Software Engineering Institute

# Assume-Guarantee Reasoning for Deadlock

Sagar Chaki, Software Engineering Institute

Nishant Sinha, Carnegie Mellon University

**September 2006**

**TECHNICAL NOTE**
CMU/SEI-2006-TN-028

**Predictable Assembly from Certifiable Components Initiative**


Carnegie Mellon

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.



This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html

# Table of Contents

# List of Figures

# Abstract

The use of learning to automate assume-guarantee style reasoning has received a lot of attention in recent years. This paradigm has already been used successfully for checking trace containment, as well as simulation between concurrent systems and their specifications. In this report, the learning-based automated assume-guarantee paradigm is extended to perform compositional deadlock detection. *Failure automata* is defined as a generalization of finite automata that accept regular failure sets. A learning algorithm $L^F$ is developed that constructs the minimal deterministic failure automaton accepting any unknown regular failure set using a minimally adequate teacher. This report shows how $L^F$ can be used for compositional regular failure language containment and deadlock detection, using non-circular and circular assume-guarantee rules. Finally, an implementation of techniques and encouraging experimental results on several nontrivial benchmarks are presented.

# 1 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 [de Roever 98, McMillan 97, Grumberg 94] is recognized to be one of the most promising approaches for alleviating statespace explosion. This report presents an automated compositional deadlock detection procedure based on assume-guarantee (AG) [Pnueli 85] reasoning.

AG reasoning revolves around a proof rule that relates system components and assumptions about them to global system properties. Typically, to apply the proof rule, you need to construct manually appropriate assumptions that can discharge the premises of the rule. In most realistic situations, suitable assumptions are complicated. The absence of automated assumption-generation techniques has held back the wider practical adoption of AG reasoning.

An important breakthrough has been the use of learning algorithms for assumption construction [Cobleigh 03]. The general idea is to learn an automaton corresponding to the *weakest assumption* [Giannakopoulou 02] 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 [Cobleigh 03], the weakest assumptions are naturally represented as deterministic *finite* automata, and this leads to the use of the $L^*$ learning algorithm [Angluin 87]. Similarly, in the case of simulation [Chaki 05a], the corresponding choices are deterministic *tree* automata and the $L^T$ learning algorithm.

Nonetheless, neither learning algorithm is appropriate for deadlock detection. Word and tree automata are unable to capture *failures* [Hoare 85], a critical concept for understanding and detecting deadlocks. While you can transform any deadlock detection problem to an ordinary trace containment, such schemes invariably introduce new components and an exponential number of actions. As a result, these strategies are not scalable. Our work started with a search for an appropriate automata-theoretic formalism that can handle failures directly. Our deadlock detection algorithm uses learning-based automated AG reasoning and does not require additional actions or components.

Two key parts of our solution are: (1) a new type of acceptors for regular failure languages (RFLs) with a non-standard accepting condition and (2) a notion of parallel composition between these acceptors that is consistent with the parallel composition of the languages they accept. Our accepting condition is novel and employs a notion of maximality that crucially avoids introducing 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 report presents the first use of learning in the context of automated AG reasoning for deadlock detection.

In Section 2, we present the theory of *regular* failure languages 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 but not under complementation, which is an acceptable tradeoff for the use of maximality. Further, we show a Myhill-Nerode-like theorem for RFLs and failure automata.

We show, in Section 3, that the failure language of a labeled transition system (LTS) $M$ is regular and checking deadlock freedom for $M$ is a particular instance of the problem of checking RFL containment. Then, we present an algorithm for checking containment of RFLs. We cannot check containment between failure languages $L_1$ and $L_2$ by complementing $L_2$ and intersecting with $L_1$, since (as we noted above) RFLs are not closed under complementation.

In Section 4, we present a sound and complete non-circular AG rule, **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 $L_A$ such that $L_1 \parallel L_A \subseteq L_S$, $L_A \subseteq L_W$. We then show constructively that if failure languages $L_1$ and $L_2$ are regular, then $L_W$ uniquely exists, is regular, and is accepted by a minimum failure automaton $A_W$.

Section 5 details the development of an algorithm $L^F$ 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.

In Section 6, we develop an automated and compositional deadlock detection algorithm that employs **AG-NC** and $L^F$.

Section 7 defines a circular AG proof rule **AG-Circ** for deadlock detection and shows how to use it for automated and compositional deadlock detection.

As we show in Section 8, we have implemented our approach in the ComFoRT [Chaki 05b] reasoning framework. We present encouraging results on several nontrivial benchmarks, including an embedded OS and Linux device drivers.

Finally, Section 9 summarizes our conclusions.

# 2 Related Work

Machine-learning techniques have been used in several contexts related to verification [Peled 99, Groce 02, Alur 05a, Habermehl 05, Ernst 99]. We follow the approach of Cobleigh, Giannakopoulou, and Păsăreanu [Cobleigh 03] (respectively Chaki and colleagues [Chaki 05a]) to automate assume-guarantee reasoning for trace-containment (or simulation) between finite state systems.[1] However, we apply this general paradigm for deadlock detection. This $L^F$ algorithm may also be of independent interest. Rivest and Schapire proposed an improvement to Angluin's $L^*$ that substantially improves its complexity [Rivest 93]. $L^F$ has the same spirit as this improved version of $L^*$. The use of circular AG rules was also investigated in the context of trace containment by Barringer, Giannakopoulou, and Păsăreanu [Barringer 03].

Overkamp explored the synthesis of supervisory controller for discrete-event systems [Overkamp 97] based on failure semantics [Hoare 85]. 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 as follows: (1) We use failure automata to represent failure traces; (2) We use learning to compute the weakest failure assumption automatically; and (3) Our focus is on checking deadlocks in software modules. Williams, Thies and Ernst investigated an approach based on static analysis for detecting deadlocks that incorrect lock manipulation by Java programming language libraries can cause [Williams 05].[2] The problem of detecting deadlocks for pushdown programs communicating only via nested locking has been investigated by Kahlon, Ivancic and Gupta [Kahlon 05]. 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 [Holzmann 03] and for checking refinement of CCS processes using a notion called stuck-free conformance that's more discriminative than failure trace refinement [Fournet 04].

Brookes and Roscoe use the failure model to show the absence of deadlock in undirectional networks [Brookes 91]. 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, to analyze the deadlock freedom of a set of concurrent programs compositionally, we use both circular and non-circular assume-guarantee rules [Pnueli 85, de Roever 98, Barringer 03]. Amla and colleagues have presented a sound and complete assume-guarantee method in the context of an abstract process composition framework [Amla 03]. However, they do not discuss deadlock detection or explore the use of learning.

[1] Alur, Madhusudan, and Nam have also investigated symbolic learning in this context [Alur 05b].

[2] Williams, Thies and Ernst also also provide an excellent survey of related research [Williams 05].

# 3 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 (RFLs) and failure automata (FLA), analogous to Myhill-Nerode theorem for ordinary regular languages. We begin with a few standard definitions [Roscoe 97].

**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 $s \xrightarrow{\alpha} s'$ to mean $(s, \alpha, s') \in \delta$. A trace is any finite (possibly empty) sequence of actions, that is, the set of all traces is $\Sigma^*$. We denote an empty trace by $\epsilon$, a singleton trace $\langle\alpha\rangle$ by $\alpha$, and the concatenation of two traces $t_1$ and $t_2$ by $t_1 \bullet t_2$. For any LTS $M = (S, Init, \Sigma, \delta)$, we define the function $\widehat{\delta} : 2^S \times \Sigma^* \rightarrow 2^S$ as follows:

$$\widehat{\delta}(X, \epsilon) = X \quad \text{and} \quad \widehat{\delta}(\mathrm{X}, \mathrm{t} \bullet \alpha) = \left\{ \mathrm{s}' | \exists \mathrm{s} \in \widehat{\delta}(\mathrm{X}, \mathrm{t}) \centerdot \mathrm{s} \xrightarrow{\alpha} \mathrm{s}' \right\}$$

$M$ is said to be deterministic if $|Init| = 1$ and $\forall s \in S \centerdot \forall \alpha \in \Sigma \centerdot |\widehat{\delta}(\{s\}, \alpha)| \leq 1$ and complete if $\forall s \in S \centerdot \forall \alpha \in \Sigma \centerdot |\widehat{\delta}(\{s\}, \alpha)| \geq 1$. Thus if $M$ is both deterministic and complete then $|Init| = 1$ and $\forall s \in S \centerdot \forall t \in \Sigma^* \centerdot |\widehat{\delta}(\{s\}, t)| = 1$. In this case, we write $\widehat{\delta}(s, t)$ to mean the only element of $\widehat{\delta}(\{s\}, t)$.

**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 $\forall s' \in S \centerdot (s, \alpha, s') \notin \delta$. 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) $\forall s \centerdot Ref(s, \emptyset)$ and (ii) $\forall s, R, R' \centerdot Ref(s, R) \wedge R' \subseteq R \implies Ref(s, R')$ (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 $2^\Sigma$ by $\widehat{\Sigma}$. Note that $L \subseteq \Sigma^* \times \widehat{\Sigma}$.

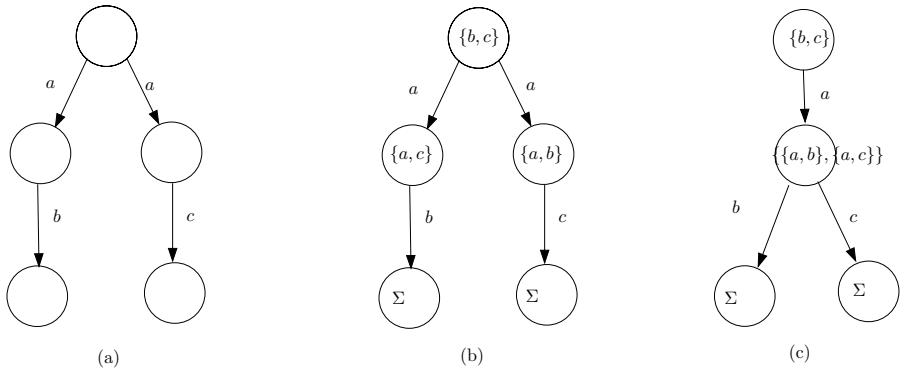


*Figure 1: (a) LTS $M$ on $\Sigma = \{a, b, c\}$, (b) its FLA, and (c) its deterministic FLA. All states of FLAs are accepting.*

The union and intersection of failure languages are 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 the following holds:

$$\forall t \in \Sigma^* \,.\, \forall R \in \widehat{\Sigma} \,.\, (t, R) \in L \implies \forall R' \subseteq R.\ (t, R') \in L$$

In general, failure languages may not be downward closed; however, we will show later that failure languages generated from LTSs are always downward closed because the refusal sets at each state of an LTS are downward closed. In this report, we focus on downward-closed failure languages, in particular, *regular* failure languages.

**Definition 5 (Deadlock)** *An LTS $M$ is said to deadlock iff $\mathcal{F}(M) \cap (\Sigma^* \times \{\Sigma\}) \neq \emptyset$. 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 $\widehat{\Sigma}$. The set of maximal elements of $P$ is denoted by $Max(P)$ and defined as follows: $Max(P) = \{R \in P \mid \forall R' \in P \,.\, R \not\subset R'\}$.

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 $R \subseteq R'$. With the concept of maximality, we use only the upper bounds of a set (according to subset partial order) to represent the complete set and thereby concisely represent downward-closed failed languages.

**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 $\mu : F \to 2^{\widehat{\Sigma}}$ is a mapping from the final states to $2^{\widehat{\Sigma}}$ such that: $\forall s \in F \,.\, \mu(s) \neq \emptyset \wedge \mu(s) = Max(\mu(s))$.*

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). Part (a) of Figure 1 shows an LTS over $\Sigma$=$\{a, b, c\}$. Parts (b) and (c) show the corresponding FLA and its deterministic version, respectively.

**Definition 7 (Language of an FLA)** *Let $A = (M, F, \mu)$ be an FLA such that $M = (S, Init, \Sigma, \delta)$. Then, a failure $(t, R)$ is accepted by $A$ iff the following holds:*

$$\exists s \in F \,.\, \exists R' \in \mu(s) \,.\, s \in \widehat{\delta}(Init, t) \wedge R \subseteq R'$$

*The language of $A$, denoted by $\mathcal{L}(A)$, is the set of all failures accepted by $A$.*

Every deterministic FLA (DFLA) $A$ can be extended to a complete DFLA $A'$ such that $\mathcal{L}(A') = \mathcal{L}(A)$ by adding a non-final *sink* state. In the rest of this report, we consider FLA and languages over a fixed alphabet $\Sigma$.[3]

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

*Proof.* By subset construction. Let $L$ be a language accepted by some FLA $A = (M, F, \mu)$. We construct a deterministic FLA $A' = (M', F', \mu')$ as follows: The deterministic finite automaton $G' = (M', F')$ is obtained by the standard subset construction from the finite automaton $G = (M, F)$. For any state $s'$ of $M'$, let us denote $\Psi(s')$ as the set of states of $M$ from which $s'$ was derived by the subset construction. To define $\mu'$, consider any final state $s' \in F'$. We know that $\Psi(s') \cap F \neq \emptyset$. Let $P = \bigcup_{s \in \Psi(s') \cap F} \mu(s)$. Then $\mu'(s') = Max(P)$.

Let $Init$ and $Init'$ be the initial states of $M$ and $M'$, respectively. Now, to show that $\mathcal{L}(A') = L$, consider any failure $(t, R)$. Then

$$(t, R) \in \mathcal{L}(A') \iff \exists s' \in \widehat{\delta}(Init', t) \cap F' \,.\, \exists R' \in \mu'(s') \,.\, R \subseteq R'$$

$$\iff \exists s' \in \widehat{\delta}(Init', t) \cap F' \,.\, \exists s \in \Psi(s') \cap F \,.\, \exists R' \in \mu(s) \,.\, R \subseteq R'$$

$$\iff \exists s \in \widehat{\delta}(Init, t) \cap F \,.\, \exists R' \in \mu(s) \,.\, R \subseteq R' \iff (t, R) \in \mathcal{L}(A) = L$$

**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.[4] In addition, every RFL 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.

[3] FLA are closely related to automata on guarded strings [Kozen 01], 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 an FLA.

[4] 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.

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

*Proof.* Our proof follows that of the Myhill-Nerode theorem for finite automata. Let $L$ be any RFL. Let us define an equivalence relation $\equiv$ over $\Sigma^*$ as follows:

$$u \equiv v \iff \forall (t, R) \in \Sigma^* \times \widehat{\Sigma} \,.\, (u \bullet t, R) \in L \iff (v \bullet t, R) \in L$$

For any $u \in \Sigma^*$, we denote the equivalence class of $u$ by $[u]$. Let us define a finite automaton $G = (M, F)$ where $M = (S, Init, \Sigma, \delta)$ such that: (i) $S = \{[u] \mid u \in \Sigma^*\}$, (ii) $Init = \{[\epsilon]\}$, (iii) $\forall u \in \Sigma^* \,.\, \forall \alpha \in \Sigma \,.\, [u] \xrightarrow{\alpha} [u \bullet \alpha]$, and (iv) $F = \left\{ [u] \mid \exists R \in \widehat{\Sigma} \,.\, (u, R) \in L \right\}$.

Also, let us define a function $\mu$ as follows: Consider any $[u] \in F$ and let $P \subseteq \widehat{\Sigma}$ be defined as $P = \{R \mid \exists v \,.\, v \equiv u \wedge (v, R) \in L\}$. Note that since $[u] \in F$, $P \neq \emptyset$. Then $\mu([u]) = Max(P)$. Let $A$ be the FLA $(M, F, \mu)$.

We first show by contradiction that $A$ is deterministic. First, note that $|Init| = 1$. Next, suppose that $A$ is nondeterministic. Then there exists two traces $u \in \Sigma^*$ and $v \in \Sigma^*$ and an action $\alpha \in \Sigma$ such that $u \equiv v$ but $u \bullet \alpha \not\equiv v \bullet \alpha$. Then there exists a failure $(t, R)$ such that $(u \bullet \alpha \bullet t, R) \in L \iff (v \bullet \alpha \bullet t, R) \notin L$. But then there exists a failure $(t', R) = (\alpha \bullet t, R)$ such that $(u \bullet t', R) \in L \iff (v \bullet t', R) \notin L$. This implies that $u \not\equiv v$ which is a contradiction.

Next, we show that: **(C1)** for any trace $t$, $\widehat{\delta}(Init, t) = [t]$. The proof proceeds by induction on the length of $t$. For the base case, suppose $t = \epsilon$. Then $\widehat{\delta}(Init, t) = Init = [\epsilon]$. Now suppose $t = t' \bullet \alpha$ for some trace $t'$ and action $\alpha$. By the inductive hypothesis, $\widehat{\delta}(Init, t') = [t']$. Also, from the definition of $A$, we know that $[t'] \xrightarrow{\alpha} [t' \bullet \alpha]$. Hence, $\widehat{\delta}(Init, t) = \widehat{\delta}(Init, t' \bullet \alpha) = [t' \bullet \alpha] = [t]$. This completes the proof.

Now consider any DFLA $A' = (M', F', \mu')$ where $M' = (S', Init', \Sigma, \delta')$ such that $\mathcal{L}(A') = L$. Let us define a function $\Omega : S' \to S$ as follows: $\forall t \in \Sigma^* \,.\, \Omega(\widehat{\delta}(Init', t)) = \widehat{\delta}(Init, t)$. First we show that $\Omega$ is well-defined. Consider any two traces $u$ and $v$ such that $\widehat{\delta}(Init', u) = \widehat{\delta}(Init', v)$. Then for any failure $(t, R)$, $A'$ accepts $(u \bullet t, R)$ iff it also accepts $(v \bullet t, R)$. Since $A'$ accepts $L$, we find that $u \equiv v$. Combining this equality with **C1** above we have $\widehat{\delta}(Init, u) = [u] = [v] = \widehat{\delta}(Init, v)$. Therefore, $\widehat{\delta}(Init, u) = \widehat{\delta}(Init, v)$ which proves that $\Omega$ is well-defined. In addition, $\Omega$ is a surjection since for any state $[u]$ of $A$ we have the following from **C1** above: $[u] = \widehat{\delta}(Init, u) = \Omega(\widehat{\delta}(Init', u))$.

We are now ready to prove the main result. In essence, we show that $A$ is the unique minimal DFLA that accepts $L$. We have already shown that $A$ is deterministic. To show that $\mathcal{L}(A) = L$ we observe that for any trace $t$ and any refusal $R$, the following holds:

$$(t, R) \in L \iff [t] \in F \wedge \exists R' \in \mu([t]) \,.\, R \subseteq R' \iff (t, R) \in \mathcal{L}(A)$$

Next, recall that $\Omega$ defined above is a surjection. Hence, $A'$ must have at least as many states as $A$. Since $A'$ is an arbitrary DFLA accepting $L$, $A$ must be a minimal DFLA that accepts $L$. To show that $A$ is unique up to isomorphism, let $A'$ be another minimal DFLA accepting $L$. In this case, $\Omega$ must be a bijection. We show that $\Omega$ is also an isomorphism.

Let us write $\Omega^{-1}$ to mean the inverse of $\Omega$. Note that $\Omega^{-1}$ is also a bijection, and more specifically, $\forall t \in \Sigma^* \,.\, \Omega^{-1}([t]) = \Omega^{-1}(\widehat{\delta}(Init, t)) = \widehat{\delta}(Init', t)$. We will now prove the following statements:

$$(\mathbf{C2})\ \Omega^{-1}(Init) = Init'$$

$$(\mathbf{C3})\ \forall u \in \Sigma^* \,.\, \forall v \in \Sigma^* \,.\, \forall \alpha \in \Sigma \,.\, [u] \xrightarrow{\alpha} [v] \iff \Omega^{-1}([u]) \xrightarrow{\alpha} \Omega^{-1}([v])$$

$$(\mathbf{C4})\ \forall s \in S \,.\, s \in F \iff \Omega^{-1}(s) \in F'$$

$$(\mathbf{C5})\ \forall s \in F \,.\, \mu(s) = \mu'(\Omega^{-1}(s))$$

First, **C2** holds since $\Omega^{-1}(Init) = \Omega^{-1}(\widehat{\delta}(Init, \epsilon)) = \widehat{\delta}(Init', \epsilon) = Init'$. To prove **C3**, suppose that $[u] \xrightarrow{\alpha} [v]$. Since $[u] = \widehat{\delta}(Init, u)$ we have $[v] = \widehat{\delta}(Init, u \bullet \alpha)$. Hence, $\Omega^{-1}([u]) = \widehat{\delta}(Init', u)$ and $\Omega^{-1}([v]) = \widehat{\delta}(Init', u \bullet \alpha)$. But this implies that $\Omega^{-1}([u]) \xrightarrow{\alpha} \Omega^{-1}([v])$, which proves the forward implication. For the reverse implication suppose that $\Omega^{-1}([u]) \xrightarrow{\alpha} \Omega^{-1}([v])$. Since $\Omega^{-1}([u]) = \widehat{\delta}(Init', u)$ we again have $\Omega^{-1}([v]) = \widehat{\delta}(Init', u \bullet \alpha)$. Therefore, $[u] = \widehat{\delta}(Init, u)$ and $[v] = \widehat{\delta}(Init, u \bullet \alpha)$, and hence $[u] \xrightarrow{\alpha} [v]$.

To prove **C4**, consider any $s \in S$ such that $s = [u] = \widehat{\delta}(Init, u)$. Hence, $\Omega^{-1}(s) = \Omega^{-1}([u]) = \widehat{\delta}(Init', u)$. Then

$$s \in F \iff [u] \in F \iff \exists R \,.\, (u, R) \in L \iff \widehat{\delta}(Init', u) \in F' \iff \Omega^{-1}(s) \in F'$$

Finally, we prove **C5** by contradiction. Suppose that there exists $s = [u] \in F$ such that $\mu(s) \neq \mu'(\Omega^{-1}(s))$. Without loss of generality, we can always pick a refusal $R$ such that $\exists R' \in \mu(s) \,.\, R \subseteq R'$ and $\forall R' \in \mu'(\Omega^{-1}(s)) \,.\, R \not\subseteq R'$. Now, we also know that $s = \widehat{\delta}(Init, u)$ and $\Omega^{-1}(s) = \widehat{\delta}(Init', u)$. Therefore, $(u, R) \in \mathcal{L}(A) \setminus \mathcal{L}(A')$, which implies that $\mathcal{L}(A) = L \neq L = \mathcal{L}(A')$, a contradiction.

Note that for any LTS $M$, $\mathcal{F}(M)$ is regular.[5] Indeed, the failure automaton corresponding to $M = (S, Init, \Sigma, \delta)$ is $A = (M, S, \mu)$ such that $\forall s \in S \,.\, \mu(s) = Max(\{R \mid Ref(s, R)\})$.

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

# 4 Assume Guarantee for Deadlock

We now present an assume-guarantee style [Pnueli 85] proof rule for deadlock detection in systems composed of two components. We use the notion of parallel composition proposed in the theory of CSP [Hoare 85] 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 $\Sigma$. 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 [Chaki 05a] with two components, each having the same alphabet $\Sigma_1 \cup \Sigma_2$. Thus, all languages and automata we consider here 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 lemma are detailed in Section 5.

**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:*

$$(L_1 \subseteq L'_1) \wedge (L_2 \subseteq L'_2) \implies (L_1 \parallel L_2) \subseteq (L'_1 \parallel L'_2)$$

*Proof.* Let $(t, R)$ be any failure in $(L_1 \parallel L_2)$. Then there exists refusals $R_1$ and $R_2$ such that: **(A)** $R = R_1 \cup R_2$, **(B)** $(t, R_1) \in L_1$ and **(C)** $(t, R_2) \in L_2$. From **(B)**, **(C)**, and the premise of the lemma, we have **(D)** $(t, R_1) \in L'_1$ and **(E)** $(t, R_2) \in L'_2$. But then from **(A)**, **(D)**, **(E)** and Definition 9, we have $(t, R) \in (L'_1 \parallel L'_2)$, which completes the proof.

**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* $A_1 \amalg A_2$*,*[6] *is defined as the FLA* $(M_1 \amalg M_2, F_1 \times F_2, \mu)$ *such that:*

$$\mu(s_1, s_2) = Max(\{R_1 \cup R_2 \mid R_1 \in \mu_1(s_1) \wedge R_2 \in \mu_2(s_2)\})$$

[6] 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 its context.

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.

**Lemma 4** $\mathcal{F}(M_1 \amalg M_2) = \mathcal{F}(M_1) \parallel \mathcal{F}(M_2)$.

*Proof.* For any LTSs $M_1$ and $M_2$ over the same alphabet $\Sigma$, it can be proved that

$$\mathcal{F}(M_1 \amalg M_2) = \{(t, R_1 \cup R_2) \mid (t, R_1) \in \mathcal{F}(M_1) \wedge (t, R_2) \in \mathcal{F}(M_2)\}$$

The lemma then follows from the above fact and Definition 9.

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

*Proof.* Let $A_1 = (M_1, F_1, \mu_1)$ and $A_2 = (M_2, F_2, \mu_2)$ where $M_1 = (S_1, Init_1, \Sigma, \delta_1)$ and $M_2 = (S_2, Init_2, \Sigma, \delta_2)$. Then we know that $A_1 \amalg A_2 = (M_1 \amalg M_2, F_1 \times F_2, \mu)$. Let $(t, R)$ be any element of $\mathcal{L}(A_1 \amalg A_2)$. Then, we know that

$$\exists (s_1, s_2) \in \widehat{\delta}(Init_1 \times Init_2, t) \cap F_1 \times F_2 \centerdot \exists R' \in \mu(s_1, s_2) \centerdot R \subseteq R'$$

From the definition of $\mu$, we find that

$$\exists R_1 \in \mu_1(s_1) \centerdot \exists R_2 \in \mu_2(s_2) \centerdot R \subseteq R_1 \cup R_2$$

Therefore, $(t, R_1) \in \mathcal{L}(A_1)$, $(t, R_2) \in \mathcal{L}(A_2)$, and $(t, R) \in \mathcal{L}(A_1) \parallel \mathcal{L}(A_2)$. This statement proves that $\mathcal{L}(A_1 \amalg A_2) \subseteq \mathcal{L}(A_1) \parallel \mathcal{L}(A_2)$. Now, let $(t, R)$ be any element of $\mathcal{L}(A_1) \parallel \mathcal{L}(A_2)$. Then, we know that

$$\exists s_1 \in \widehat{\delta}(Init_1, t) \cap F_1 \centerdot \exists s_2 \in \widehat{\delta}(Init_2, t) \cap F_2 \centerdot \exists R_1 \in \mu_1(s_1) \centerdot \exists R_2 \in \mu_2(s_2) \centerdot R \subseteq R_1 \cup R_2$$

Therefore, $(s_1, s_2) \in \widehat{\delta}(Init_1 \times Init_2, t) \cap F_1 \times F_2$ and $\exists R' \in \mu(s_1, s_2) \centerdot R \subseteq R'$. Hence $(t, R) \in \mathcal{L}(A_1 \amalg A_2)$. This show that $\mathcal{L}(A_1) \parallel \mathcal{L}(A_2) \subseteq \mathcal{L}(A_1 \amalg A_2)$ and completes the proof.

**Regular Failure Language Containment (RFLC).** We develop a general compositional framework for checking RFLC. This framework is also applicable to deadlock detection since, as we will show later, deadlock freedom is a form of RFLC. Recall that RFLs are not closed under complementation. 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 that corresponds 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:*

$$s_1 \in F_1 \implies (s_2 \in F_2 \wedge (\forall R_1 \in \mu_1(s_1) \centerdot \exists R_2 \in \mu_2(s_2) \centerdot R_1 \subseteq R_2))$$

*Proof.* First, we note that $\mathcal{L}(A_1) = \mathcal{L}(A_1')$ and $\mathcal{L}(A_2) = \mathcal{L}(A_2')$. Now let $M_1 = (S_1, Init_1, \Sigma, \delta_1)$ and $M_2 = (S_2, Init_2, \Sigma, \delta_2)$. For the forward implication, we prove the contrapositive. Suppose that there exists a reachable state $(s_1, s_2)$ of $M_1 \amalg M_2$ such that $s_1 \in F_1$ and either $s_2 \notin F_2$ or $\exists R_1 \in \mu_1(s_1) \centerdot \forall R_2 \in \mu_2(s_2) \centerdot R_1 \not\subseteq R_2$. Since $M_1$ and $M_2$ are deterministic, let $t \in \Sigma^*$ be a trace such that $(s_1, s_2) = \widehat{\delta}(Init_1 \times Init_2, t)$. Now, we choose a refusal $R$ as follows. If $s_2 \notin F_2$ then let $R$ be any element of $\mu_1(s_1)$. Otherwise let $R$ be some $R_1 \in \mu_1(s_1)$ such that $\forall R_2 \in \mu_2(s_2) \centerdot R_1 \not\subseteq R_2$. Now, it follows that $(t, R) \in \mathcal{L}(A_1') \setminus \mathcal{L}(A_2')$. Hence $\mathcal{L}(A_1') \not\subseteq \mathcal{L}(A_2')$ and therefore $\mathcal{L}(A_1) \not\subseteq \mathcal{L}(A_2)$.

For the reverse implication we also prove the contrapositive. Suppose $\mathcal{L}(A_1) \not\subseteq \mathcal{L}(A_2)$ and let $(t, R)$ be any element of $\mathcal{L}(A_1) \setminus \mathcal{L}(A_2) = \mathcal{L}(A_1') \setminus \mathcal{L}(A_2')$. Let $s_1 = \widehat{\delta}(Init_1, t)$ and $s_2 = \widehat{\delta}(Init_2, t)$. But we know that $\exists R_1 \in \mu_1(s_1) \centerdot R \subseteq R_1$ and either $s_2 \notin F_2$ or $\forall R_2 \in \mu_2(s_2) \centerdot R \not\subseteq R_2$. However, this implies that $s_1 \in F_1$ and either $s_2 \notin F_2$ or $\exists R_1 \in \mu_1(s_1) \centerdot \forall R_2 \in \mu_2(s_2) \centerdot R_1 \not\subseteq R_2$. In addition, $(s_1, s_2)$ is a reachable state of $M_1 \amalg M_2$. This completes the proof.

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. In essence, the condition 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. To check RFLC between $L_1 \parallel L_2$ and $L_S$, the approach presented in Lemma 6 requires 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 RFLC.** 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, this 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, we find that 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 $A_1 \amalg A_2$ (see Lemma 10) and then check if $\mathcal{L}(A_1 \amalg A_2) \subseteq \mathcal{L}(A_S)$ (see Lemma 5 and 6). The problem with this naive approach is statespace explosion. To alleviate this problem, we present a compositional language containment scheme based on AG-style reasoning in the next section.

## 4.1 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, called **AG-NC**, is both sound and complete:

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

*Proof.* The completeness of **AG-NC** follows from the fact that if the conclusion holds, then $L_2$ can be used as $L_A$ to discharge the two premises. To prove soundness, let us assume that the two premises hold. Then from the second premise and Lemma 3, we have $L_1 \parallel L_2 \subseteq L_1 \parallel L_A$. Combining this statement with the first premise, we get $L_1 \parallel L_2 \subseteq L_S$ which is the desired conclusion.

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, we are left with two critical problems. First, it provides no effective method for constructing an appropriate assumption $L_A$. Second, if no appropriate assumption exists; that is, 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 report we develop and employ a learning algorithm that solves both the above problems. 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 one of the following two situations in a finite number of steps and to 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 $CE$.

We present complete details of our algorithm, as well as its complexity, later in Section 5. First we discuss formally the notion of the weakest assumption $L_W$.

## 4.2 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.* We first prove (i) by contradiction. Suppose there exists $(t, R_1) \in L_1$ and $(t, R_2) \in L_W$ such that $(t, R_1 \cup R_2) \notin L_S$. But then $(t, R_1 \cup R_2) \in L_1 \parallel \{(t, R_2)\}$ which means $L_1 \parallel \{(t, R_2)\} \not\subseteq L_S$. However, this contradicts $(t, R_2) \in L_W$.

Now, we only prove the forward implication of (ii). The reverse implication follows from (i) and Lemma 3. This proof is also by contradiction. Suppose there exists a language $L$ such that $L_1 \parallel L \subseteq L_S$ and $L \not\subseteq L_W$. Then there exists some $(t, R_2) \in L \setminus L_W$. But since $(t, R_2) \notin L_W$, there exists $(t, R_1) \in L_1$ such that $(t, R_1 \cup R_2) \notin L_S$. However, this means that $(t, R_1 \cup R_2) \in L_1 \parallel L$, which contradicts $L_1 \parallel L \subseteq L_S$.

Finally, to prove that $L_W$ is regular we construct an FLA $A_W$ such that $\mathcal{L}(A_W) = L_W$. Let $A_1 = (M_1, F_1, \mu_1)$ and $A_S = (M_S, F_S, \mu_S)$ be deterministic and complete FLA accepting $L_1$ and $L_S$ respectively such that $M_1 = (S_1, Init_1, \Sigma, \delta_1)$ and $M_S = (S_S, Init_S, \Sigma, \delta_S)$. Then $A_W = (M_1 \amalg M_S, F_W, \mu_W)$. To define the set of final states $F_W$ and the labeling function $\mu_W$ of $A_W$, we define the extended labeling function $\widehat{\mu} : S \to 2^{\widehat{\Sigma}}$ of any FLA as follows: $\widehat{\mu}(s) = \mu(s)$ if $s$ is a final state and $\emptyset$ otherwise. Then the extended labeling function $\widehat{\mu}$ of $A_W$ is defined as follows:

$$\widehat{\mu}(s_1, s_S) = \left\{ R \in \widehat{\Sigma} \mid \forall R_1 \in \widehat{\mu}(s_1) \centerdot \exists R_S \in \widehat{\mu}(s_S) \centerdot (R_1 \cup R) \subseteq R_S \right\}$$

Note that the set $\widehat{\mu}(s_1, s_S)$ is always downward-closed. In other words

$$\forall R \in \widehat{\Sigma} \centerdot \forall R' \in \widehat{\Sigma} \centerdot R \in \widehat{\mu}(s_1, s_S) \wedge R' \subseteq R \implies R' \in \widehat{\mu}(s_1, s_S)$$

Then the definitions of $F_W$ and $\mu_W$ follow naturally as below:

$$F_W = \{(s_1, s_S) \mid \widehat{\mu}(s_1, s_S) \neq \emptyset\}$$

$$\forall (s_1, s_S) \in F_W \centerdot \mu_W(s_1, s_S) = Max(\widehat{\mu}(s_1, s_S))$$

Note that since $A_1$ and $A_S$ are both deterministic and complete, so is $A_W$. Also, for any state $(s_1, s_S)$ of $A_W$ and any $t \in \Sigma^*$, we have $\widehat{\delta}((s_1, s_S), t) = (\widehat{\delta}(s_1, t), \widehat{\delta}(s_S, t))$. We now prove that $\mathcal{L}(A_W) = L_W$. Consider any failure $(t, R) \in (\Sigma^* \times \widehat{\Sigma})$. Let $(s_1, s_S) = \widehat{\delta}((Init_1, Init_S), t)$. We consider two sub-cases.

**Case 1** $[(t, R) \in \mathcal{L}(A_W)]$**.** Then we know that $R \in \widehat{\mu}(s_1, s_S)$. Now consider the language $L = L_1 \parallel \{(t, R)\}$. By Definition 9, any element of $L$ must be of the form $(t, R_1 \cup R)$ for some $R_1 \in \widehat{\mu}(s_1)$. Also, from the definition of $\widehat{\mu}$ above we have $\exists R_S \in \widehat{\mu}(s_S) \centerdot (R_1 \cup R) \subseteq R_S$. Hence $(t, R_1 \cup R) \in L_S$. Since $(t, R_1 \cup R)$ is an arbitrary element of $L$ we conclude that $L \subseteq L_S$. Hence, from the definition of $L_W$ above we have $(t, R) \in L_W$ which completes the proof of this subcase.

**Case 2** $[(t, R) \notin \mathcal{L}(A_W)]$**.** In this case, $R \notin \widehat{\mu}(s_1, s_S)$. Then, from the definition of $\widehat{\mu}$ above, we have $\exists R_1 \in \widehat{\mu}(s_1) \centerdot \forall R_S \in \widehat{\mu}(s_S) \centerdot (R_1 \cup R) \not\subseteq R_S$. Now consider the language $L = L_1 \parallel \{(t, R)\}$. By Definition 9, $(t, R_1 \cup R) \in L$. However, from $\forall R_S \in \widehat{\mu}(s_S) \centerdot (R_1 \cup R) \not\subseteq R_S$, we have $(t, R_1 \cup R) \notin L_S$. Hence, $L \not\subseteq L_S$. Thus, from the definition of $L_W$ above we have $(t, R) \notin L_W$, which completes the proof of this subcase and of the entire theorem.

Now that we have proved that the weakest environment assumption $L_W$ is regular, we can apply a learning algorithm to iteratively construct an FLA assumption that accepts $L_W$. In

particular, we develop a learning algorithm $L^F$ that iteratively learns the minimal DFLA corresponding to $L_W$. $L^F$ asks queries about $L_W$ to a minimally adequate teacher (MAT) and learns from the answers. In the next Section, we present $L^F$. Subsequently, in section 6, we describe how $L^F$ is used in our compositional language containment procedure. If you are only interested in the overall compositional deadlock-detection algorithm and not the intricacies of $L^F$, you should skip to Section 6.

# 5 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 DFLA $C$, the MAT returns TRUE if $\mathcal{L}(C) = U$. Otherwise, it returns FALSE along with a counterexample failure $CE \in (\mathcal{L}(C) \setminus U) \bigcup (U \setminus \mathcal{L}(C))$.

## 5.1 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 the following holds:

$$\forall t_1 \in \mathbb{T} \,.\, \forall t_2 \in \mathbb{T} \,.\, t_1 \neq t_2 \implies \mathbb{R}(t_1) \neq \mathbb{R}(t_2)$$

Essentially, this means that any two distinct rows $t_1$ and $t_2$ of a well-formed table can be distinguished by some experiment $e \in \mathbb{E}$. There is also 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 $|\mathbb{T}| \leq n$.*

*Proof.* The proof is by contradiction. Suppose that $|\mathbb{T}| > n$. Let the minimal DFLA accepting $U$ be $A$. Then there exists two distinct traces $t_1$ and $t_2$ in $\mathbb{T}$ such that $\widehat{\delta}(Init, t_1) = \widehat{\delta}(Init, t_2)$. In other words, the FLA $A$ reaches the same state on input $t_1$ and $t_2$. But since $\mathcal{T}$ is well-formed, there exists some failure $e = (t, p) \in \mathbb{E}$ such that $\mathbb{R}(t_1, e) \neq \mathbb{R}(t_2, e)$. In other words, $(t_1 \bullet t, p) \in U$ iff $(t_2 \bullet t, p) \notin U$. This case is impossible, since $A$ would reach the same state on inputs $t_1 \bullet t$ and $t_2 \bullet t$.

**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 $\forall t' \in \mathbb{T} \centerdot \mathbb{R}(t \bullet \alpha) \neq \mathbb{R}(t')$
  **add** $t \bullet \alpha$ to $\mathbb{T}$ and update $\mathbb{R}$ accordingly
**return** $\mathcal{T}$

*Figure 2: Algorithm* **MakeClosed**.

**Closed Observation Table.** An observation table $\mathcal{T} = (\mathbb{T}, \mathbb{E}, \mathbb{R})$ is said to be *closed* iff it satisfies the following:

$$\forall t \in \mathbb{T} \centerdot \forall \alpha \in \Sigma \centerdot \exists t' \in \mathbb{T} \centerdot \mathbb{R}(t \bullet \alpha) = \mathbb{R}(t')$$

Intuitively, if we extend any trace $t \in \mathbb{T}$ by any action $\alpha$, then the result is indistinguishable from an existing trace $t' \in \mathbb{T}$ under 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 extension can be achieved by the algorithm **MakeClosed** shown in Figure 2. 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 are required only if all possible refusals occur in $\mathbb{E}$.

## 5.2 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 $\mathbb{E} = \emptyset$. 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 so that it guarantees an upper bound on the total number of iterations of $L^F$. This construction 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).

## 5.3 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

- $S = \mathbb{T}$ : Each state of $M$ corresponds to a distinct row of $\mathcal{T}$.
- $Init = \epsilon$ : The initial state of $M$ corresponds to the empty trace. The empty trace always belongs to $\mathbb{T}$, since initially $\mathbb{T} = \{\epsilon\}$ and subsequently $\mathbb{T}$ grows monotonically.
- $\delta$ is constructed as follows: Consider any $t \in \mathbb{T}$ and $\alpha \in \Sigma$. Since $\mathcal{T}$ is well-formed and closed, we know that there exists an unique $t' \in \mathbb{T}$ such that $\mathbb{R}(t \bullet \alpha) = \mathbb{R}(t')$. Then, we add $t \xrightarrow{\alpha} t'$ to $\delta$. In other words

$$\delta = \left\{ t \xrightarrow{\alpha} t' \mid \mathbb{R}(t \bullet \alpha) = \mathbb{R}(t') \right\}$$

- The state corresponding to a row $t$ is final if there exists a successful failure $e \in \mathbb{E}$ from $t$ such that the trace component of $e$ is empty. In other words

$$F = \{t \mid \exists e = (\epsilon, p) \in \mathbb{E} \centerdot \mathbb{R}(t, e) = 1\}$$

Finally, the mapping $\mu$ is constructed as follows. Let $t \in F$ be any final state of $M$. Consider the set $P = \{R \mid e = (\epsilon, R) \in \mathbb{E} \wedge \mathbb{R}(t, e) = 1\}$. From the definition of $F$ above, we know that $P \neq \emptyset$. Then $\mu(t) = Max(P)$. We now present the algorithm to add new failures to $\mathcal{T}$ using a counterexample $CE$ to a candidate query made with a DFLA $C$ constructed as above.

## 5.4 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:** $[b_0 = 0]$ means that $CE \notin U$ and hence $CE \in \mathcal{L}(C)$. Recall that $CE = (t, R)$ and $t = \alpha_1 \bullet \ldots \bullet \alpha_k$. Consider the state $s_k = \widehat{\delta}(t)$ as described above. Since $CE \in \mathcal{L}(C)$, we know that $s_k \in F$ and $\exists R' \in \mu(s_k) \centerdot R \subseteq R'$.

Also, since $C$ was constructed (see Section 5.3) from a table $\mathcal{T} = (\mathbb{T}, \mathbb{E}, \mathbb{R})$, we know that $(\epsilon, R') \in \mathbb{E}$ and $\mathbb{R}(s_k, R') = 1$. However, this means that the failure $(s_k, R') \in U$. Since

$R \subseteq R'$, it follows that $(s_k, R) \in U$ and therefore $b_k = 1$. Since $b_0 = 0$ and $b_k = 1$, there exists an index $j \in \{0, \ldots, k\}$ such that $b_j = 0$ and $b_{j+1} = 1$. In this case, $L^F$ finds such an index $j$ and adds the failure $\left(t^{j+1}, R\right)$ to $\mathbb{E}$. We now show that the failure $e = \left(t^{j+1}, R\right)$ has a special property.

Since $C$ contained a transition $s_j \xrightarrow{\alpha_{j+1}} s_{j+1}$, it must be the case that $\mathbb{R}(s_j \bullet \alpha_{j+1}) = \mathbb{R}(s_{j+1})$. However, $\mathbb{R}(s_j \bullet \alpha_{j+1}, e) = b_j \neq b_{j+1} = \mathbb{R}(s_{j+1}, e)$. Thus, after adding $e$ to $\mathbb{E}$, the table is no longer closed. Hence, when $L^F$ attempts to make $\mathcal{T}$ closed in the very next iteration, it will be forced to increase the number of rows of $\mathcal{T}$ by at least one.

**Case 2:** $[b_0 = 1]$ means that $CE \in U$ and hence $CE \notin \mathcal{L}(C)$. We consider two subcases. First, suppose that $b_k = 0$. Then there exists an index $j \in \{0, \ldots, k\}$ such that $b_j = 1$ and $b_{j+1} = 0$. In this case, $L^F$ finds such an index $j$ and adds the failure $\left(t^{j+1}, R\right)$ to $\mathbb{E}$. As in Case 1 above, this guarantees that the number of rows of $\mathcal{T}$ must strictly increase in the next iteration of $L^F$.

Otherwise, we have $b_k = 1$, but this means that the failure $(s_k, R) \in U$. However, since $CE \notin \mathcal{L}(C)$ we know that either $s_k$ is not a final state of $C$ or $\forall R' \in \mu(s_k) \centerdot R \not\subseteq R'$. In this scenario, $L^F$ computes a maximal element $R_{max}$ such that $R \subseteq R_{max}$ and $(s_k, R_{max}) \in U$. It then adds the failure $(\epsilon, R_{max})$ to $\mathbb{E}$.

The addition of $(\epsilon, R_{max})$ to $\mathbb{E}$ must lead to at least one of two consequences in the next iteration of $L^F$ in terms of the next computed candidate DFLA. First, the number of rows of $\mathcal{T}$ and states of the candidate may increase. Otherwise, either the state $s_k$ changes from a non-final to a final state or the set $\mu(s_k)$ gets an additional element, namely, $R_{max}$.

**Relationship Between $L^F$ and $L^*$.** Although $L^F$ and $L^*$ are similar in their overall structure, there are a number of differences. First, since $L^F$ learns a failure automaton, the columns of the observation table store failures instead of traces as in $L^*$. Second, when $L^F$ learns from a counterexample, every iteration may not involve an increase in the number of states; instead, the failure label on one or more states may be enlarged.

## 5.5 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 confirm that $L^F$ always terminates, observe that in every iteration, the candidate $C$ that $L^F$ computes undergoes at least one of these three changes:

- **(Ch1)** The number of states of $C$ and the number of rows in 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 states $s$ 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 must 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 $\Sigma$ 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 $\mathcal{O}(m')$. The 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 $\mathcal{O}(n + n')$. 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, the 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 the total space required for the observation table is $\mathcal{O}((n + m)(n + n'))$. The 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 $m$.

# 6 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 that $L^F$ poses. 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 or is otherwise FALSE: $\forall (t, R_1) \in L_1 \centerdot (t, R_1 \cup R) \in L_S$.

**Candidate Query.** A candidate query with an FLA $C$ 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, namely, $\mathcal{L}(C)$, 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 these queries, 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 assumption follows from the correctness of $L^F$ algorithm: In the worst case, the candidate query will be made with an FLA $C$ such that $\mathcal{L}(C) = L_W$. In this scenario, termination is guaranteed to occur due to Theorem 1.

# 7 Arbitrary Components and Circularity

We investigated two approaches for handling more than two components. First, we applied **AG-NC** recursively. This approach 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 (see Section 6). 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.*

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

*Proof.* We first prove soundness by contradiction. Assume that three premises hold but the conclusion does not. There exists a trace $t$ and a refusal $R$ such that $(t, R) \in L_1$ and $(t, \Sigma \setminus R) \in L_2$. From the first premise, we see that $(t, \Sigma \setminus R) \notin L_A^1$. Similarly, from the second premise, we get $(t, R) \notin L_A^2$. Therefore, we have $(t, R) \in W(L_A^1)$ and $(t, \Sigma \setminus R) \in W(L_A^2)$. But then $(t, \Sigma) \in W(L_A^1) \parallel W(L_A^2)$, which contradicts the third premise.

We now prove completeness. Let us assume the conclusion. We show that if we set $L_A^1 = W(L_1)$ and $L_A^2 = W(L_2)$, then all three premises are satisfied. The first two premises follow from the definitions of $W(L_1)$ and $W(L_2)$. We prove the third premise by contradiction. Suppose there exists a trace $t$ and a refusal $R$ such that $(t, R) \in W(W(L_1))$ and $(t, \Sigma \setminus R) \in W(W(L_2))$, but then we know that $(t, \Sigma \setminus R) \notin W(L_1)$ and $(t, R) \notin W(L_2)$.

However, this supposition means that $(t, R) \in L_1$ and $(t, \Sigma \setminus R) \in L_2$ and implies that $(t, \Sigma) \in L_1 \parallel L_2$, which contradicts the conclusion.

**Implementation.** To use this rule for deadlock detection of two components $L_1$ and $L_2$ (the approach generalizes to any finite number of components), we use this iterative procedure:

1. Using the first premise, construct a candidate $C_1$ similar to Step 1 of the candidate query in **AG-NC** (see Section 6). 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 check 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 $R \in \widehat{\Sigma}$ 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 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 $(t, R) \in W(\mathcal{L}(C_1))$, $(t, \Sigma \setminus R) \notin \mathcal{L}(C_1)$, which is equivalent to a failed candidate query for $C_1$ with counterexample $(t, \Sigma \setminus R)$. 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.

# 8 Experimental Validation

We implemented our algorithms in the ComFoRT [Chaki 05b] reasoning framework and experimented with a set of real-life examples. All our experiments were done on a 2.4GHz Pentium 4 machine running RedHat Linux 9 and with a time limit of one hour and a memory limit of 2GB. Our results are summarized in Table 3. 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 interprocess 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 one 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 [Chaki 05b] 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. These function calls make 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, that is, 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 3 as "Plain"), the non-circular AG rule (referred to as **AG-NC**), and the circular AG rule (referred to as **AG-Circ**). When using **AG-Circ** (i.e., checking if $W(\mathcal{L}(C_1)) \parallel W(\mathcal{L}(C_2)) \subseteq \overline{L_{Dlk}}$), Step 2 was done via compositional language containment using **AG-NC**.

We observe that the AG-based methods outperform the naive approach for most benchmarks. More importantly, for each benchmark, the growth in memory consumption combined with the increasing number of components is benign for both AG-based approaches. This bounded growth 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 generally faster than **AG-NC**, but it consumed negligible extra memory on a few occasions. 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.

| 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 | $\beta$ | 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** | $\alpha$ | 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 | - | - | - | - | - | - | - | - | - |

*Figure 3: 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 shown in bold.*

# 9 Conclusion

In this report, we have extended the learning-based automated assume-guarantee paradigm to deadlock detection. We have defined a new kind of automata, which are similar to finite automata but accept failures instead of traces. We have also developed an algorithm called $L^F$ that is similar to $L^*$ and 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 obtained encouraging experimental results on several nontrivial benchmarks.

We believe this work leaves several avenues open for further investigation. An intriguing question concerns the relationship between learning and abstraction refinement. While both approaches construct approximations iteratively, abstraction refinement always strengthens its approximation. On the other hand, learning may either strengthen or weaken its assumption depending on the counterexample to the candidate query that the MAT returns. Another issue is the possibility of increasing the efficiency of our approach via symbolic implementations. Finally, the question of extending the automated assume-guarantee via learning paradigm to yet other types of conformances is not yet settled. For instance, it is unclear how you may use this paradigm to carry out model checking against specifications written in a temporal logic, such as the $\mu$-calculus, CTL or LTL.

# References


**[Alur 05a]** Alur, R.; Cerny, P.; Gupta, G.; Madhusudan, P.; Nam, W.; & Srivastava, A. "Synthesis of Interface Specifications for Java Classes," 98–109. Edited by Palsberg, J. & Abadi, M. *Symposium on Principles of Programming Languages 2005 (Popl 05)*. Long Beach, CA, January 12–14, 2005. New York, NY: ACM Press, January 2005.

**[Alur 05b]** Alur, R.; Madhusudan, P.; & Nam, W. "Symbolic Compositional Verification by Learning Assumptions," 548–562. Edited by Etessami, K. & Rajamani, S. K. *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, Volume 3576 of *Lecture Notes in Computer Science*. Edinburgh, Scotland, July 6–10, 2005. New York, NY: Springer-Verlag, July 2005.

**[Amla 03]** Amla, N.; Emerson, E. A.; Namjoshi, K. S.; & Trefler, R. J. "Abstract Patterns of Compositional Reasoning," 423–438. Edited by Amadio, R. M. & Lugiez, D. *Proceedings of the 14th International Conference on Concurrency Theory (CONCUR '03)*, Volume 2761 of *Lecture Notes in Computer Science*. Marseille, France, September 3–5, 2003. New York, NY: Springer-Verlag, September 2003.

**[Angluin 87]** Angluin, D. "Learning Regular Sets from Queries and Counterexamples." *Information and Computation 75*, 2 (November 1987): 87–106.

**[Barringer 03]** Barringer, H.; Giannakopoulou, D.; & Păsăreanu, C. S. "Proof Rules for Automated Compositional Verification," 14–21. *Proceedings of the 2nd Workshop on Specification and Verification of Component Based Systems (SAVCBS '03)*. Helsinki, Finland, September 1–2, 2003. Ames, Iowa: Iowa State University, September 2003.

**[Brookes 91]** Brookes, S. D. & Roscoe, A. W. "Deadlock Analysis of Networks of Communicating Processes." *Distributed Computing 4* (December 1991): 209–230.

**[Chaki 05a]** Chaki, S.; Clarke, E. M.; Sinha, N.; & Thati, P. "Automated Assume-Guarantee Reasoning for Simulation Conformance," 534–547. Edited by Etessami, K. & Rajamani, S. K. *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, Volume 3576 of *Lecture Notes in Computer Science*. Edinburgh, Scotland, July 6–10, 2005. New York, NY: Springer-Verlag, July 2005.

**[Chaki 05b]** Chaki, S.; Ivers, J.; Sharygina, N.; & Wallnau, K. "The ComFoRT Reasoning Framework," 164–169. Edited by Etessami, K. & Rajamani, S. K. *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, Volume 3576 of *Lecture Notes in Computer Science*. Edinburgh, Scotland, July 6–10, 2005. New York, NY: Springer-Verlag, July 2005.

**[Cobleigh 03]** Cobleigh, J. M.; Giannakopoulou, D.; & Păsăreanu, C. S. "Learning Assumptions for Compositional Verification," 331–346. Edited by Garavel, H. & Hatcliff, J. *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '03)*, Volume 2619 of *Lecture Notes in Computer Science*. Warsaw, Poland, April 7–11, 2003. New York, NY: Springer-Verlag, April 2003.

**[de Roever 98]** de Roever, W. P.; Langmaack, H.; & Pnueli, A., editors. *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Revised Lectures*, Volume 1536 of *Lecture Notes in Computer Science*, Bad Malente, Germany, September 8–12, 1997. New York, NY: Springer-Verlag, 1998.

**[Ernst 99]** Ernst, M. D.; Cockrell, J.; Griswold, W. G.; & Notkin, D. "Dynamically Discovering Likely Program Invariants to Support Program Evolution," 213–224. *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, Los Angeles, CA, May 16–22, 1999. Los Angeles, CA: IEEE Computer Society Press, May 1999.

**[Fournet 04]** Fournet, C.; Hoare, C. A. R.; Rajamani, S. K.; & Rehof, J. "Stuck-Free Conformance," 242–254. Edited by Alur, R. & Peled, D. *Proceedings of the 16th International Conference on Computer Aided Verification (CAV '04)*, Volume 3114 of *Lecture Notes in Computer Science*, Boston, MA, July 13–17, 2004. New York, NY: Springer-Verlag, July 2004.

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

**[Groce 02]** Groce, A.; Peled, D.; & Yannakakis, M. "Adaptive Model Checking," 357–370. Edited by Katoen, J. P. & Stevens, P. *Proceedings of the Eighth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '02)*, Volume 2280 of *Lecture Notes in Computer*

*Science*. Grenoble, France, April 8-12, 2002. New York, NY: Springer-Verlag, April 2002.

**[Grumberg 94]** Grumberg, O. & Long, D. E. "Model Checking and Modular Verification." *ACM Transactions on Programming Languages and System (TOPLAS) 16*, 3 (May 1994): 843–871.

**[Habermehl 05]** Habermehl, P. & Vojnar, T. "Regular Model Checking Using Inference of Regular Languages," 21–36. *Proceedings of the 6th International Workshop on Verification of Infinite-State Systems (INFINITY '04)*, London, England, September 4, 2004, Volume 138(3) of *Electronic Notes in Theoretical Computer Science*. December 2005.

**[Hoare 85]** Hoare, C. A. R. *Communicating Sequential Processes*. London, England: Prentice Hall, 1985.

**[Holzmann 03]** Holzmann, G. *The SPIN Model Checker: Primer and Reference Manual*. Boston, MA: Addison-Wesley, 2003.

**[Kahlon 05]** Kahlon, V.; Ivancic, F.; & Gupta, A. "Reasoning About Threads Communicating via Locks," 505–518. Edited by Etessami, K. & Rajamani, S. K. *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, Volume 3576 of *Lecture Notes in Computer Science*. Edinburgh, Scotland, July 6–10, 2005. New York, NY: Springer-Verlag, July 2005.

**[Kozen 01]** Kozen, D. *Automata on Guarded Strings and Applications* (Technical Report TR2001-1833). Ithaca, NY: Cornell University, 2001.

**[McMillan 97]** McMillan, K. L. "A Compositional Rule for Hardware Design Refinement," 24–35. Edited by Grumberg, O. *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, Volume 1254 of *Lecture Notes in Computer Science*. Haifa, Israel, June 22–27, 1997. New York, NY: Springer-Verlag, June 1997.

**[Overkamp 97]** Overkamp, A. "Supervisory Control Using Failure Semantics and Partial Specifications." *IEEE Transactions on Automatic Control 42*, 4 (April 1997): 498–510.

**[Peled 99]** Peled, D.; Vardi, M. Y.; & Yannakakis, M. "Black Box Checking," 225–240. Edited by Wu, J.; Chanson, S. T.; & Gao, Q. *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE '99)*, Volume 156 of *IFIP Conference Proceedings*, Beijing, China, October 5–8, 1999. Norwell, MA: Kluwer Academic Publishers, October 1999.

**[Pnueli 85]** Pnueli, A. "In Transition from Global to Modular Temporal Reasoning About Programs." *Logics and Models of Concurrent Systems 13* (1985): 123–144.

**[Rivest 93]** Rivest, R. L. & Schapire, R. E. "Inference of Finite Automata Using Homing Sequences." *Information and Computation 103*, 2 (April 1993): 299–347.

**[Roscoe 97]** Roscoe, A. W. *The Theory and Practice of Concurrency.* New York, NY: Prentice-Hall International, 1997.

**[Williams 05]** Williams, A.; Thies, W.; & Ernst, M. D. "Static Deadlock Detection for Java Libraries," 602–629. Edited by Black, A. P. *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP '05)*, Volume 3586 of *Lecture Notes in Computer Science*. Glasgow, UK, July 25–29, 2005. New York, NY: Springer-Verlag, July 2005.

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| 1. AGENCY USE ONLY | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| (Leave Blank) | September 2006 | Final |

| 4. TITLE AND SUBTITLE | 5. FUNDING NUMBERS |
|---|---|
| Assume-Guarantee Reasoning for Deadlock | FA8721-05-C-0003 |

6. AUTHOR(S)

Sagar Chaki and Nishant Sinha

| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) | 8. PERFORMING ORGANIZATION REPORT NUMBER |
|---|---|
| Software Engineering Institute<br>Carnegie Mellon University<br>Pittsburgh, PA 15213 | CMU/SEI-2006-TN-028 |

| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | 10. SPONSORING/MONITORING AGENCY REPORT NUMBER |
|---|---|
| HQ ESC/XPK<br>5 Eglin Street<br>Hanscom AFB, MA 01731-2116 | |

11. SUPPLEMENTARY NOTES

| 12A DISTRIBUTION/AVAILABILITY STATEMENT | 12B DISTRIBUTION CODE |
|---|---|
| Unclassified/Unlimited, DTIC, NTIS | |

13. ABSTRACT (MAXIMUM 200 WORDS)

The use of learning to automate assume-guarantee style reasoning has received a lot of attention in recent years. This paradigm has already been used successfully for checking trace containment, as well as simulation between concurrent systems and their specifications. In this report, the learning-based automated assume-guarantee paradigm is extended to perform compositional deadlock detection. *Failure automata* is defined as a generalization of finite automata that accept regular failure sets. A learning algorithm $L^F$ is developed that constructs the minimal deterministic failure automata accepting any unknown regular failure set using a minimally adequate teacher. This report shows how $L^F$ can be used for compositional regular failure language containment and deadlock detection, using non-circular and circular assume-guarantee rules. Finally, an implementation of techniques and encouraging experimental results on several nontrivial benchmarks are presented.

| 14. SUBJECT TERMS | 15. NUMBER OF PAGES |
|---|---|
| deadlock detection, learning algorithm, assume guarantee, failure automata, finite automata | 38 |

16. PRICE CODE

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| Unclassified | Unclassified | Unclassified | UL |

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102