_1 \cup R_2) \mid (t, R_1) \in L_1 \wedge (t, R_2) \in L_2\}$.

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

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 $A_1 \amalg A_2$[4], is defined as the FLA $(M_1 \amalg M_2, F_1 \times F_2, \mu)$ such that $\mu(s_1, s_2) = Max(\{R_1 \cup R_2 \mid R_1 \in \mu_1(s_1) \wedge R_2 \in \mu_2(s_2)\})$.

Note that we have used different notation ($\amalg$ and $\parallel$ 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.

[1] FLA 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.

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

[3] However, 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: $\exists R \subseteq \Sigma \centerdot (\epsilon, R) \in L$. Thus, the RFL $\{(\alpha, \emptyset)\}$ does not correspond to any LTS.

[4] We overload the operator $\amalg$ 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: $\mathcal{F}(M_1 \amalg M_2) = \mathcal{F}(M_1) \parallel \mathcal{F}(M_2)$.

Lemma 5: $\mathcal{L}(A_1 \amalg A_2) = \mathcal{L}(A_1) \parallel \mathcal{L}(A_2)$.

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 $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_2)$ iff for every reachable state (s_1, s_2) of $M_1 \amalg M_2$ the following condition holds: $s_1 \in F_1 \implies (s_2 \in F_2 \wedge (\forall R_1 \in \mu_1(s_1) \,.\, \exists R_2 \in \mu_2(s_2) \,.\, R_1 \subseteq R_2))$.

In other words, we can check if $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_2)$ by determinizing 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 $\mathcal{L}(A_1)$ is obtained by composing two RFLs L_1 and L_2, i.e., $\mathcal{L}(A_1) = L_1 \parallel L_2$ and let $\mathcal{L}(A_2) = L_S$, the specification language. In order to check RFLC between $L_1 \parallel L_2$ and L_S, the approach presented in lemma 6 will require us to directly compose L_1, L_2 and L_S, 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_S, we can use our regular language containment algorithm to verify whether $(L_1 \parallel L_2) \subseteq L_S$. If this is the case, then our algorithm returns TRUE. Otherwise it returns FALSE along with a counterexample $CE \in (L_1 \parallel L_2) \setminus L_S$. Also, we assume that L_1, L_2 and L_S are represented as FLA. To use our algorithm for deadlock detection, recall that for any two LTSs M_1 and M_2, $M_1 \amalg M_2$ is deadlock free iff $\mathcal{F}(M_1 \amalg M_2) \subseteq \overline{L_{Dlk}}$. Let $L_1 = \mathcal{F}(M_1)$, $L_2 = \mathcal{F}(M_2)$ and $L_S = \overline{L_{Dlk}}$. Using Lemma 4, the above deadlock check reduces to verifying if $L_1 \parallel L_2 \subseteq L_S$. Observe that we can use our RFLC algorithm provided L_1, L_2 and L_S are regular. Recall that since M_1 and M_2 are LTSs, L_1 and L_2 are regular. Also, $\overline{L_{Dlk}}$ is regular since it is accepted by the failure automaton $A = (M, F, \mu)$ such that: (i) $M = (\{s\}, \{s\}, \Sigma, \delta)$, (ii) $\delta = \left\{ s \xrightarrow{\alpha} s \mid \alpha \in \Sigma \right\}$, (iii) $F = \{s\}$, and (iv) $\mu(s) = Max(\{R \mid R \subset \Sigma\})$. 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_S in the form of their accepting FLA A_1, A_2 and A_S. To check $L_1 \parallel L_2 \subseteq L_S$, we can construct the FLA $A_1 \amalg A_2$ (cf. Lemma 10) and then check if $\mathcal{L}(A_1 \amalg A_2) \subseteq \mathcal{L}(A_S)$ (cf. Lemma 5 and 6). The problem with this naive approach is statespace 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_S. We are interested in checking whether $L_1 \parallel L_2 \subseteq L_S$. In this context, the following non-circular AG proof rule, which we call **AG-NC**, is both sound and complete:

$$\frac{L_1 \parallel L_A \subseteq L_S \qquad L_2 \subseteq L_A}{L_1 \parallel L_2 \subseteq L_S}$$

In principle, **AG-NC** enables us to prove $L_1 \parallel L_2 \subseteq L_S$ 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 $L_1 \parallel L_2 \subseteq L_S$. 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_W 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 $L_1 \parallel L_2 \subseteq L_S$, and returns FALSE along with CE.

Weakest Assumption. Consider the proof rule **AG-NC**. For any L_1 and L_S, let $\widehat{L}$ be the set of all languages that can discharge the first premise of **AG-NC**. In other words, $\widehat{L} = \{L_A \mid (L_1 \parallel L_A) \subseteq L_S\}$. The following central theorem asserts that $\widehat{L}$ 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_S be any RFLs and f is a failure. Let us define a language L_W as follows: $L_W = \{f \mid (L_1 \parallel \{f\}) \subseteq L_S\}$. Then the following holds: (i) $L_1 \parallel L_W \subseteq L_S$, (ii) $\forall L \,.\, L_1 \parallel L \subseteq L_S \iff L \subseteq L_W$, 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_S 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 $\mu(s_1, s_2) = Max(X)$. ∎

Now that we have proved that the weakest environment assumption 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^F 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 TRUE if $e \in U$ and FALSE otherwise. **(2) Candidate query:** Given a deterministic FLA C, the MAT returns TRUE if $\mathcal{L}(C) = U$. Otherwise it returns FALSE along with a counterexample failure $CE \in (\mathcal{L}(C) \setminus U) \bigcup (U \setminus \mathcal{L}(C))$.

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 $(\mathbb{T}, \mathbb{E}, \mathbb{R})$ where: (i) $\mathbb{T} \subseteq \Sigma^*$ is a set of traces, (ii) $\mathbb{E} \subseteq \Sigma^* \times \widehat{\Sigma}$ is a set of failures or experiments, and (iii) $\mathbb{R}$ is a function from $\widehat{\mathbb{T}} \times \mathbb{E}$ to $\{0, 1\}$ where $\widehat{\mathbb{T}} = \mathbb{T} \cup (\mathbb{T} \bullet \Sigma)$.

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

An observation table $\mathcal{T} = (\mathbb{T}, \mathbb{E}, \mathbb{R})$ is said to be well-formed iff: $\forall t_1 \in \mathbb{T} \,.\, \forall t_2 \in \mathbb{T} \,.\, t_1 \neq t_2 \implies \mathbb{R}(t_1) \neq \mathbb{R}(t_2)$. Essentially, this means that any two distinct rows t_1 and t_2 of a well-formed table can be distinguished by some experiment $e \in \mathbb{E}$. 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 $\mathcal{T} = (\mathbb{T}, \mathbb{E}, \mathbb{R})$ be any well-formed observation table. Then $|\mathbb{T}| \leq n$.

Closed observation table. An observation table $\mathcal{T} = (\mathbb{T}, \mathbb{E}, \mathbb{R})$ is said to be *closed* iff it satisfies the following: $\forall t \in \mathbb{T} \,.\, \forall \alpha \in \Sigma \,.\, \exists t' \in \mathbb{T} \,.\, \mathbb{R}(t \bullet \alpha) = \mathbb{R}(t')$. Intuitively, this means that if we extend any trace $t \in \mathbb{T}$ by any action α then the result is indistinguishable from an existing trace $t' \in \mathbb{T}$ by the current set of experiments $\mathbb{E}$. 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 **MakeClosed** shown in Figure 1. Observe that at every step of **MakeClosed**, the table $\mathcal{T}$ remains well-formed and hence, by Lemma 7, cannot grow infinitely. Also note that restricting the occurrence of refusals to $\mathbb{E}$ 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 $\mathbb{E}$.

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

Fig. 1. Algorithm **MakeClosed** extends an input well-formed table $\mathcal{T}$ 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 $\mathcal{T} = (\mathbb{T}, \mathbb{E}, \mathbb{R})$ such that $\mathbb{T} = \{\epsilon\}$ and $\mathbb{E} = \emptyset$. Note that the initial table is well-formed. Subsequently, in each iteration L^F performs the following steps:

1) Make $\mathcal{T}$ closed by invoking **MakeClosed**.
2) Construct candidate DFLA C from $\mathcal{T}$ and make candidate query with C.
3) If the answer is 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 $\mathbb{E}$ and repeats from step 1.

In each iteration, L^F either terminates with the correct answer (step 3) or adds a new failure to $\mathbb{E}$ (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 $\mathcal{T}$ (used in step 2 above), and (ii) adding a new failure to $\mathbb{E}$ based on a counterexample to a candidate query (step 4).

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

Adding new failures. Let $C = (M, F, \mu)$ be a candidate DFLA such that $M = (S, Init, \Sigma, \delta)$. Let $CE = (t, R)$ be a counterexample to a candidate query made with C. In other words, $CE \in \mathcal{L}(C) \iff CE \notin U$. The algorithm **NewExp** adds a single new failure to $\mathcal{T}$ as follows. Let $t = \alpha_1 \bullet \ldots \bullet \alpha_k$. For $0 \leq i \leq k$, 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 $t_i \bullet t^i = t$.

Additionally, for $0 \leq i \leq k$, let s_i be the state of C reached by executing t_i. In other words, $s_i = \widehat{\delta}(t_i)$. Since the candidate C was constructed from an observation table $\mathcal{T}$, it corresponds to a row of $\mathcal{T}$, 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 $\left(s_i \bullet t^i, R\right) \in U$ and 0 otherwise. Note that we can compute b_i by evaluating s_i and then making a membership query with $\left(s_i \bullet t^i, R\right)$. In particular, $s_0 = \epsilon$, 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 $j \in$

$\{0, \ldots, k\}$ such that $b_j = 0$ and $b_{j+1} = 1$. L^F finds such an index j and adds the failure $\left(t^{j+1}, R\right)$ to $\mathbb{E}$. As a result, the table $\mathcal{T}$ becomes non-closed and therefore, the next candidate FLA has strictly more states than the current candidate C. Complete details can be found in the full version of this paper.

Case 2: [$b_0 = 1$] In this case, L^F adds a new failure to $\mathbb{E}$ 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. Complete details can be found in the full version of this paper.

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:

- **(Ch1)** The number of states of C, and hence the number of rows of the observation table $\mathcal{T}$, increases.
- **(Ch2)** The states and transitions of C remain unchanged but a state of C that was previously non-final becomes final.
- **(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, **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 **Ch1**, there can only be a finite number of occurrences of **Ch2** and **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 **Ch1** can happen at most n times. Since a final state can never become non-final, **Ch2** can also occur at most n times. Now let the minimal DFLA accepting U be $A = (M, F, \mu)$ such that $M = (S, Init, \Sigma, \delta)$. Consider the set $P = \bigcup_{s \in F} \mu(s)$ and let $n' = |P|$. Since each **Ch3** adds an element to $\mu(s)$ for some $s \in F$, the total number of occurrences of **Ch3** is at most n'. Therefore the maximum number of iterations of L^F is $2n + n' = \mathcal{O}(n + n')$.

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

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

Space complexity. Let us again make the standard assumption that each MAT query takes $\mathcal{O}(1)$ space. Since the queries are made sequentially, total space requirement for all of them is still $\mathcal{O}(1)$. Also, the procedure for constructing a new failure can be performed in $\mathcal{O}(1)$ space. A trace corresponding to a table row can be $\mathcal{O}(n)$ long and there are $\mathcal{O}(n)$ of them. A failure corresponding to a table column can be $\mathcal{O}(m)$ long and there are $\mathcal{O}(n + n')$ of them. Space required to store the table elements is $\mathcal{O}(n(n + n'))$. Hence total space required for the observation table is $\mathcal{O}((n + m)(n + n'))$. Space required to store computed candidates is $\mathcal{O}(n^2)$. Therefore, the total space complexity is $\mathcal{O}((n + m)(n + n'))$ 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 FLA that accept them) we want to check whether $L_1 \parallel L_2 \subseteq L_S$. If not, we also want to generate a counterexamples $CE \in (L_1 \parallel L_2) \setminus L_S$. 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 $\mathcal{L}(A_1) = L_1$, $\mathcal{L}(A_2) = L_2$ and $\mathcal{L}(A_S) = L_S$.

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: $\forall (t, R_1) \in L_1 \centerdot (t, R_1 \cup R) \in L_S$.

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

1) Check if $\mathcal{L}(A_1 \amalg C) \subseteq \mathcal{L}(A_S)$. If not, let $(t, R_1 \cup R)$ be the counterexample obtained. Note that $(t, R) \in \mathcal{L}(C) \setminus U$. We return FALSE to L^F along with the counterexample (t, R). If $\mathcal{L}(A_1 \amalg C) \subseteq \mathcal{L}(A_S)$, we proceed to step 2.
2) Check if $\mathcal{L}(A_2) \subseteq \mathcal{L}(C)$. If so, we have obtained an assumption, viz., $\mathcal{L}(C)$, 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 $(t', R'_1) \in \mathcal{L}(A_1)$ such that $(t', R'_1 \cup R') \notin \mathcal{L}(A_S)$. If so, then $(t', R'_1 \cup R') \in \mathcal{L}(A_1 \amalg A_2) \setminus \mathcal{L}(A_S)$ and the overall language containment algorithm terminates with FALSE and the counterexample $(t', R'_1 \cup R')$. Otherwise $(t', R') \in U \setminus \mathcal{L}(C)$ 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 statespace 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 $\mathcal{L}(C) = L_W$. In this scenario, termination is guaranteed to occur due to Theorem 1.

Exp	LOC	C	St	No Deadlock								Deadlock							
				Plain		AG-NC			AG-Circ			Plain		AG-NC			AG-Circ		
				T	M	T	M	A	T	M	A	T	M	T	M	A	T	M	A
MC	7272	2	2874	-	*	308	**903**	**5**	**307**	**903**	6	372	β	386	980	**13**	**313**	**979**	16
MC	7272	3	2874	-	*	766	**1155**	**11**	**459**	**1155**	12	-	-	-	-	-	-	-	-
MC	7272	4	2874	-	*	*	1453	-	**716**	**1453**	24	-	-	-	-	-	-	-	-
ide	18905	3	672	571	*	338	50	**11**	**62**	**47**	12	755	*	*	80	-	**557**	**551**	**125**
ide	18905	4	716	972	*	*	63	-	**195**	**55**	**24**	978	*	*	84	-	2913	*	-
ide	18905	5	760	1082	*	*	84	-	**639**	**85**	**48**	1082	*	*	89	-	*	498	-
syn	17262	4	117	733	*	1547	**19**	**21**	**58**	21	24	864	*	**127**	**181**	**2**	133	**181**	6
syn	17262	5	127	713	*	*	19	-	**224**	**47**	**48**	1088	*	844	*	-	867	*	-
syn	17262	6	137	767	*	*	27	-	**1815**	**189**	**96**	-	*	1188	*	-	-	*	-
mx	15717	3	1995	1154	*	2079	140	**11**	**639**	**123**	12	1182	*	657	**364**	**2**	**630**	**364**	5
mx	15717	4	2058	1545	*	-	168	-	**713**	**139**	24	1309	*	1627	*	-	1206	*	-
mx	15717	5	2121	1660	*	-	179	-	**2131**	**185**	48	-	*	3368	*	-	2276	*	-
tg3	36774	3	1653	971	*	1568	118	**11**	**406**	**111**	12	894	*	**486**	**393**	**2**	499	**393**	5
tg3	36774	4	1673	927	*	-	149	-	**486**	**131**	24	1096	*	1036	*	-	1037	*	-
tg3	36774	5	1693	1086	*	-	158	-	**1338**	**165**	48	-	*	2186	*	-	1668	*	-
tg3	36774	6	1713	1252	*	-	157	-	**3406**	**313**	96	1278	*	*	-	-	1954	*	-
IPC	818	3	302	**195**	α	703	**338**	**49**	478	355	**49**	-	-	-	-	-	-	-	-
DP	82	6	30	274	*	**100**	**330**	11	286	414	**9**	-	-	-	-	-	-	-	-
DP	109	8	30	302	*	**1551**	**565**	**11**	*	1474	-	-	-	-	-	-	-	-	-

TABLE I

EXPERIMENTAL RESULTS. C = # OF COMPONENTS; ST = # OF STATES OF LARGEST COMPONENT; T = TIME (SECONDS); M = MEMORY (MB); A = # OF STATES OF LARGEST ASSUMPTION; * = RESOURCE EXHAUSTION; - = DATA UNAVAILABLE; $\alpha = 1247$; $\beta = 1708$. BEST FIGURES ARE HIGHLIGHTED.

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.

$$\frac{L_1 \parallel L_A^1 \subseteq L_S \qquad \dfrac{L_2 \parallel L_A^2 \subseteq L_A^1 \qquad L_3 \subseteq L_A^2}{L_2 \parallel L_3 \subseteq L_A^1}}{L_1 \parallel L_2 \parallel L_3 \subseteq L_S}$$

At the top-level, we apply **AG-NC** on the two languages L_1 and $L_2 \parallel L_3$. Now the second premise becomes $L_2 \parallel L_3 \subseteq L_A^1$ 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 $\mathcal{L}(A_2)$, $\mathcal{L}(A_3)$ and $\mathcal{L}(C)$ instead of checking directly for $\mathcal{L}(A_2) \subseteq \mathcal{L}(C)$. 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, $\forall L' \,.\, L \parallel L' \subseteq \overline{L_{Dlk}} \iff L' \subseteq W(L)$. It can be shown that: **(PROP)** $\forall t \in \Sigma^* \,.\, \forall R \in \widehat{\Sigma} \,.\, (t, R) \in L \iff (t, \Sigma \setminus R) \notin W(L)$. 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.

$$\frac{L_1 \parallel L_A^1 \subseteq \overline{L_{Dlk}} \qquad L_2 \parallel L_A^2 \subseteq \overline{L_{Dlk}} \qquad W(L_A^1) \parallel W(L_A^2) \subseteq \overline{L_{Dlk}}}{L_1 \parallel L_2 \subseteq \overline{L_{Dlk}}}$$

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 $W(\mathcal{L}(C_1)) \parallel W(\mathcal{L}(C_2)) \subseteq \overline{L_{Dlk}}$. This is done either directly or via a compositional language containment using **AG-NC**. We compute the automata for $W(\mathcal{L}(C_1))$ and $W(\mathcal{L}(C_2))$ using the procedure described in the proof of Theorem 1. If the check succeeds then there is no deadlock in $L_1 \parallel L_2$ and we exit successfully. Otherwise, we proceed to Step 3.
3) From the counterexample obtained above construct $t \in \Sigma^*$ and $R \in \widehat{\Sigma}$ be such that $(t, R) \in W(\mathcal{L}(C_1))$ and $(t, \Sigma \setminus R) \in W(\mathcal{L}(C_2))$. Check if $(t, R) \in L_1$ and $(t, \Sigma \setminus R) \in L_2$. 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 $(t, R) \notin L_1$. But then, from **PROP**, $(t, \Sigma \setminus R) \in W(L_1)$. Again from **PROP**, since $(t, R) \in W(\mathcal{L}(C_1))$, $(t, \Sigma \setminus R) \notin \mathcal{L}(C_1)$. 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 Micro-C 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 $W(\mathcal{L}(C_1)) \parallel W(\mathcal{L}(C_2)) \subseteq \overline{L_{Dlk}}$) 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

[1] W. P. de Roever, H. Langmaack, and A. Pnueli, Eds., *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Revised Lectures*, ser. LNCS, vol. 1536. Springer, 1998.

[2] K. McMillan, "A compositional rule for hardware design refinement," in *Proc. of CAV*, 1997.

[3] O. Grumberg and D. E. Long, "Model checking and modular verification," *TOPLAS*, vol. 16, no. 3, pp. 843–871, May 1994.

[4] A. Pnueli, "In transition from global to modular temporal reasoning about programs," *Logics and models of concurrent systems*, vol. 13, pp. 123–144, 1985.

[5] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu, "Learning assumptions for compositional verification," in *Proc. of TACAS*, 2003, pp. 331–346.

[6] D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer, "Assumption generation for software component verification," in *Proc. of ASE*, 2002.

[7] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987.

[8] S. Chaki, E. M. Clarke, N. Sinha, and P. Thati, "Automated assume-guarantee reasoning for simulation conformance," in *Proc. of CAV*, 2005, pp. 534–547.

[9] C. A. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[10] S. Chaki, J. Ivers, N. Sharygina, and K. Wallnau, "The ComFoRT reasoning framework," in *Proc. of CAV*, 2005, pp. 164–169.

[11] D. Peled, M. Vardi, and M. Yannakakis, "Black box checking," in *Proc. of FORTE/PSTV*, October 1999.

[12] A. Groce, D. Peled, and M. Yannakakis, "Adaptive model checking," in *Proc. of TACAS*, 2002, pp. 357–370.

[13] R. Alur, P. Cerný, P. Madhusudan, and W. Nam, "Synthesis of interface specifications for java classes." in *Proc. of POPL*, 2005, pp. 98–109.

[14] P. Habermehl and T. Vojnar, "Regular model checking using inference of regular languages," in *Proc. of INFINITY'04*, 2004.

[15] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proc. of ICSE*, 1999.

[16] R. Alur, P. Madhusudan, and W. Nam, "Symbolic compositional verification by learning assumptions," in *Proc. of CAV*, 2005.

[17] H. Barringer, D. Giannakopoulou, and C. S. Păsăreanu, "Proof rules for automated compositional verification," in *Proc. of the 2nd Workshop on SAVCBS*, 2003.

[18] A. Overkamp, "Supervisory control using failure semantics and partial specifications," *Automatic Control, IEEE Transactions on*, vol. 42, no. 4, pp. 498–510, April 1997.

[19] A. Williams, W. Thies, and M. D. Ernst, "Static deadlock detection for Java libraries," in *Proc. of ECOOP*, 2005, pp. 602–629.

[20] V. Kahlon, F. Ivancic, and A. Gupta, "Reasoning about threads communicating via locks." in *Proc. of CAV*, 2005, pp. 505–518.

[21] G. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[22] C. Fournet, C. A. R. Hoare, S. K. Rajamani, and J. Rehof, "Stuck-free conformance." in *CAV*, 2004, pp. 242–254.

[23] S. D. Brookes and A. W. Roscoe, "Deadlock analysis in networks of communicating processes." *Distributed Computing*, vol. 4, pp. 209–230, 1991.

[24] N. Amla, E. A. Emerson, K. S. Namjoshi, and R. J. Trefler, "Abstract patterns of compositional reasoning." in *Proc. of CONCUR*, 2003, pp. 423–438.

[25] S. Chaki and N. Sinha, "Assume-guarantee reasoning for deadlock," Software Engineering Institute, Pittsburgh, PA, Technical note CMU/SEI-2006-TN-028, 2006.

[26] A. W. Roscoe, *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[27] D. Kozen, "Automata on guarded strings and applications," in *Matematica Contempornea 24 (2003)*, pp. 117–139.