Automated Compositional Analysis for Checking Component Substitutability

Nishant Sinha

December 2007

Electrical and Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Edmund M. Clarke, Chair
Don Thomas
Dawn Song
Corina Păsăreanu
Oded Maler

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Keywords: Compositional Verification, Assume-Guarantee Reasoning, Component Substitutability, Machine Learning, Trace-containment, Simulation, Deadlock, SAT, SMT

For my family

Abstract

Model checking is an automated technique to verify hardware and software systems formally. Most of the model checking research has focused on developing scalable techniques for verifying large systems. A number of techniques, e.g., symbolic methods, abstractions, compositional reasoning, etc. have been proposed towards this goal. While methods based on symbolic reasoning (using binary decision diagrams or satisfiability solving) and methods based on computing abstractions automatically in a counterexample-driven manner have proved to be useful in verifying hardware and software systems, they do not directly scale to systems with large number of modules or components. The reason is that they try to verify the complete system in a monolithic manner, which inevitably leads to the state-space explosion problem, i.e., there are too many states in the system to explore exhaustively. Compositional reasoning techniques try to address this problem by following a divide-and-conquer approach: the task of system verification is divided into several sub-tasks, each involving a small subset of system components. Assume-Guarantee Reasoning (AGR) is a particular form of compositional verification, where one first generates environment assumptions for a component and then discharges them on its environment (i.e., the other components) separately. Assume-Guarantee Reasoning methods have been mainly studied in a theoretical context traditionally. The central bottleneck in making them practical is the lack of algorithms to automatically compute appropriate environment assumptions for components.

A recent approach for computing these assumptions relies on combining machine learning algorithms together with model checking techniques to achieve its goal. The technique uses machine learning algorithms for finite state machines in an iterative counterexample-driven manner, assisted by a model checker. In this thesis, we build an abstract framework for automated AGR based on machine learning algorithms and propose new algorithms for instantiating this framework for several different notions of composition and conformances. In particular, we propose compositional techniques for checking simulation conformance, based on learning regular tree languages, and for checking deadlock based on learning failure languages. Moreover, we present an approach to scale this framework to real-life systems communicating via shared memory by using new algorithms for learning machines with large alphabets together with symbolic model checking.

Most industrial hardware and software systems are designed using previously available off-the-shelf components. Such component technologies are gaining acceptance in both hardware and software engineering as effective tools for quickly assembling complex systems from pre-developed components. During their life-cycle, these components may undergo several bug-fixes and upgrades and therefore need to be verified after every such component *substitution* step. In this thesis, we refer to this problem as checking *component substitutability*. This problem is pervasive across both software and hardware engineering

communities, where a large amount of effort is spent on re-validating systems from scratch after each update. In the thesis, we first formalize the problem for software systems taking into account that evolution of components may involve both addition of new features and removal of old behaviors. Then, we propose a solution based on an incremental automated AGR technique, together with counterexample-driven automated abstraction techniques.

The new techniques proposed in this thesis have been implemented and evaluated on both software and hardware benchmarks and are shown to be useful in practice.

Acknowledgements

I would like to thank my adviser, Prof. Edmund Clarke for his valuable guidance and support. I am greatly indebted to him for teaching me the basic techniques involved in doing research, thinking about problems and presenting the material. His constant support kept me motivated across the ups and downs of my PhD research. I would also like to thank the members of my thesis committee, Prof. Don Thomas, Prof. Dawn Song, Corina Păsăreanu and Oded Maler, for their comments and feedback.

I would like to extend a special thanks to all members of the model checking group at CMU. In particular, I would like to thank Sagar Chaki, with whom I worked closely for a large portion of research presented here. I was extremely fortunate to have Flavio Lerda as my officemate. I would also like to thank my friends and acquaintances in Pittsburgh for making my stay memorable.

Finally, I would like to thank my parents and brother for their unassuming support and love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software and hardware systems play an important role in our day-to-day lives. With passage of time, these systems are growing to be increasing complex and therefore more prone to faults. Since these complex systems have an ubiquitous presence in our society, it is important that they function correctly and are error-free. Given the complexity of such systems, it is extremely challenging to verify their correctness.

Component technologies are gaining acceptance in the software and hardware systems engineering community as effective tools for quickly assembling complex systems from components. A software component must provide a set of pre-specified functionalities and different vendors are allowed to ship different software implementations of the same set of requirements. During their design, component-based systems may undergo several changes on account of efforts to select and adapt the set of available components to satisfy the requirements. Moreover, such systems continue to evolve throughout their product life-cycle due to bug-fixes and upgrades. Verification of these systems during their design, and more importantly, after each evolution step (also known as the *re-validation* problem) is an extremely important and challenging task.

Automated formal verification techniques hold considerable promise in being able to validate complex hardware and software systems across their evolution. Due to their

exhaustive nature, these approaches are known to outperform testing or simulation-based methods for validating such systems or detecting complex errors.

Model checking [37, 42] is an automated technique to perform formal verification. In this approach, the system to be verified is presented as a finite-state transition graph M (often, a Kripke structure [42]) and the specification is expressed as a temporal logic (e.g., computational tree logic or linear time logic) formula ϕ. The model checking problem is then to decide if M is a model for ϕ, denoted by $M \vDash \phi$. Fully automated and efficient approaches to solve this problem exist and are implemented as tools, viz., the model checkers. Moreover, in many cases when $M \nvDash \phi$, the model checkers can generate a witness counterexample behavior of M that precisely identifies how M *violates* ϕ. In general, the specification can also be represented by a finite-state transition graph P and the model checking question is to determine if model M conforms to P according to some notion of conformance, e.g., trace containment [84] or simulation [104].

However, most real-life systems have an enormous number of possible states that are reachable on some execution and it is challenging for a model checker to efficiently explore the vast state space. This is known as the *state-space explosion* and is the main bottleneck for model checking tools. Given a hardware or software system consisting of multiple modules, its state space may be large due to a number of reasons. Each of the system modules may have a large state space because it contains a large number of internal variables. Moreover, all the modules execute concurrently and therefore the state space of the system may increase exponentially with increase in number of modules.

Most of the research on model checking has focused on alleviating the state-space explosion problem. A number of techniques have been developed, that may be broadly classified into the following categories: (i) *symbolic* techniques, e.g., Binary Decision Diagrams (BDDs) [26] and Boolean satisfiability (SAT) [106], for efficiently exploring the state space; (ii) Techniques based on computing appropriate property-preserving abstractions automatically for model checking [39, 40, 91], (iii) Compositional Reasoning methods [73, 97, 112]

for sub-dividing the verification task into more tractable subtasks. Symbolic techniques based on BDD-based representation and exploration of state spaces allowed model checkers to verify circuits with hundreds of state elements. The emergence of powerful SAT solvers has made model checking even more scalable [22, 113] by handling larger designs which BDDs could not handle. For verifying software systems with possibly infinite number of states, techniques based on automatically computing finite state abstractions using counterexamples [40, 91] (e.g., predicate abstraction [15, 69]) have been quite successful. Although both symbolic and abstraction based methods are very powerful, they are not able to solve many real-life model checking problems in their own capacity. In particular, for systems with several components executing concurrently, these techniques do not scale well as the number of components increase.

Compositional reasoning techniques come to the rescue in such cases: they advocate a *divide-and-conquer* approach to mitigate the state-space explosion: the task of system verification is divided into several sub-tasks, each involving a small subset of system components. Each of these subsets is analyzed separately, perhaps together with an environment that represents a simplified model of the rest of the system and the results obtained for the sub-tasks are combined into a single result about the complete system. Much of work on compositional reasoning has concentrated on devising sound and complete proof rules by which properties of systems can be derived from properties of their components.

Assume-Guarantee Reasoning (AGR) [88, 105, 112] is a form of compositional reasoning which allows us to break down the proof of system correctness in a compositional way into several sub-proofs. Each of the sub-proofs involve showing guarantees on behaviors of a system component under assumptions about behaviors of other components in the system (environment). The collection of these sub-proof antecedents together with a consequent characterizing the system correctness, constitutes an AGR inference rule. An inference rule is said to be syntactically circular if the assumptions of a component form the guarantees of the rest of the components in *each* of the antecedents. Otherwise, a rule is said to be

non-circular.

Suppose we have a system with two components M_1 and M_2 and our goal is to show that a property P holds on the system according to a given conformance relation $\sqsubseteq$, i.e., $M_1 \parallel M_2 \sqsubseteq P$. We can achieve our goal in a compositional manner using the following non-circular AGR inference rule.

$$\frac{M_1 \parallel A \sqsubseteq P \qquad\qquad M_2 \sqsubseteq A}{M_1 \parallel M_2 \sqsubseteq P}$$

This rule provides a compositional way to check the specification P on a system which consists of two components M_1 and M_2, using a suitable assumption A. More precisely, if we can find an assumption A such that the two premises ($M_1 \parallel A \sqsubseteq P$ and $M_2 \sqsubseteq A$) of the rule hold, then the conclusion ($M_1 \parallel M_2 \sqsubseteq P$) of the rule must also hold. In most cases, AGR inference rules are both sound and complete for safety specifications P, and finite-state models M_1 and M_2. Soundness ensures that if the premises of the rule hold, then the consequent must hold. Completeness ensures that if the composition $M_1 \parallel M_2$ satisfies P, then it can be deduced from the premises of the rule, i.e., there exists a witness assumption A that satisfies the premises. Unlike other techniques for verification of finite-state systems, which have been largely automated (e.g., automated abstraction), the compositional verification techniques, till recently, have completely relied on user guidance.

1.1 Automated Assume-Guarantee Reasoning

Most of the research on compositional reasoning and assume-guarantee reasoning (AGR), in particular, has focused on the investigation of theoretical aspects and much less attention has been paid to the task of automating it to make it practical. Although researchers performed several case studies to apply AGR to both hardware and software systems (cf. Related work in Section 3.5), significant manual effort was required to implement the methodology.

In order to automate AGR for verifying a system with respect to its specification, we need to answer two questions: (i) how do we decompose the system in order to identify appropriate verification sub-tasks? (ii) given a decomposition, how do we compute these assumptions automatically? While decompositions can be obtained with small manual effort using the modular description of the system, the problem of computing these assumptions often requires considerable amount of manual intervention. The lack of automation in AGR, mainly due to the unavailability of a method to compute the assumptions automatically, has hindered its application to real-life systems.

Recently in a seminal paper, Cobleigh et al. proposed an automated AGR methodology [44] for checking safety properties on a composition of labeled transition systems. They show that for finite systems and specifications, the assumptions can always be represented by ordinary finite automata. Their technique, therefore, uses machine learning algorithms for regular languages in terms of an automaton, together with a model checker to compute the assumptions automatically. The main idea of the approach is as follows. The algorithm computes an assumption hypothesis iteratively and then presents it to a model checker. The model checker checks if the assumption satisfies the premises of the AGR rule. A particular assumption hypothesis may be either too weak, i.e., it accepts disallowed traces, or too strong, i.e., it rejects traces that must be accepted. In such a case, the model checker generates a counterexample trace that is used by the learning algorithm to improve its hypothesis.

Verification of safety properties involves checking for errors in finite system executions and intuitively involves showing that *something bad never happens*. Although, a large number of interesting correctness properties fall into the safety class, verification techniques vary depending on the particular sub-class which a given property belongs to. For example, verifying linear-time safety properties corresponds to checking finite *trace-containment*, i.e., all finite traces of the system are also *contained* in the specification. In contrast, checking branching-time safety properties involves checking *simulation* or *bisimulation* between the

system and the specification. Checking deadlock also involves reasoning over finite execution traces but it involves explicitly monitoring the disabled actions at each system state; direct extension of trace-containment based methods proves to be inefficient for checking deadlocks. Moreover, there are many other important correctness properties that fall outside the safety class, e.g., liveness properties involve reasoning about infinite behaviors of reactive systems and intuitively correspond to showing that *something good must eventually happen*.

The automated AGR technique presented by Cobleigh et al. focuses on checking trace-containment for systems communicating via rendezvous [82] and executing asynchronously. This thesis builds upon the basic approach presented by Cobleigh et al. and investigates the problem of performing AGR for other kinds of properties, viz., simulation and deadlock. The thesis also proposes an efficient solution for scaling up the approach to synchronously executing hardware systems communicating via shared variables.

1.2 Checking Component Substitutability

As mentioned earlier, the problem of verifying component-based systems during their initial design and across their evolution/deployment is an important one. For example, any software component is inevitably transformed as designs take shape, requirements change, and bugs are discovered and fixed. In general, such evolution results in the *removal* of previous behaviors from the component and *addition* of new ones. Since the behavior of the updated components has no direct correlation to that of its older counterpart, substituting it directly can lead to two kinds of problems. First, the removal of behavior can lead to unavailability of previously provided services. Second, the addition of new behavior can lead to violation of global correctness properties that were previously respected.

Model checking can be used at each stage of a system's evolution to solve the above two problems. In fact, due to the modular nature of component-based systems, assume-

guarantee reasoning can be directly applied to verify global correctness properties. Conventionally, model checking is applied to the entire systems after every update irrespective of how much modification the system has actually undergone. The amount of time and effort required to verify an entire system can be prohibitive and repeating the exercise after each (even minor) system update is, therefore, impractical. Typically, such upgrades involve modification of only a small number of components. Therefore, an efficient revalidation technique should be able to reuse the previous verification results in order to reduce the time and effort required for revalidation.

In this thesis, we propose an incremental method to revalidate a component-based system (hardware or software) which avoids verification from scratch after each update. Since each update involves modification of one or more components, we refer to the problem as checking if the modified components can *substitute* their older counterparts. The technique relies on using both automated abstraction and compositional reasoning techniques. An important characteristic of the proposed method is that it is able to efficiently *reuse* the verification results from the previous step.

1.3 Overview of the thesis

The thesis investigates the problem of automated assume-guarantee reasoning (AGR) based on machine learning for verifying various kinds of properties and proposes a solution to the component substitution problem using incremental AGR and automated abstraction. The new approaches presented in the thesis serve numerous theoretical and practical verification goals: (i) developing automated AGR techniques for multiple notions of conformance and properties (ii) making automated AGR scalable and (iii) provide a solution for the pervasive problem of checking component substitutability.

The thesis is organized into the following chapters:

Chapter 2 contains details about the machine learning algorithms for regular languages

and the L^* algorithm.

Chapter 3 describes the automated AGR method proposed by Cobleigh et al. in an abstract manner, without appealing to any particular notion of composition or conformance.

Chapter 4 presents the component substitutability problem and our solution based on automated abstraction and incremental assume-guarantee reasoning.

Chapter 5 presents an automated AGR framework for checking simulation compositionally for concurrent programs, which execute asynchronously and use rendezvous-based communication.

Chapter 6 describes a SAT-based methodology to scale up automated AGR for synchronous hardware systems which communicate by shared variables. A new lazy approach to learning assumptions with large alphabets is presented.

Chapter 7 presents an automated AGR framework for checking deadlock compositionally for concurrent programs.

Finally, *Chapter 8* summarizes the thesis and presents directions for future research.

Chapter 2

Learning Regular Languages

This chapter describes machine learning algorithms for inferring an unknown regular language. These learning algorithms belong to the broader class of inductive inference methods in the theory of machine learning.

2.1 Machine Learning and Inductive Inference

Machine learning is a process which causes systems to improve with experience. It is an important subfield of artificial intelligence which is concerned with development of algorithms that allow systems to *learn*. Broadly speaking, there are two kinds of learning paradigms: *inductive* and *deductive*. Inductive methods try to hypothesize or extract a general rule or pattern from a given data set. In contrast, deductive methods involve systematic inference of new facts based on the given data set. While the conclusion of inductive reasoning methods may not be correct in general, the deductive methods lead to a correct conclusion if the premises are true. Machine learning methods try to extract information from data automatically by computational and statistical methods. The efficiency and computational analysis of machine learning algorithms is studied under the field of computational learning theory.

In this thesis, we will be concerned with the paradigm of inductive inference, which was first established by Gold [67] in the context of inference of formal languages. Gold introduced two concepts for characterizing inductive inference technique: *identification in the limit* and *identification by enumeration*. An inference method of the first kind proceeds by modifying its initial hypothesis based on an iteratively increasing set of observations obtained from the concept. If the method stops modifying its hypothesis after a finite number of steps and the final hypothesis is correct, then the method is said to identify the unknown concept in limit on the given set of observations. In contrast, an identification by enumeration method enumerates the set of hypotheses in an iterative manner until it finds a correct hypothesis that is compatible with a given set of observations. Although this technique is quite powerful, it is rather impractical since it may not achieve correct identification in the limit or even be computable [13], for example, if the set of hypotheses cannot be effectively enumerated.

Inductive inference of a large variety of objects, including finite state machines, formal languages, stochastic grammars and logical formula, has been studied. An inductive inference problem involves specification of the following items:

- the class of the unknown concepts or rules, e.g., a class of languages.
- the hypothesis space, containing objects describing each unknown concept item.
- the set of observations for each concept.
- the class of inference methods under consideration.
- the criteria for a successful inference.

Consider, for example, the problem of inference of regular languages in form of regular expressions. Suppose we are given that the strings $\{0011, 000011, 0000, 011, 00, 000000\}$ are in the unknown regular language $\mathbb{L}_U$, while the strings $\{0010, 0, 00110, 111, 0001111, 00000\}$ are not in $\mathbb{L}_U$. A plausible hypothesis may be that $\mathbb{L}_U$ consists of an even number of zeros or any number of zeros followed by two ones, defined by the regular expression $(00)^* +$

0^*11. More formally, the problem consists of the following elements. The class of unknown concepts is the set of regular languages over the alphabet $\Sigma = \{0, 1\}$. The hypothesis space consists of all regular expressions over Σ. For a particular concept (regular language), an observation is of form (s, d), where $s \in \Sigma^*$ and $d \in \{true, false\}$. We are interested in methods that are terminating algorithms that output a regular expression starting from a finite presentation (or observation set) of $\mathbb{L}_U$. We may choose identification in the limit as the criterion of success.

2.2 Inductive Inference of Regular Languages

Given an unknown regular language $\mathbb{L}_U$, the problem of inductive inference of our interest involves computing the minimum deterministic finite automaton (DFA) D_m, such that its language, $\mathbb{L}(D_m)$, is equivalent to $\mathbb{L}_U$. A number of algorithms have been proposed for inductive inference of regular languages using automata as the hypothesis object [50]. We will consider efficient algorithms based on learning in the limit paradigm. The basic setup for all these algorithms involves two entities: the *Learner* and the *Teacher*. The Learner represents the algorithm which tries to estimate $\mathbb{L}_U$. The Teacher is knowledgeable about $\mathbb{L}_U$ and is able to provide sample traces from $\mathbb{L}_U$ to the Learner, by answering queries if required. These algorithms can be broadly classified into *active* and *passive* categories. In the passive algorithms, the Teacher provides the Learner with a fixed initial set of trace samples from $\mathbb{L}_U$. The Learner is supposed to infer D_m based upon this initial set. In contrast, the Teacher in the active algorithms is capable of answering queries and providing *counterexample* traces from $\mathbb{L}_U$. Therefore, the Learner is able to enlarge its sample set by asking queries to the Teacher.

In this thesis, we will be concerned with active learning algorithms only. Here, the Teacher is able to answer two kinds of queries posed by the Learner:

1. *Membership query:* Given a string $t \in \Sigma^*$, 'is $t \in \mathbb{L}_U$?'

2. *Candidate query:* Given a hypothesis DFA D, 'is $\mathbb{L}(D) = \mathbb{L}'_U$?

If the candidate query succeeds, the Teacher replies with a TRUE answer. Otherwise, it provides a *counterexample* CE such that $CE \in \mathbb{L}_U - \mathbb{L}(D)$ or $CE \in \mathbb{L}(D) - \mathbb{L}_U$. In the first case, we refer to CE as a *positive* counterexample, since it must be *added* to D. In the second case, we call CE a negative counterexample, since it must be *removed* from D. The Learner computes an initial hypothesis DFA D and improves it iteratively based on the results of the membership and candidate queries, until a correct hypothesis is obtained.

We now discuss the L^* algorithm [12, 117], a well-known algorithm for learning regular languages.

2.3 L^* Algorithm

The L^* algorithm was originally proposed by Angluin [12]. Rivest and Schapire later proposed a variant of the algorithm [117] with a lower complexity. We now describe the algorithm in a step-by-step manner based on the presentation by Rivest and Schapire.

The L^* algorithm learns the minimum DFA D corresponding to an unknown regular language $\mathbb{L}_U$ defined over an alphabet Σ. The algorithm is based on the Nerode congruence [84] $\equiv_N$: For u, $u' \in \Sigma^*$, $u \equiv_N u'$ iff

$$\forall v \in \Sigma^*, u \cdot v \in \mathbb{L}_U \;\Leftrightarrow\; u' \cdot v \in \mathbb{L}_U$$

Intuitively, L^* iteratively identifies the different Nerode congruence classes for $\mathbb{L}_U$ by discovering a representative prefix trace ($u \in \Sigma^*$) for each of the classes with the help of a set of distinguishing suffixes $V \subseteq \Sigma^*$ that differentiate between these classes.

Notation. We represent the empty trace by ϵ. For a trace $u \in \Sigma^*$ and symbol $a \in \Sigma$, we say that $u \cdot a$ is an extension of u. The membership function $[\![\cdot]\!]$ is defined as follows: if $u \in \mathbb{L}_U$, $[\![u]\!] = 1$, otherwise $[\![u]\!] = 0$. A counterexample trace ce is *positive* if $[\![ce]\!] = 1$,

otherwise, it is said to be *negative*.

2.3.1 Observation Table.

Formally, L^* maintains an observation table $\mathcal{T} = (U, UA, V, T)$ consisting of trace samples from $\mathbb{L}_U$, where:

- $U \subseteq \Sigma^*$ is a prefix-closed set of traces,
- The set UA consists of extensions of elements in U on all alphabet symbols, i.e.,

$$UA = \{u \cdot a | u \in U, a \in \Sigma\}$$

- $V \subseteq \Sigma^*$ is a set of experiment traces, used to distinguish states in the candidate automaton, and,
- $T : (U \cup (UA)) \times V \rightarrow \{0, 1\}$ is a function such that:

$$\forall u \in (U \cup (UA)) \centerdot \forall v \in V \centerdot T(u, v) = 1 \equiv u \cdot v \in \mathbb{L}_U$$

Intuitively, one can think of $\mathcal{T}$ as a two-dimensional table. The rows of $\mathcal{T}$ are labeled with the elements of $U \cup (UA)$ while the columns are labeled with elements of V. Finally T denotes the table entries. In other words, the entry corresponding to row u and column v is simply $T(u, v)$. The value of $T(u, v)$ is 1 if $u \cdot v \in \mathbb{L}_U$, otherwise $T(u, v)$ is 0. Figure 2.3(left) shows an example of an observation table. Intuitively, $\mathcal{T}$ contains the results of membership queries on trace samples of form $u \cdot v$, where $u \in U \cup UA$ and $v \in V$.

Table Congruence. We define a congruence $\equiv$ as follows: for $u, u' \in U \cup UA$, $u \equiv u'$ iff $\forall v \in V$, $T(u, v) = T(u', v)$. We can view $\equiv$ as a restriction of the Nerode congruence $\equiv_N$ to prefixes in $U \cup UA$ and suffixes in V. Also, for all $u \in (U \cup UA)$, we denote the set

of traces equivalent to u by $[u]$, where

$$[u] = \{u' \in (U \cup UA) \mid u \equiv u'\}$$

.

Well-formed Table. An observation table $\mathcal{T}$ is said to be well-formed if for all u, $u' \in U$, $u \not\equiv u'$. The L^* algorithm always keeps $\mathcal{T}$ well-formed.[1].

Table Closure. The observation table $\mathcal{T}$ is said to be closed if for each $ua \in UA$, there is a $u' \in U$, so that $ua \equiv u'$. Given any observation table $\mathcal{T}$, we assume that a procedure `CloseTable` makes it closed in the following way. The procedure iteratively selects some $ua \in UA$ so that for all $u \in U$, $ua \not\equiv u$. Then, it adds ua to U and updates the map T by asking membership queries for all extensions of ua using the `FillAllSuccs` procedure. The procedure `CloseTable` terminates with a closed table when no such ua can be found.

DFA Construction. Given a closed table $\mathcal{T}$, L^* obtains a DFA $D = \langle Q, q_0, \delta, F \rangle$ from it as follows: $Q = \{[u] \mid u \in U\}$, where a state $q \in Q$ corresponds to the equivalence class $[u]$ of a trace $u \in U$, $q_0 = [\epsilon]$, $\delta([u], a) = [u \cdot a]$ for each $u \in U$ and $a \in \Sigma$. $F = \{[u] \mid u \in U \wedge T(u, \epsilon) = 1\}$. Suppose that a procedure called `MkDFA` implements this construction. Note that D is both deterministic and complete.

Figure 2.1 shows the pseudocode for the L^* algorithm. The *Init* block performs the initialization steps while the *Repeat* block performs the learning task iteratively. We assume that the procedures `AskMemQ` and `AskCandQ` are implemented by the Teacher to answer membership and candidate queries respectively. In the *Init* block, the sets U and V are both initialized to $\{\epsilon\}$. Next, the algorithm performs membership queries for ϵ (using the `Fill` procedure) and on all extensions $a \in \Sigma$ (using the `FillAllSuccs` procedure), and updates the map T with the results.

[1]A notion of *consistency* is usually used in presentation of L^* [12]. We ignore it because a well-formed table is consistent by definition.

Learner L^*
Let $\mathcal{T} = (U, V, T)$ be an observation table
Init:
$U := V := \{\epsilon\}$
`Fill` (ϵ,ϵ)
`FillAllSuccs` (ϵ)
Repeat:
`CloseTable`$(\mathcal{T})$
DFA $D :=$ `MkDFA`$(\mathcal{T})$
if (`AskCandQ` $(D) = true$)
return D;
else
Let the counterexample be ce
`LearnCE` (ce)

`CloseTable`$(\mathcal{T})$
while $\mathcal{T}$ is not closed
Pick $u' \in UA$ such that $\forall u \in U.\ u \not\equiv u'$
$U := U \cup \{u'\},\ UA := UA \setminus \{u'\}$
`FillAllSuccs`(u$'$)

`FillAllSuccs` (u)
For all $a \in \Sigma$
$UA := UA \cup \{u \cdot a\}$
For each $v \in V$: `Fill`$(u \cdot a,\ v)$

`Fill` (u, v)
$T(u, v) :=$ `AskMemQ`$(u \cdot v)$

Figure 2.1: Pseudocode for the L^* algorithm

The *Repeat* block then performs the learning task in the following way. The `CloseTable` procedure is first used to make $\mathcal{T}$ closed. When a closed table is obtained, the procedure `MkDFA` is used to construct a DFA hypothesis D from it. The algorithm then performs a candidate query with D using the `AskCandQ` procedure. If the candidate query succeeds, L^* finishes by returning D as the correct hypothesis; otherwise, the *Repeat* block continues by trying to compute a more accurate hypothesis by learning from the counterexample CE obtained (using the procedure `LearnCE`). We now present the details of the core learning procedure `LearnCE`.

Learning from Counterexamples. If a candidate query with a hypothesis DFA D fails, a counterexample CE is returned, where either $CE \in \mathbb{L}_U - \mathbb{L}(D)$ or $CE \in \mathbb{L}(D) - \mathbb{L}_U$. The procedure `LearnCE` is analyzes CE and updates the observation table $\mathcal{T}$ so that an improved hypothesis can be derived from $\mathcal{T}$ in the next iteration. In order to describe `LearnCE`, we need a few definitions.

Given $w \in \Sigma^*$, we define its *representative* element, denoted by $[w]^r$, to be some $u \in U$ such that if $q = \delta_D^*(q_0, w)$, then $q = [u]$. It follows from the construction of DFA D that such a u must exist and is unique. Also, $[u]^r = u$ for all $u \in U$. Intuitively, $[w]^r$ is the representative element of the unique state q (equivalence class) that w reaches when it is run on D starting at q_0. We can compute $[w]^r$ by simulating w on D to find q and then

picking the corresponding element $u \in U$.

We define an i-split ($0 \leq i \leq |ce|$) of a counterexample ce to be a tuple (u_i, v_i), where $ce = u_i \cdot v_i$ and $|u_i| = i$. In words, an i-split of ce, consists of its prefix u_i of length i and the corresponding suffix v_i.

For an i-split of ce, we define $\alpha_i = [\![[u_i]^r \cdot v_i]\!]$ ($0 \leq i \leq |ce|$). Intuitively, α_i corresponds to checking if the suffix v_i is accepted starting from the equivalence class represented by $[u_i]^r$. We say that a prefix u_i is *mis-classified* if $[\![u_i \cdot v_i]\!] \neq [\![[u_i]^r \cdot v_i]\!]$. For example, suppose ce is rejected in $\mathbb{L}_U$, i.e., $[\![ce]\!] = 0$. Therefore, $[\![u_i \cdot v_i]\!] = 0$ for all the i-splits. It follows that each of the prefixes u_i must be mapped to an equivalence class $[u_i]$ (with a representative element $[u_i]^r$) such that $[u_i]$ rejects the corresponding suffix v_i, i.e., $[\![[u_i]^r \cdot v_i]\!] = \alpha_i = 0$. Otherwise, if for some i, $\alpha_i = 1$, then u_i is mis-classified.

Intuitively, α_i checks whether the i^{th} prefix u_i is mapped into the correct equivalence class $[u_i]$. More precisely, if $\alpha_i = [\![ce]\!]$ for some i, it implies that u_i is classified correctly. Otherwise u_i is mis-classified and L^* must *re-classify* u_i to a different equivalence class in order to eliminate the counterexample. [2] Note that we can compute α_i by computing $[u_i]^r$, say u, and then asking a membership query for the word $u \cdot v_i$.

The `LearnCE` procedure is given by the pseudocode in Figure 2.2.

```
LearnCE (ce)
  Find i by binary search such that α_i ≠ α_{i+1}
  V := V ∪ {v_{i+1}}
  For all u ∈ U ∪ UA. Fill(u, v_{i+1})
```

Figure 2.2: Pseudocode for the `LearnCE` procedure

The procedure tries to find an index i ($0 \leq i \leq |ce|$) such that $\alpha_i \neq \alpha_{i+1}$ and updates $\mathcal{T}$ by adding the suffix v_{i+1} to V. Since $\alpha_0 \neq \alpha_{|ce|}$, there must exist some i, $0 \leq i \leq |ce|$, so that $\alpha_i \neq \alpha_{i+1}$. In this case, u_{i+1} is mis-classified into the equivalence class represented by the word $[u_{i+1}]^r$, (which corresponds to a state, say q, in D), and v_{i+1} is a witness for

[2] Note that for $i = |ce|$, $\alpha_{|ce|} \neq [\![ce]\!]$, i.e., a counterexample is always mis-classified.

this mis-classification. Adding v_{i+1} to V distinguishes the correct equivalence class (say q') from q and redirects u_{i+1} to q'. We call this a *state partition* (of q).

		E	
		ϵ	
S	ϵ	0	(s_0)
	α	1	(s_1)
$S \cdot \Sigma$	β	0	
	$\alpha\alpha$	1	
	$\alpha\beta$	0	

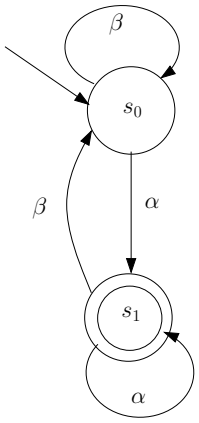


Figure 2.3: An Observation Table and the Corresponding Candidate DFA

Consider Figure 2.3. On the left is an observation table (S, E, T) where S and E correspond to rows and columns respectively and T corresponds to the table entries. Here, $\Sigma = \{\alpha, \beta\}$. From this table we see that $\{\alpha, \alpha \cdot \alpha\} \subseteq U$. On the right is the corresponding candidate DFA. The states s_0 and s_1 of the DFA correspond to the elements ϵ and α of S respectively. The state s_0 is marked initial since it corresponds to word ϵ. The state s_1 is marked final since the table entry $T(\alpha, \epsilon) = 1$. Finally, the transitions are determined as described in the procedure `MkDFA`.

2.4 Termination and Correctness

We now prove some results about the termination and correctness of the L^* algorithm and the procedures `CloseTable` and `MkDFA`. We first show that the L^* algorithm always maintains a well-formed observation table.

Lemma 1 *The L^* algorithm always maintains a well-formed table.*

Proof. Consider the pseudo-code of L^* in Figure 2.1. Given an observation table $\mathcal{T} = (U, V, T)$, the set U is updated only by the `CloseTable` procedure. Hence, we need to

show that `CloseTable` always maintains a well-formed table at each iteration.

We proceed by induction. Note that at the first iteration of the L^* loop (first call to `CloseTable`), U only has a single element and hence the table is well-formed. Assume that the input observation table $\mathcal{T}$ to `CloseTable` is well-formed at k^{th} iteration ($k > 1$). The procedure `CloseTable` (cf. Figure 2.1) only adds a new element t to S if for all $s \in S$, $s \not\equiv t$. Therefore, all the elements in $S \cup \{t\}$ are non-equivalent and the resultant table is also well-formed.

□

The following lemma is crucial for proving termination of L^*. It essentially provides an upper bound on the size of S.

Lemma 2 *Let $\mathcal{T} = (S, E, T)$ be a well-formed observation table. Let U be an unknown regular language and n be the number of states in the minimum DFA M such that $\mathbb{L}M = U$. Then the size of the trace set S cannot exceed n.*

Proof. Let δ denote the transition relation of M (Δ extended to words) (cf. procedure `MkDFA`) and q_0 denote the initial state for M.

The proof is by contradiction. Suppose that the size of S exceeds n. Then by the pigeonhole principle, there exist two elements s_1 and s_2 of S such that $\delta(s_1, \{q_0\}) = \delta(s_2, \{q_0\}) = q$ (say), i.e., s_1 and s_2 must reach the same state q in M. Since M is the minimum DFA for U, we know that the states of M correspond to equivalence classes of the Nerode congruence [84] for U. Since s_1 and s_2 reach the same state in M (same Nerode equivalence class), it follows that

$$\forall e \in \Sigma^*, s_1 \cdot e \in U \text{ iff } s_2 \cdot e \in U \tag{2.1}$$

But, $\mathcal{T}$ is well-formed and hence $s_1 \not\equiv s_2$. Therefore there exists some $e \in E$, such that $T(s_1 \cdot e) \neq T(s_2 \cdot e)$, i.e., $s_1 \cdot e \in U$ and $s_2 \cdot e \notin U$ or vice versa. Together with (2.1), we reach a contradiction.

□

The following lemma shows that the procedure `CloseTable` cannot increase the size of S indefinitely and must terminate in finite number of steps.

Lemma 3 *The procedure* `CloseTable` *always terminates with a closed table. Moreover, the procedure maintains a well-formed observation table* $\mathcal{T} = (S, E, T)$ *at each iteration.*

Proof. It follows from the pseudo-code (Figure 2.1) that the procedure `CloseTable` keeps adding new elements to S until $\mathcal{T}$ is closed. Since the size of S is bounded by the number of states in the minimum DFA for the unknown language U (Lemma 2), `CloseTable` terminates with a closed table in finite number of steps.

It follows from Lemma 1 that the procedure always maintains a well-formed table.

□

The following lemma shows that the `MkDFA` procedure always constructs a candidate DFA starting from a well-formed and closed observation table.

Lemma 4 *Given a well-formed and closed observation table as an input, the procedure* `MkDFA` *always terminates with a candidate DFA* D *as a result.*

Proof. Since the input table $\mathcal{T}$ is well-formed, the states of D are uniquely defined by the elements of S. Moreover, the initial state is unique by definition and the final states are well-defined. Since the table is closed, the transition relation of D is also well-defined. Hence, the candidate DFA is well-defined.

□

Lemma 5 *Given a counterexample* ce*, the procedure* `LearnCE` *always finds an index* i *such that* $\alpha_i \neq \alpha_{i+1}$*. Also,* `LearnCE` *always terminates with a non-closed observation table.*

Proof. From the definition of counterexample, it follows that $\alpha_0 \neq \alpha_{|ce|}$. Hence, there exists an index i where $\alpha_i \neq \alpha_{i+1}$.

Given a closed observation table $\mathcal{T} = (U, V, T)$, the `LearnCE` procedure adds a suffix v_{i+1} to V so that $\alpha_i \neq \alpha_{i+1}$, i.e., $[\![[u_i]^r \cdot v_i]\!] \neq [\![[u_{i+1}]^r \cdot v_{i+1}]\!]$. Let $u_{i+1} = u_i \cdot a$ for some

$a \in \Sigma$. Therefore, $v_i = a \cdot v_{i+1}$ and it follows that

$$[\![[u_i]^r \cdot a \cdot v_{i+1}]\!] \neq [\![[u_i \cdot a]^r \cdot v_{i+1}]\!]. \tag{2.2}$$

By definition, we know that both $[u_i]^r$ and $u = [u_i \cdot a]^r$ are in U and hence $u' = [u_i]^r \cdot a$ is in $U \cup UA$. But, using the definition of α, $[u_i \cdot a] = [u']$. Hence $u \equiv u'$ in $\mathcal{T}$ and since $\mathcal{T}$ is well-formed, it follows that $u' \in UA$. Also, we know that for all $u_1 \in U$ except u, $u' \not\equiv u_1$.

Now, using Eq. 2.2, we have $T(u', v_{i+1}) \neq T(u, v_{i+1})$. Therefore, after adding v_{i+1} to V, $u' \not\equiv u$. As a result, $u' \in UA$ and for all $u_1 \in U$, $u' \not\equiv u_1$. Hence, the observation table obtained as a result is not closed.

□

Theorem 1 *Algorithm L^* always terminates with the correct result.*

Proof. That L^* terminates with the correct result is obvious since it stops only after a candidate query has passed. To prove that it terminates, it suffices to show that there can only be a finite number of failed candidate queries and therefore only a finite number of iterations of the top-level loop (Figure 2.1, line 2).

For each failed candidate query, the procedure `LearnCE` is executed. It follows from Lemma 5 that the observation table obtained at end of execution of `LearnCE` is not closed. Hence, the procedure `CloseTable` must add at least one element to S in the next iteration of the top-level loop. However, the size of S is bounded (cf. Lemma 2) and hence the loop will execute only finite number of times.

□

2.5 Discussion on Learning Algorithms

A comprehensive survey of various learning algorithms can be found in a work by Higuera [50]. Language *identification in the limit* paradigm was introduced by Gold [50, 67].

This forms the basis of *active* algorithms which learn in an online fashion by querying an oracle or a teacher. Gold also proposed another paradigm, namely *identification from given data*, for learning from a fixed training sample set [68]. The training set consists of a set of positive and negative samples from the unknown language and must be a *characteristic* [68] set of the language. Algorithms have been proposed in this setting for learning word languages [54, 109], tree languages [20, 62] and stochastic tree languages [27]. Learning tree languages in form of a tree automaton was investigated in the context of context-free grammars [119].

2.5.1 Comparison with CEGAR approach

The learning algorithms are similar to the counterexample-guided abstraction refinement (CEGAR) approach [16, 41, 91] in that both approaches search the space of hypotheses/abstractions in order to find a suitable hypothesis/abstraction that meets a given constraint on the language of the hypothesis. Both the approaches begin with an initial hypothesis and then modify/refine the hypothesis iteratively until the correct hypothesis is found. Moreover, both the approaches are counterexample-driven, i.e., the modification of the hypothesis/abstraction at each step is based on the analysis of an erroneous counterexample behavior provided by an external oracle. However, there are fundamental differences between the two approaches:

- In the CEGAR approach, each state is characterized by a predicate on the variables in the program. In contrast, each state in the learning algorithm is uniquely identified by a representative trace and the corresponding state characterization vector (or a *row* in the observation table), which is based on the set of distinguishing traces. While a state in the CEGAR approach represents the set of concrete states corresponding to the predicate label, a state in the learning algorithm corresponds to a collection of one or more Nerode congruence classes [84] of the unknown language.

- The CEGAR approach removes spurious behaviors from an abstract model by partitioning the abstract states with the help of new predicates. These predicates are obtained by analyzing the counterexamples with respect to concrete model semantics and identifying a transition in the counterexample that is infeasible in the concrete model. The state partition, in turn, leads to elimination of spurious transitions. In contrast, the learning algorithm tries to remove or add counterexample behaviors to a hypothesis with the help of the Nerode congruence directly. More precisely, the algorithm analyzes the counterexample to find *erroneous prefixes*, i.e., the prefixes that have been mapped to the wrong congruence class, and then maps each of them to a new congruence class with the help of new traces that distinguish them from the previous class. This re-mapping, in turn, leads to partitioning of congruence classes or states in the previous hypothesis.
- In the CEGAR approach, the language of the abstraction decreases monotonically, due to elimination of spurious counterexamples from the abstraction. In contrast, the language accepted by the candidate hypotheses may change non-monotonically across iterations of the learning algorithm.

2.5.2 Applications of Learning

The application of learning is extremely useful from a pragmatic point of view since it is amenable to complete automation, and it is gaining rapid popularity in formal verification. This thesis focuses on using learning algorithms together with a model checker for compositional verification of hardware and software systems. The details of this approach will be presented in the subsequent chapters. Besides application to compositional verification, learning has been applied together with predicate abstraction in the context of interface synthesis [6, 78] and for automated software verification [28]. Other applications include automatic synthesis of interface specifications for application programs [6, 11],

automatically learning the set of reachable states in regular model checking [76, 126], *black-box-testing* [111] and its subsequent extension to *adaptive model-checking* [72] to learn an accurate finite state model of an unknown system starting from an approximate one, and learning likely program invariants based on observed values in sample executions [57].

Chapter 3

Automated Compositional Verification

Verification approaches based on compositional reasoning allow us to prove properties (or discover bugs) for large concurrent systems in a divide-and-conquer fashion. Assume-guarantee Reasoning (AGR) [88, 105, 112] is a particular form of compositional verification, where we first generate environment assumptions for a component and discharge them on its environment (i.e., the other components). The primary bottleneck is that these approaches require us to manually provide appropriate environment assumptions.

In this chapter, we present an automated AGR approach based on using machine learning algorithms for computing the environment assumptions. Our description is based on the initial approach proposed by Cobleigh et al. [43]. Instead of fixing a particular representation for the system models, their composition and the notion of conformance, we present the automated AGR framework in an abstract manner here. The following chapters contain particular instantiations of the abstract framework presented here.

3.1 Preliminaries

The set of finite state models is denoted by MOD and includes instances of all different finite state transition systems, e.g., finite state automata, labeled transition systems etc. We adopt this general setting since different finite state model types can be used for describing the implementation, specification and the assumption models.

The set of behaviors exhibited by a finite-state model $M \in MOD$ is said to be its language $\mathbb{L}(M)$. Let the universal set of behaviors be denoted by $\mathcal{U}$, such that for each model M, $\mathbb{L}(M) \subseteq \mathcal{U}$. The $\overline{L}$ of a language L is defined to be $\mathcal{U} \setminus L$. We will consider the following two operators, complementation and composition, on these models.

- **Complementation.** The complement of a finite-state model M is a finite state model denoted by $\overline{M}$.
- **Composition.** The composition of two finite state models M_1 and M_2 is a finite state model denoted by $M_1 \parallel M_2$.

We further require that the above operators, if defined, obey the following language constraints:

- (**LC1**) $\mathbb{L}(\overline{M}) = \overline{\mathbb{L}(M)}$.
- (**LC2**) $\mathbb{L}(M_1 \parallel M_2) = \mathbb{L}(M_1) \cap \mathbb{L}(M_2)$.

A (finite state) *system* consists of one or more (finite state) models.

Model Checking problem. Given an implementation system M, a specification system P and a conformance relation $\vDash$, the model checking problem is denoted by $M \vDash P$. In words, our goal is to check whether the implementation M *conforms* to the specification P.

Model Checking as Language Containment. Using the notion of language of a system, we can cast the model checking problem as a language containment problem, i.e., $M \vDash P$ iff $\mathbb{L}(M) \subseteq \mathbb{L}(P)$.

In this thesis, we will always solve the model checking problem by considering the

corresponding language containment problem. As we will see, even the problems of checking simulation and deadlock can be mapped to language containment problems based on an appropriate definition of the language.

3.2 Assume-Guarantee Reasoning

We now describe the assume-guarantee reasoning (AGR) framework for abstract finite state systems. AGR rules may be syntactically circular or non-circular in form. In this thesis, we will be concerned mainly with the following two AGR rules, **NC** and **C**.

Definition 1 Non-circular AGR (NC) *Given finite-state systems M_1, M_2 and P, show that $M_1 \parallel M_2 \models P$, by picking a finite-state assumption model A, such that both* **(n1)** *$M_1 \parallel A \models P$ and* **(n2)** *$M_2 \models A$ hold.*

The non-circular rule was first proposed by Pnueli [112]. The following circular rule has also been proposed in literature [17, 107].

Definition 2 Circular AGR (C) *Show that $M_1 \parallel M_2 \models P$ holds by picking an assumption tuple, $\langle A_1, A_2 \rangle$, such that each of the following hold:* **(c1)** *$M_1 \parallel A_1 \models P$* **(c2)** *$M_2 \parallel A_2 \models P$ and* **(c3)** *$\overline{A_1} \parallel \overline{A_2} \models P$.*

Both **NC** and **C** rules are sound and complete for various notions of languages, composition and conformance. Moreover, both can be extended to a system of n models $M_1 \ldots M_n$ by picking a set of assumptions (represented as a tuple) $\langle A_1 \ldots A_{n-1} \rangle$ for **NC** and $\langle A_1 \ldots A_n \rangle$ for **C** respectively [17, 43, 107]. The proofs of completeness for both these rules rely on the notion of weakest assumptions.

Definition 3 (Weakest Assumptions) *Given a finite system M with a property P and an assumption WA, we say that WA is the weakest assumption for M and P iff: (i) $M \parallel WA \models P$ holds, and (ii) for all assumptions A where $M \parallel A \models P$, $\mathbb{L}(A) \subseteq \mathbb{L}(WA)$ holds. The weakest assumption language $L_W = \mathbb{L}(WA)$.*

Lemma 6 (Weakest Assumption Language) *Given a finite system M and a property model P where* **LC2** *holds for M and P, the weakest assumption language $L_W = \overline{\mathbb{L}(M)} \cup \mathbb{L}(P)$.*

Proof. Using **LC2** and the notion of model checking as language containment, our goal is to show that (i) $\mathbb{L}(M) \cap L_W \subseteq P$ holds and (ii) for all A where $\mathbb{L}(M) \cap \mathbb{L}(A) \subseteq P$ holds, it follows that $\mathbb{L}(A) \subseteq L_W$ holds. Assume $\mathbb{L}(M) \cap \mathbb{L}(A) \subseteq P$ holds for some assumption model A. On rearranging, we get, $\mathbb{L}(A) \subseteq \overline{\mathbb{L}(M)} \cup \mathbb{L}(P)$. Let $L = \overline{\mathbb{L}(M)} \cup \mathbb{L}(P)$. Therefore $\mathbb{L}(A) \subseteq L$ and (ii) holds. Also, (i) holds for L on solving. Hence $L_W = L$.

The following lemma shows that we can check if a given trace t is in $\mathbb{L}(WA)$ without constructing WA directly by checking if $M \parallel t \sqsubseteq \varphi$ holds.

Lemma 7 *Supposed WA be the weakest assumption for finite system M and specification system φ. Given a trace t, $t \in L(WA)$ if $M \parallel t \sqsubseteq \varphi$.*

Proof. It follows from the definition of weakest assumptions that for all assumptions such that $M \parallel A \sqsubseteq \varphi$, $L(A) \subseteq L(WA)$ holds. Let M_t be the automaton representation of t. Since $M \parallel M_t \sqsubseteq \varphi$ holds, therefore it follows that $L(M_t) \subseteq L(WA)$. Since $L(M_t) = \{t\}$, hence $t \in L(WA)$.

Representing Weakest Assumptions.

Definition 4 (Language Set Cover by a Model Class) *A model class $\mathcal{M} \subseteq MOD$ is said to cover a language set $\mathcal{L}$ if $\forall L \in \mathcal{L}.\ \exists M \in \mathcal{M}.\ \mathbb{L}(M) = L$. $\mathcal{M}$* uniquely *covers $\mathcal{L}$ if there exists an unique $M \in (M)$ so that $\mathbb{L}(M) = L$.*

Lemma 8 (Sufficient Condition for Weakest Assumption Models) *A model class $\mathcal{M}$ can be used to represent the weakest assumption WA if it covers the language set consisting of arbitrary boolean combinations of languages of model classes for M and P. Moreover, WA is* unique *if the model class uniquely covers above language set.*

Proof. Follows from the Lemma 6.

As we shall see in the subsequent chapters, this sufficient condition dictates the choice of our assumption model representation in the AGR framework.

3.2.1 Soundness and Completeness

In the proof of the following theorems, we use propositional logic notation for representing intersection, complementation and union of languages. Let $lm_1 = \mathbb{L}(M_1)$ and $lm_2 = \mathbb{L}(M_2)$ and $lp = \mathbb{L}(P)$. Given assumptions A_1 and A_2, $la_1 = \mathbb{L}(A_1)$ and $la_2 = \mathbb{L}(A_2)$. The weakest assumption language is $lwa_i = \neg(lm_i) \vee lp$ $(i \in \{1, 2\})$.

Theorem 2 *Rule* **NC** *is sound and complete.*

Proof.

- *Soundness.* We need to show that if $lm_1 \wedge la_1 \implies lp$ and $lm_2 \implies la_1$ hold, then $lm_1 \wedge lm_2 \implies lp$ holds. Since, $lm_2 \implies la_1$, therefore $lm_1 \wedge lm_2 \implies lm_1 \wedge la_1 \implies lp$.
- *Completeness.* We show that if $lm_1 \wedge lm_2 \implies lp$, then both the premises of **NC** hold for the weakest assumption. The first premise, $lm_1 \wedge la_1 \implies lp$ holds since $lm_1 \wedge \neg(lm_1 \vee lp) = lm_1 \wedge lp$ and $lm_1 \wedge lp \implies lp$. Also we can rewrite $lm_1 \wedge lm_2 \implies lp$ as $lm_2 \implies (\neg lm_1 \vee lp)$, which is same as the second premise $lm_2 \implies lwa_1$.

Theorem 3 *Rule* **C** *is sound and complete.*

Proof.

- *Soundness.* We need to show that if $lm_i \wedge la_i \implies lp$ $(i \in \{1, 2\})$ and $\neg la_1 \wedge \neg la_2 \implies lp$, then $lm_1 \wedge lm_2 \implies lp$. Given an element $t \in lm_1 \wedge lm_2$, we consider three cases:
  - $t \in \neg la_1 \wedge \neg la_2$: It follows from the third premise that $t \in lp$.

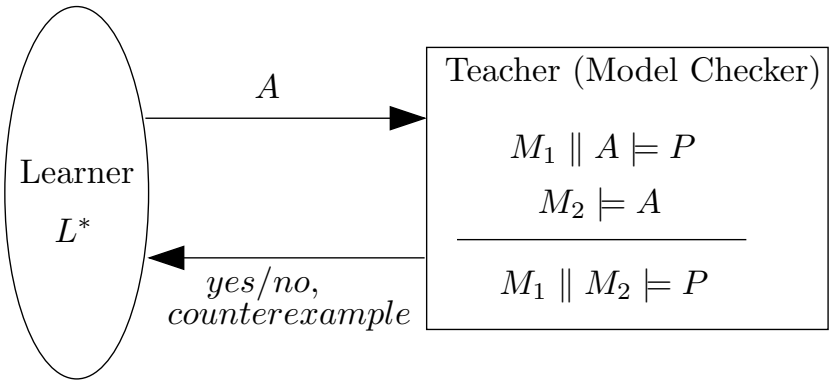


Figure 3.1: Learning-based Automated Assume-Guarantee Reasoning Procedure (simplified).

- $t \in la_1$: Since $t \in lm_1$, it follows from the first premise that $t \in lp$.
- $t \in la_2$: Since $t \in lm_2$, it follows from the second premise that $t \in lp$.

- *Completeness.* Given (i) $lm_1 \wedge lm_2 \implies lp$, we show that all the three premises hold for the weakest assumption languages lwa_1 and lwa_2. The first two premises hold by definition. Note that for $i \in \{0, 1\}$, $\neg lwa_i = lm_i \wedge \neg lp$. Now, $\neg lwa_1 \wedge \neg lwa_2 = lm_1 \wedge lm_2 \wedge \neg lp$ which implies lp due to (i).

3.3 Automated Assume-Guarantee Reasoning

As mentioned earlier, the main obstacle in practical usage of the rules **NC** and **C** is the task of computing assumptions. Cobleigh et al. presented an iterative strategy to automatically compute these assumptions using the L^* algorithm [12, 117]. Recall (cf. Chapter 2) that the L^* algorithm (which acts as the Learner) computes the minimum DFA corresponding to an unknown regular language by asking membership and candidate queries to a Teacher. The assumptions are represented by DFAs and a model checker serves as a Teacher for the L^* algorithm. Figure 3.1 gives a simplified view of this approach.

In the following, we present the strategies for carrying out automated assume-guarantee reasoning (AGR) using both the rules. However, we do not go into the details of the

learning algorithm here (cf. Chapter 2). Instead, we encapsulate the algorithm inside an *assumption generator* (AGEN) interface. We assume that this interface is able to output (i) an initial assumption hypothesis, and (ii) a new assumption hypothesis, when presented with a counterexample behavior to the previous hypothesis.

3.3.1 Rule NC

Figure 3.2 shows the overview of the strategy (called $\mathcal{S}_{NC}$) used for AGR with **NC**. The main idea is the following: given an assumption hypothesis, a model checker is used to check the two premises of **NC**. If any of these premises do not hold, a counterexample is returned. The procedure then checks if the counterexample is spurious. If found to be spurious, the counterexample is returned to the AGEN interface, so that AGEN can generate an improved hypothesis. Otherwise, CE is returned as a counterexample witness for $M_1 \parallel M_2 \models P$. More precisely, $\mathcal{S}_{NC}$ consists of the following steps:

1. Use model checking to verify that the current assumption A provided by AGEN satisfies the first premise of **NC**. If the verification fails, obtain a counterexample CE and proceed to step 2. Otherwise, proceed to step 3.
2. Check if $CE \in \mathbb{L}(M_2)$. If so, then return CE as a witness to $M_1 \parallel M_2 \not\models P$. Otherwise, return CE to AGEN.
3. Use model checking to verify the second premise. If the verification succeeds, return TRUE and terminate. Otherwise, obtain a counterexample CE.
4. Check if $CE \in \mathbb{L}(WA)$, i.e., $CE \in \mathbb{L}(M_1)$ and $CE \notin \mathbb{L}(P)$. If so, return CE as a witness to $M_1 \parallel M_2 \not\models P$. Otherwise return CE to AGEN

Termination We show here that if the AGEN interface is instantiated by the L^* algorithm, $\mathcal{S}_{NC}$ must converge to the weakest assumption WA in finite number of steps.

Proof. Define a run of L^* to be the sequence of labeled traces (trace with a boolean label) returned by membership and candidate queries. For any run ρ of L^* using the above

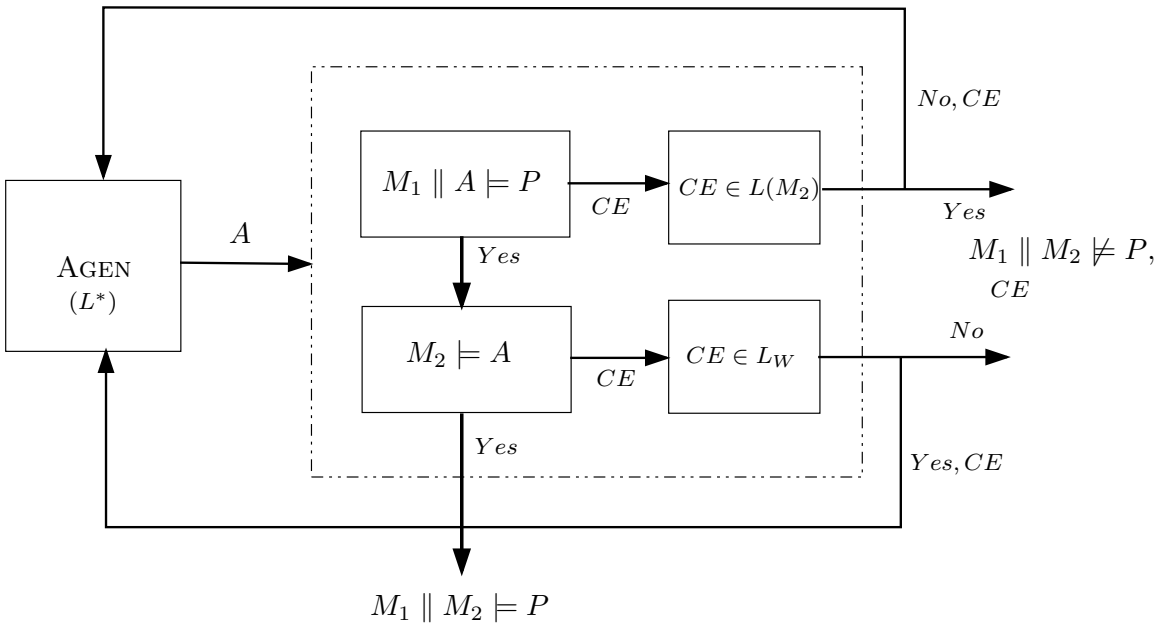


Figure 3.2: Automated AGR for the rule **NC**

strategy $\mathcal{S}_{NC}$, we can construct an equivalent L^* run, say ρ', with $L(WA)$ as the unknown language. Since we know that any ρ' consists of finite number of steps and terminates with WA as the result, ρ must also terminate with WA.

We use induction on ρ to show that ρ' is a run with $L(WA)$ as the unknown language. In the base case, $\rho = \epsilon = \rho'$. Let ρ_n denote the prefix of ρ of length n. Assume ρ has length k and $\rho_{k-1} = \rho'_{k-1}$. Let (t, b) be the last labeled trace in ρ; we know that either (t, b) was obtained as a result of a membership or a candidate query. (i) Membership query: Since $\mathcal{S}_{NC}$ always answers queries with respect to $L(WA)$ and given ρ_{k-1}, t is unique, we conclude that (t, b) is also the last element of ρ'. (ii) Candidate query: From the definition of $\mathcal{S}_{NC}$, we know that either (Step 2) $t \in \mathbb{L}(A)$ and $t \notin \mathbb{L}(WA)$ and $b = false$, or (Step 4) $t \in \mathbb{L}(WA)$ and $t \notin \mathbb{L}(A)$ and $b = true$. Therefore, (t, b) will also be returned by $\mathcal{S}'_{NC}$.

Correctness. It follows from the definition that $\mathcal{S}_{NC}$ must either terminate with some assumption A that satisfies the premises of **NC** or with a counterexample CE. In the first case, the correctness follows from the soundness of **NC**. In the second case, either Step 2 or Step 4 generates CE. Therefore, $CE \in \mathbb{L}(M_2)$ and $CE \notin \mathbb{L}(WA)$. Hence, CE is an

actual counterexample.

Remarks.

1. It is possible to omit the check in Step 2 above, while maintaining the correctness and termination properties of $\mathcal{S}_{NC}$. Step 2 can be viewed as an *early termination* check.

2. The strategy $\mathcal{S}_{NC}$ is biased towards WA, since we proceed to Step 3 (second premise) only if Step 1 (first premise) holds. Alternatively, we could bias the strategy towards M_2 by always checking Step 3 before Step 1.

3. When instantiating the AGEN interface with the L^* algorithm, we further bias $\mathcal{S}_{NC}$ towards WA, since membership queries are answered with respect to WA. More precisely, given a trace $t \in \mathbb{L}(WA)$ and $t \notin \mathbb{L}(M_2)$, the membership query result for t is TRUE. Similarly, we can also bias $\mathcal{S}_{NC}$ towards M_2 by answering membership queries only with respect to M_2.

4. In practice, one obtains a set of counterexamples in Step 1 and 3 (represented as a trace on a smaller set of alphabet symbols of either $M_1 \parallel \overline{P}$ or M_2 respectively). The steps 2 and 4 involve choosing a particular counterexample from this set.

3.3.2 Rule C

Figure 3.3 shows the overview of the strategy $\mathcal{S}_C$ used for AGR with **C**. Again, the model checker checks the three premises of **C** iteratively and spurious counterexamples are used to improve the hypotheses. This rule employs multiple AGEN interfaces; we will use AGEN_1 and AGEN_2 in the following. The strategy consists of the following steps:

1. Obtain assumption A_1 from AGEN_1 and model check the first premise of **C**. If the check fails, return the obtained counterexample to AGEN_1. Otherwise, continue to Step 2.

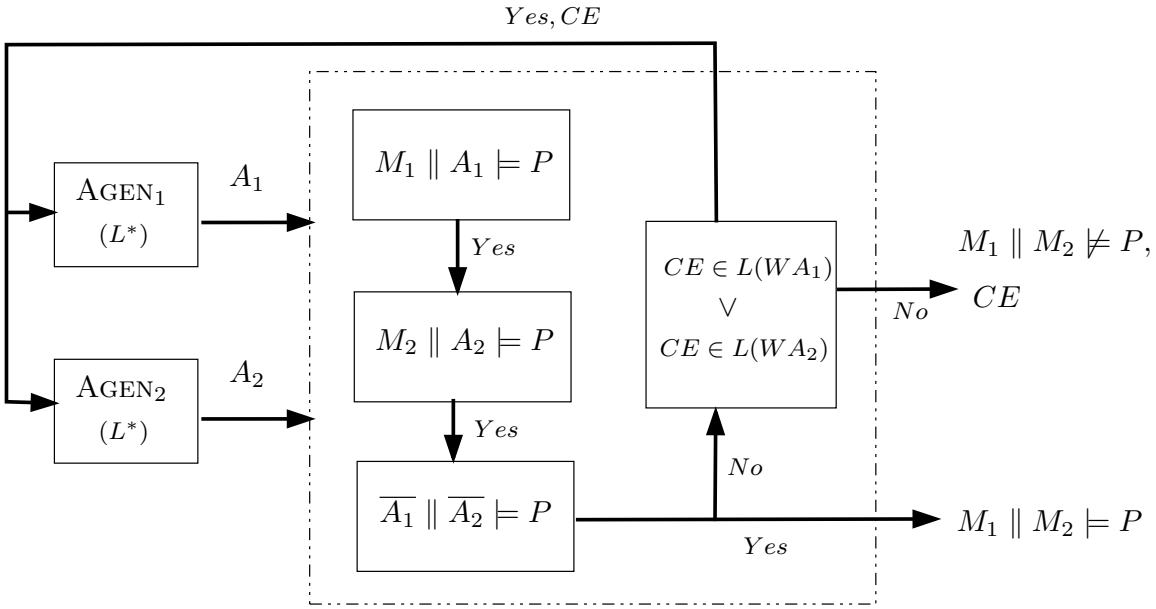


Figure 3.3: Automated AGR for the rule **C**

2. Obtain assumption A_2 from AGEN$_2$ and model check the second premise of **C**. If the check fails, return the obtained counterexample to AGEN$_2$. Otherwise, continue to Step 3.

3. Obtain assumptions A_1 and A_2 from AGEN$_1$ and AGEN$_2$ respectively and verify the last premise of **C**. If the verification succeeds, return TRUE and terminate. Otherwise, continue to Step 4 with counterexample CE.

4. For each $i \in \{1, 2\}$ such that $CE \in \mathbb{L}(WA_i)$, return CE to AGEN$_i$. If no such i exists, return CE as a counterexample witness to $M_1 \parallel M_2 \vDash P$.

We now show that $\mathcal{S}_C$ terminates with the correct result when AGEN$_i$ ($i \in \{1, 2\}$) is instantiated by L^*.

Termination. Similar to the **NC** rule, it can be shown that any run of AGEN$_i$ ($i \in \{1, 2\}$) using $\mathcal{S}_C$ corresponds to a run of L^* with $L(WA_i)$ as the unknown language. Since the latter run must terminate in a finite number of steps with WA_1 and WA_2 as the final result, the strategy $\mathcal{S}_C$ must also terminate with the same result.

Correctness. The strategy $\mathcal{S}_C$ terminates either with an assumption tuple $\langle A_1, A_2 \rangle$ which satisfies the premises of **C** or with a counterexample witness CE. In the first case, correctness follows from the soundness of **C**. In the second case, note that $CE \notin \mathbb{L}(WA_i)$ $(i \in \{1, 2\})$. Therefore, we have $CE \in L(M_1)$ and $CE \in \mathbb{L}(M_2)$ but $CE \notin \mathbb{L}(P)$. Hence CE is a true counterexample.

Remarks.

1. The strategy $\mathcal{S}_C$ can be directly generalized to systems with more than two modules [107].
2. It is sufficient to execute Step 4 for some $i \in \{1, 2\}$ such that $CE \in \mathbb{L}(WA_i)$.

3.4 Discussion

Cobleigh et al. [44] first presented the automated AGR approach using L^* for the rule **NC** in the context of labeled transition systems (both as implementation and specification) using rendezvous-based communication [82] and finite trace-containment as the notion of conformance (cf. Chapter 1). They chose finite automata as the assumption model since these automata cover the set of regular languages (cf. Lemma 8). An explicit state model checker is used to answer membership and candidate queries. This approach was extended to the rule **C** later [17] for the same kind of models.

The initial approach that relies on using machine learning algorithms together with model checking for performing assume-guarantee reasoning, is now viewed as a seminal contribution. The technique has sparked a whole new direction of research at the confluence of model checking and machine learning methodologies. In particular, the problem of performing automated AGR effectively for both hardware and software systems has received considerable attention.

One direction of research focuses on extending the basic methodology to other kinds of systems and specifications. Formally, the automated AGR framework is parameterized by

three items:

- the notion of composition or execution, e.g., synchronous, asynchronous, etc.,
- the kind of communication, e.g., shared memory, message-passing, rendezvous etc., and
- the notion of conformance, e.g., simulation conformance, deadlock checking, ω-trace containment, etc.

While hardware systems are characterized by synchronous composition and shared variable communication, software systems execute asynchronously and use both message-passing and shared memory for communication. Therefore, automated AGR frameworks must be extended to handle systems and properties that are involves different combinations of the above notions of composition, communication and conformance. A major part of this thesis is devoted to investigation of problems of this kind.

Another important problem is to make the automated AGR framework scalable and efficient. Although the learning algorithm makes automated AGR feasible, it does not scale to large systems directly. In particular, the size of alphabets in the assumptions grows exponentially in the number of shared variables. Therefore, a combination of symbolic methods for learning and model checking must be investigated for scaling the technique to real-life software and hardware systems. Moreover, we need to develop techniques that compute the appropriate assumptions quickly, i.e., in small number of iterations.

Besides the task of computing assumptions, an important problem is to obtain a suitable decomposition of a system for using a particular AGR rule. In many cases, the natural decompositions of a system according to its modular syntactic description may not be suitable for compositional reasoning. Therefore, techniques for obtaining good decompositions automatically are required. Finally, techniques for based counterexample-guided automated abstraction have been successful for verifying sequential software programs. Automated AGR can be used together with such abstraction techniques to verify concurrent software.

3.5 History and Related Work

A variety of compositional reasoning methods have been investigated in literature. A comprehensive survey of existing methods can be found in [4, 51].

3.5.1 Compositional Minimization

Traditionally, much of the work on compositional analysis has focused on *compositional reduction* or *minimization* methods. These methods do not avoid computing the global state space of the composed system when checking if $M_1 \parallel M_2 \models P$ holds. Instead, they first compute *smaller* abstractions of one or more components, say A_i for M_i, and then check if $A_1 \parallel A_2 \models P$ holds. These abstractions try to exactly characterize the *useful* and *visible* behaviors of each component in the context of its environment and the property being verified. The method is useful if (i) composing the abstractions leads to a smaller state space and makes the model checking problem more tractable, and (ii) the abstractions are property preserving, i.e., if P holds on $A_1 \parallel A_2$ then P also holds on $M_1 \parallel M_2$.

In order to compute these abstractions, the proposed approaches use either equivalence- or a preorder-based reduction techniques. In [48], Dams et al. use reduction with respect to simulation equivalence to verify properties expressed in the universal fragment of CTL*. Clarke et al. [39] present a method to compute abstractions using user-provided *homomorphism* preorders, which also preserve properties expressed in the universal fragment of CTL*.

Property-driven abstraction. Equivalence-based reductions preserve a whole class of properties, and may not yield a small abstraction in many cases. Therefore, in order to obtain small abstractions, it is important to eliminate the behaviors that are irrelevant with respect to the property being verified. Clarke et al. [38] proposed an approach to obtain abstractions in a *property-directed* manner: if P only refers to visible characteristics of M_2, then they abstract features of M_1 not relevant to its interaction with M_2 and then use

equivalence-based minimization to further reduce the size of M_1 to obtain A_1. The method was applied to verify nexttime-less CTL* properties using the stuttering equivalence [42]. A similar method for general CTL properties was presented in [122]. A general method of obtaining property-preserving abstractions under various notions of composition and preorders was presented in [18, 93].

Context-driven abstraction. Moreover, equivalence-based reductions do not take in account the interaction of the module with its environment. Therefore, the obtained abstractions may still have behaviors that are impossible in the context of the complete system. Graf et al. [70, 71] proposed a method to compute abstraction based on using simulation preorders instead of equivalence. The component behaviors that should not take place in context of the environment components are removed during abstraction computation with the help of interface specifications of the environment components. For systems with more than two components, the components are composed in a stepwise manner and a reduction operation is performed before each composition step. This approach is further developed and automated by Cheung and Kramer [35, 36] by abstracting the environment of a component to obtain interface specifications automatically.

3.5.2 Assume-Guarantee Reasoning

Theoretical investigation of AGR methods can be found in [10, 94, 95, 108, 127]. Henzinger et al. [81] proposed an AGR methodology for verifying shared-memory systems. The tool Mocha [8] supports modular verification of shared-memory programs with requirements expressed in alternating-time temporal logic. The Calvin tool [58, 59] uses an extension of thread-modular reasoning, originally proposed by Jones [88], for the analysis of multi-threaded Java programs. The basic idea of the technique is to verify each thread separately by using an environment assumption to model interleaved steps of other threads. The environment assumption of each thread includes all updates to shared global variables

that may be performed by other threads. For hardware systems, a number of approaches were proposed by McMillan et al. [87, 97, 98, 99, 100, 101, 103].

Computing Assumptions Automatically. All the above approaches had limited impact because of the non-trivial human intervention involved in computing environment assumptions. An initial approach to automate compositional verification by proposed by Alur et al. [7]. Thread-modular model checking is a sound but incomplete approach that was proposed [60] for verifying loosely-coupled software programs (where there is little correlation between the local states of the different threads) based on thread-modular reasoning. This method infers the environment assumption for each thread automatically by first inferring a guarantee for each thread, which consists of all shared variable updates performed by the thread. The disjunction of guarantees of all the other threads forms the environment assumption for a given thread. An approach for combining automated abstraction-refinement with thread-modular model checking was proposed by Henzinger et al. [79] where the assumptions and guarantees are refined in an iterative fashion. Another sound and incomplete method for generating invariants for compositional reasoning was proposed by Jeffords and Heitmeyer [86]. Henzinger et al. [77] proposed a technique for automatic construction of the environment assumption or the context for detecting race conditions in multi-threaded software with unbounded number of threads. Instead of keeping the context information in form of invariants on global variables, they also maintain additional information like control flow location and the number of threads at each location. The context models are iteratively improved using counterexamples and minimized with respect to bisimulation. Recently, a sound and complete approach to thread-modular reasoning has been proposed by Cohen and Namjoshi [46]. An approach to compute the weakest assumptions automatically for labeled transition systems communicating via rendezvous [82] was proposed by Giannakopoulou et al. [64, 65] using set-theoretic construction.

Learning for Assumption Generation. Using machine learning to automatically compute assumptions for AGR was first proposed by Cobleigh et al. [44] for non-circular rules, followed by a method for circular rules [17]. Techniques for applying AGR to source code using assumptions computed at the design-level [66], to predictive testing [23] and concurrent message-passing programs in SPIN [114] were also proposed. The initial methodology was followed by a symbolic approach [115], application to checking component substitutability [31], extensions to different notions of conformance [32, 33], combination with automated system decomposition using hyper-graph partitioning [107], optimized learning and alphabet-enlargement approaches [33, 63], lazy learning approach [124] and a technique for computing minimal assumptions [74]. Cobleigh et al. investigate the advantages of automated AGR methods over monolithic verification techniques in the context of LTSA and FLAVERS tools [45] by experimenting with different two-way system decompositions.

Chapter 4

Checking Component Substitutability

In this chapter, we focus on a particular model checking problem, namely verification of evolving software. Software systems evolve throughout the product life-cycle. For example, any software module (or component) is inevitably transformed as designs take shape, requirements change, and bugs are discovered and fixed. In general such evolution results in the *removal* of previous behaviors from the component and *addition* of new ones. Since the behavior of the updated software component has no direct correlation to that of its older counterpart, substituting it directly can lead to two kinds of problems. First, the removal of behavior can lead to unavailability of previously provided services. Second, the addition of new behavior can lead to violation of global correctness properties that were previously being respected.

In this context, the *substitutability* problem can be defined as the verification of the following two criteria: (i) any *updated portion* of a software system must continue to provide all *services* offered by its earlier counterpart, and (ii) previously established system *correctness properties* must remain valid for the new version of the software system.

4.1 Component Substitutability Check

Model checking can be used at each stage of a system's evolution to solve both the above problems. Conventionally, model checking is applied to the entire system after every update, irrespective of the degree of modification involved. The amount of time and effort required to verify an entire system can be prohibitive and repeating the exercise after each (even minor) system update is therefore impractical. In this chapter, we present an *automated* framework that *localizes* the necessary verification to only modified system components, and thereby reduces dramatically the effort to check substitutability after every system update. Note that our framework is general enough to handle changes in the environment since the environment can also be modeled as a component.

In our framework a component is essentially a C program communicating with other components via blocking message passing. An assembly is a collection of such concurrently executing and mutually interacting components. We will define the notion of a component's behavior precisely later but for now let us denote the set of behaviors of a component C by $Behv(C)$. Given two components C and C' we will write $C \sqsubseteq C'$ to mean $Behv(C) \subseteq Behv(C')$.

Suppose we are given an assembly of components: $\mathcal{C} = \{C_1, \ldots, C_n\}$, and a safety property φ. Now suppose that *multiple* components in $\mathcal{C}$ are upgraded. In other words, consider an index set $\mathcal{I} \subseteq \{1, \ldots, n\}$ such that for each $i \in \mathcal{I}$ there is a *new* component C_i' to be used in place of its *old* version C_i. Our goal is to check the substitutability of C_i' for C_i in $\mathcal{C}$ for every $i \in \mathcal{I}$ with respect to the property φ. Our framework satisfies this goal by establishing the following two tasks:

Containment. Verify, for each $i \in \mathcal{I}$, that every behavior of C_i is also a behavior of C_i', i.e., $C_i \sqsubseteq C_i'$. If $C_i \not\sqsubseteq C_i'$, we also construct a set $\mathcal{F}_i$ of behaviors in $Behv(C_i) \setminus Behv(C_i')$ which will subsequently be used for feedback generation. Note that the upgrade may involve the removal of behaviors designated as errant, say B. In this case, we check $C_i \setminus B \sqsubseteq C_i'$

since behaviors of B will clearly be absent in $C_i^{'}$.

Compatibility. Let us denote by $\mathcal{C}'$ the assembly obtained from $\mathcal{C}$ by replacing the old component C_i with its new version $C_i^{'}$ for each $i \in \mathcal{I}$. Since in general it is not the case that for each $i \in \mathcal{I}$, $C_i^{'} \sqsubseteq C_i$. Therefore, the new assembly $\mathcal{C}'$ may have more behaviors than the old assembly $\mathcal{C}$. Hence $\mathcal{C}'$ might violate φ even though $\mathcal{C}$ did not. Thus, our second task is to verify that $\mathcal{C}'$ satisfies the safety property φ (which would imply that the new components can be safely integrated).

Note that checking compatibility is non-trivial because it requires the verification of a concurrent system where multiple components might have been modified. Moreover, this task is complicated by the fact that our goal is to focus on the components that have been modified.

The component substitutability framework is defined by the following new algorithms: 1) a technique based on simultaneous use of over and under approximations obtained via existential and universal abstractions for the containment check of the substitutable components; 2) a *dynamic* assume-guarantee algorithm developed for the compatibility check. The algorithm is based on automata-theoretic learning for regular sets. It is dynamic in the sense that it learns appropriate environment assumptions for the new components by *reusing* the environment assumptions for their older versions.

In summary, our component substitutability framework has several advantageous features:

- It allows *multiple* components to be upgraded simultaneously. This is crucial since modifications in different components often interact non-trivially to maintain overall system safety and integrity. Hence such modifications must be analyzed jointly.
- It identifies features of an old component which are absent in its updated version. It subsequently generates feedback to localize the modifications required to add the missing features back.

- It is completely automated and uses *dynamic* assume-guarantee style reasoning to scale to large software systems.
- It allows new components to have more behaviors than their old counterparts in order be replaceable. The *extra* behaviors are critical since they provide vendors with the flexibility to implement new features into the product upgrades. Our framework verifies if these new behaviors do not violate previously established global specifications of a component assembly

We have implemented the substitutability framework as part of the ComFoRT [85] reasoning framework. We experimented with an industrial benchmark and report encouraging results in Section 4.5. Section 4.2 defines the notation used in this chapter. Sections 4.3 and 4.4 describe the problem of verification of evolving systems and present a detailed description of the containment and compatibility algorithms that we have developed to overcome difficulties in the verification of evolving programs. Section 4.6 provides an overview of related work, and Section 4.7 summarizes the contributions of this chapter.

4.2 Notation and Background

In this section we present some basic definitions.

Definition 5 (Finite Automaton) *A finite automaton (FA) is a 5-tuple* $(Q, Init, \Sigma, T, F)$ *where (i)* Q *is a finite set of states, (ii)* $Init \subseteq Q$ *is the set of initial states, (iii)* Σ *is a finite alphabet of actions, (iv)* $T \subseteq Q \times \Sigma \times Q$ *is the transition relation, and (v)* $F \subseteq Q$ *is a set of accepting states.*

For any FA $M = (Q, Init, \Sigma, T, F)$, we write $s \xrightarrow{\alpha} s'$ to mean $(s, \alpha, s') \in T$. We define the function δ as follows: $\forall \alpha \in \Sigma \,.\, \forall s \in Q \,.\, \delta(\alpha, s) = \{s' | s \xrightarrow{\alpha} s'\}$. We extend δ to operate on strings and sets of states in the natural manner: for any $\sigma \in \Sigma^*$ and $Q' \subseteq Q$, $\delta(\sigma, Q')$ denotes the set of states of M reached by simulating σ on M starting from any $s \in Q'$.

The *language* accepted by a FA M, denoted by $L(M)$, is defined as follows: $L(M) = \{\sigma \in \Sigma^* \mid \delta(\sigma, Init) \cap F \neq \emptyset\}$. Each element of $L(M)$ is said to be a trace of M.

Definition 6 (Deterministic and Complete Finite Automaton) *A FA* $M = (Q, Init, \Sigma, T, F)$ *is said to be a deterministic FA, or DFA, if* $|Init| = 1$ *and* $\forall \alpha \in \Sigma . \forall s \in Q . |\delta(\alpha, s)| \leq 1$. *Also, M is said to be complete if* $\forall \alpha \in \Sigma . \forall s \in Q . |\delta(\alpha, s)| \geq 1$.

Thus, for a complete DFA, we have the following: $\forall \alpha \in \Sigma . \forall s \in Q . |\delta(\alpha, s)| = 1$. Unless otherwise mentioned, all DFA we consider in the rest of this paper are also complete. It is well-known that a language is regular if and only if it is accepted by some FA (or DFA, since FA and DFA have the same accepting power). Also, every regular language is accepted by a unique (up to isomorphism) minimum DFA. For a DFA M, we denote its complement by $\overline{M}$. Given any FA M, its complement $\overline{M}$ is defined to be $\overline{M'}$ where M' is the DFA obtained from M by the subset construction.

We now define a notion of asynchronous parallel composition between FAs which is based on the notion of composition defined for CSP [118].

Definition 7 (Parallel Composition) *Given two FA* $M_1 = (Q_1, Init_1, \Sigma_1, T_1, F_1)$ *and* $M_2 = (Q_2, Init_2, \Sigma_2, T_2, F_2)$, *their parallel composition* $M_1 \parallel M_2$ *is the FA* $(Q_1 \times Q_2, Init_1 \times Init_2, \Sigma_1 \cup \Sigma_2, T, F_1 \times F_2)$ *such that* $\forall s_1, s_1' \in Q_1 . \forall s_2, s_2' \in Q_2$, $(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')$ *if and only if :*

$$(a) \quad \alpha \in \Sigma_1 \wedge \alpha \notin \Sigma_2 \wedge s_1 \xrightarrow{\alpha} s_1' \wedge (s_2 = s_2') \text{ or,}$$

$$(b) \quad \alpha \in \Sigma_2 \wedge \alpha \notin \Sigma_1 \wedge s_2 \xrightarrow{\alpha} s_2' \wedge (s_1 = s_1') \text{ or,}$$

$$(c) \quad \alpha \in (\Sigma_1 \cap \Sigma_2) \wedge \forall i \in \{1, 2\} \; s_i \xrightarrow{\alpha} s_i'.$$

Given a string t, we write $M \parallel t$ to denote the composition of M with the automaton representation of t.

Definition 8 (Language Containment) *For any FA* M_1 *and* M_2 *(with alphabets* Σ_1 *and* Σ_2 *respectively, where* $\Sigma_2 \subseteq \Sigma_1$*), we write* $M_1 \sqsubseteq M_2$ *to mean* $L(M_1 \parallel \overline{M_2}) = \emptyset$. *A counterexample to* $M_1 \sqsubseteq M_2$ *is a string* $\sigma \in L(M_1 \parallel \overline{M_2})$.

If $M_1 \sqsubseteq M_2$, then we sometimes also say that M_2 is an abstraction of M_1. Recall (cf. Chapter 3) that for any FA M and any safety property expressed as a FA φ, the weakest (i.e., maximal w.r.t. the language-containment preorder $\sqsubseteq$) assumption FA, denoted by WA, is defined as follows: (i) $M \parallel WA \sqsubseteq \varphi$ and (ii) for any FA E, $M \parallel E \sqsubseteq \varphi$ iff $E \sqsubseteq WA$. Also, the weakest assumption WA exists and can be represented by a FA accepting the language $L(\overline{M \parallel \overline{\varphi}})$.

4.3 Containment Analysis

Recall that the containment step verifies for each $i \in \mathcal{I}$, that $C_i \sqsubseteq C_i^{'}$, i.e., every behavior of C_i is also a behavior of $C_i^{'}$. If $C_i \not\sqsubseteq C_i^{'}$, we also generate a counterexample behavior in $Behv(C_i) \setminus Behv(C_i^{'})$ which is subsequently provided as user feedback. This containment check is performed as depicted in Figure 4.1 for each modified component. (CE refers to the counterexample generated during the verification phase). For each $i \in \mathcal{I}$, the containment check proceeds as follows:

1. Abstraction. Construct finite models M and M' such that the following conditions **C1** and **C2** hold:

$$(\mathbf{C1})\ C_i \sqsubseteq M \qquad (\mathbf{C2})\ M' \sqsubseteq C_i^{'} \tag{4.1}$$

Here M is an *over-approximation* of C_i and can be constructed by standard predicate abstraction [**?**]. M' is constructed from $C_i^{'}$ via a modified predicate abstraction which produces an *under-approximation* of its input C component. We now describe the details of the abstraction steps.

Suppose that C_i consists of a set of C statements $Stmt = \{st_1, \ldots, st_k\}$. Let V be the set of variables in the C_i. A valuation of all the variables in a program corresponds to a concrete state of the given program. We denote it by $\bar{v}$.

Predicates are functions that map a concrete state $\bar{v} \in S$ into a Boolean value. Let

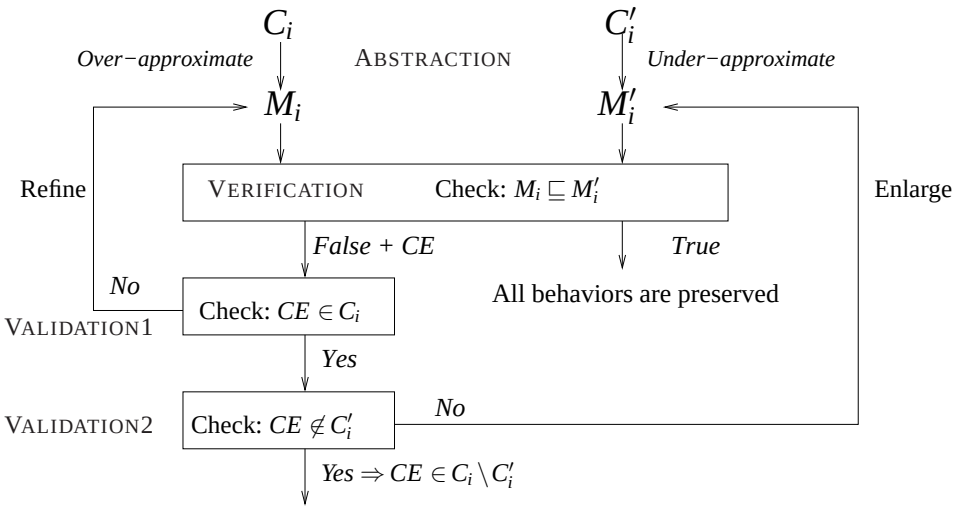


Figure 4.1: The containment phase of the substitutability framework.

$\mathcal{P} = \{\pi_1, \ldots, \pi_k\}$ be the set of predicates over the given program. On evaluating the set of predicates in $\mathcal{P}$ in a particular concrete state $\bar{v}$, we obtain a vector of Boolean values $\bar{b}$, where $\bar{b}[i] = \pi_i(\bar{v})$. The Boolean vector $\bar{b}$ represents an abstract state and we denote this operation by an abstraction function α: $\bar{b} = \alpha(\bar{v})$. The *concretization* function γ is defined as follows:

$$\gamma(\bar{b}) = \{\bar{v} \mid \bar{b} = \alpha(\bar{v})\}$$

May Predicate Abstraction: Over-approximation. This step corresponds to the standard predicate abstraction. Each statement (or basic block) St in C_i is associated with a transition relation $T(\bar{v}, \bar{v}')$. Here, $\bar{v}$ and $\bar{v}'$ represent a concrete state before and after execution of St, respectively. Given the set of predicates $\mathcal{P}$ and associated vector of Boolean variables $\bar{b}$ as before, we compute an abstract transition relation $\hat{T}(\bar{b}, \bar{b}')$ [39] as follows:

$$\hat{T}(\bar{b}, \bar{b}') \equiv \exists \bar{v}, \bar{v}' \,.\, T(\bar{v}, \bar{v}') \wedge \bar{b} = \alpha(\bar{v}) \wedge \bar{b}' = \alpha(\bar{v}') \tag{4.2}$$

$\hat{T}$ is the existential abstraction [39] of T (with respect to the abstraction function α) and is

also referred to as its *may* abstraction $\hat{T}_{may}$ [123]. In practice, we compute this abstraction using the weakest precondition (WP) transformer [52] on predicates in $\mathcal{P}$ along with an automated theorem prover [69] as follows:

$$\hat{T}(\bar{b}, \bar{b}') \equiv \gamma(\bar{b}) \wedge WP(St, \gamma(\bar{b}')) \text{ is satisfiable} \tag{4.3}$$

where $WP(St, \phi)$ denotes the weakest precondition expression for formula ϕ with respect to statement St and γ is the concretization function as defined above. Note that Equation 4.2 is equivalent to Equation 4.3 since:

$$\hat{T}(\bar{b}, \bar{b}') \equiv \exists \bar{v} \,.\, \exists \bar{v}' \,.\, T(\bar{v}, \bar{v}') \wedge \bar{b} = \alpha(\bar{v}) \wedge \bar{b}' = \alpha(\bar{v}')$$

$$\equiv \exists \bar{v} \,.\, (\bar{v} \in \gamma(\bar{b}) \wedge \exists \bar{v}' \,.\, T(\bar{v}, \bar{v}') \wedge \bar{v}' \in \gamma(\bar{b}'))$$

$$\equiv \exists \bar{v} \,.\, (\bar{v} \in \gamma(\bar{b}) \wedge \bar{v} \in WP(St, \gamma(\bar{b}')))$$

$$\equiv \gamma(\bar{b}) \wedge WP(St, \gamma(\bar{b}')) \text{ is satisfiable}$$

Note that it is sufficient to use standard weakest preconditions for sequential programs since the abstraction is performed component-wise.

Must Predicate Abstraction: Under-approximation. The modified predicate abstraction constructs an under-approximation of the concrete system via universal or *must* [123] abstraction. Given a statement St in the modified component C_i' and its associated transition relation $T(\bar{v}, \bar{v}')$ as before, we compute its must abstraction with respect to predicates $\mathcal{P}$ as follows:

$$\hat{T}(\bar{b}, \bar{b}') \equiv \forall \bar{v} \,.\, \bar{b} = \alpha(\bar{v}) \implies \exists \bar{v}' \,.\, T(\bar{v}, \bar{v}') \wedge \bar{b}' = \alpha(\bar{v}') \tag{4.4}$$

We use $\hat{T}_{must}$ to denote the above relation. Note that $\hat{T}_{must}$ contains a transition from an

abstract state $\bar{b}$ to $\bar{b}'$ iff for every concrete state $\bar{v}$ corresponding to $\bar{b}$, there exists a concrete transition to a state $\bar{v}'$ corresponding to $\bar{b}'$ [123]. Further, it has been shown [123] that the concrete transition relation T simulates the abstract transition relation $\hat{T}_{must}$. Hence, $\hat{T}_{must}$ is an under-approximation of T. Again, in practice, we compute $\hat{T}_{must}$ using the WP transformer on the predicates together with a theorem prover [75] in the following way:

$$\hat{T}(\bar{b}, \bar{b}') \equiv (\gamma(\bar{b}) \implies WP(St, \gamma(\bar{b}'))) \tag{4.5}$$

Note that Equation 4.4 is equivalent to Equation 4.5 since:

$$\hat{T}(\bar{b}, \bar{b}') \equiv (\forall \bar{v} \centerdot \bar{b} = \alpha(\bar{v}) \implies \exists \bar{v}' \centerdot T(\bar{v}, \bar{v}') \wedge \bar{b}' = \alpha(\bar{v}'))$$

$$\equiv (\forall \bar{v} \centerdot \bar{v} \in \gamma(\bar{b}) \implies \exists \bar{v}' \centerdot T(\bar{v}, \bar{v}') \wedge \bar{v}' \in \gamma(\bar{b}'))$$

$$\equiv (\forall \bar{v} \centerdot \bar{v} \in \gamma(\bar{b}) \implies \bar{v} \in WP(St, \gamma(\bar{b}')))$$

$$\equiv (\gamma(\bar{b}) \implies WP(St, \gamma(\bar{b}')))$$

At the end of the abstraction phase, we obtain M as an over-approximation of C_i and M' as an under-approximation of $C_i^{'}$, as defined in Equation 4.1. The containment check now proceeds to the next stage involving verification.

2. Verification. Verify if $M \sqsubseteq M'$ (or alternatively $M \setminus B \sqsubseteq M'$ if the upgrade involved some bug fix and the bug was defined as a finite automaton B). If so then from **(C1)** and **(C2)** (cf. **Abstraction**) above we know that $C_i \sqsubseteq C_i^{'}$ and we terminate with success. Otherwise we obtain a counterexample CE.

3. Validation and Refinement 1. Check that CE is a real behavior of C_i. This step is done in a manner similar to the counterexample validation techniques employed in software model checkers based on CEGAR [15, 30, 80]. If CE is a real behavior of C_i, we proceed to the Step 4. Otherwise we refine model M (i.e., remove the spurious CE) by

constructing a new set of predicates $\mathcal{P}'$ and repeat from Step 2. The procedure for refining the model M has been presented elsewhere [30] in detail, and we do not describe it here further.

4. Validation and Refinement 2. Check that CE is *not* a real behavior of $C_i^{'}$. The operations involved in this check are the same as those used for the validation check in Step 3. The only difference is that we complement the final result, since in this step we are interested in checking whether CE **is not** a real behavior of $C_i^{'}$, while in Step 3, we were interested in checking whether CE **is** a real behavior of C_i.

If CE is not a real behavior of $C_i^{'}$, we know that $CE \in Behv(C_i) \setminus Behv(C_i^{'})$. We add CE to the user feedback step and stop. Otherwise we enlarge M' (i.e., add CE) by constructing a new set of predicates $\mathcal{P}'$ and repeat from Step 2. The procedure for enlarging the model M' has been presented elsewhere [75] in detail, and we do not describe it here further.

Figure 4.1 depicts the individual steps of this containment check. Similar to ordinary abstraction-refinement procedures for programs, the containment check may not terminate because a sufficient set of predicates is never found. Otherwise, the check terminates either with a successful result (all behaviors of C_i are verified to be present in $C_i^{'}$) or returns an actual diagnostic behavior CE as a feedback to the developers. The following theorem proves this result.

Theorem 4 (Correctness of Containment Check) *Upon termination, if the Containment Check is successful, then $C_i \sqsubseteq C_i^{'}$ holds. Otherwise, a witness counterexample $CE \in C_i \setminus C_i^{'}$ is returned.*

Proof. The containment check terminates either when the verification check (Step 2) succeeds or both the Validation and Refinement checks (Steps 3 and 4) fail. Note that at each iteration $C_i \sqsubseteq M_i$ and $M_i' \sqsubseteq C_i^{'}$. Now, if the verification step (Step 2) succeeds, then it follows that $M_i \sqsubseteq M_i'$, and hence $C_i \sqsubseteq M_i \sqsubseteq M_i' \sqsubseteq C_i^{'}$. Therefore, $C_i \sqsubseteq C_i^{'}$ holds.

Otherwise, suppose that both the Validation and Refinement phases (Steps 3 and 4) fail. Then, from Step 3 we know that $CE \in C_i$, and from Step 4 we know that $CE \notin C_i^{'}$. Hence, we have a counterexample $CE \in C_i \setminus C_i^{'}$ which is returned by the containment check. This concludes the proof.

□

4.3.1 Feedback

Recall that for some $i \in \mathcal{I}$, if our containment check detects that $C_i \not\sqsubseteq C_i^{'}$, it also computes a set $\mathcal{F}_i$ of erroneous behaviors. Intuitively, each element of $\mathcal{F}_i$ represents a behavior of C_i that is not a behavior of $C_i^{'}$. We now present our process of generating feedback from $\mathcal{F}_i$. In the rest of this section, we write $C, C^{'}$, and $\mathcal{F}$ to mean C_i, $C_i^{'}$, and $\mathcal{F}_i$, respectively.

Consider any behavior π in $\mathcal{F}$. Recall that π is a trace of an automaton M obtained by predicate abstraction of C. By simulating π on M, we construct a sequence $Rep(\pi) = \langle \alpha_1, \ldots, \alpha_n \rangle$ of states and actions of M corresponding to π.

We also know that π represents an actual behavior of C but not an actual behavior of $C^{'}$. Thus, there is a prefix $Pref(\pi)$ of π such that $Pref(\pi)$ represents a behavior of $C^{'}$. However, any extension of $Pref(\pi)$ is no longer a valid behavior of $C^{'}$. Note that $Pref(\pi)$ can be constructed by simulating π on $C^{'}$. Let us denote the suffix of π after $Pref(\pi)$ by $Suff(\pi)$. Since $Pref(\pi)$ is an actual behavior of $C^{'}$, we can also construct a representation for $Pref(\pi)$ in terms of the statements and predicate valuations of $C^{'}$. Let us denote this representation by $Rep'(Pref(\pi))$.

As our feedback, for each $\pi \in \mathcal{F}$, we compute the following representations: $Rep(Pref(\pi))$, $Rep(Suff(\pi))$, and $Rep'(Pref(\pi))$. Such feedback allows us to identify the exact divergence point of π beyond which it ceases to correspond to any concrete behavior of $C^{'}$. Since the feedback refers to a program statement, it allows us to understand at the source code level why C is able to match π completely, but $C^{'}$ is forced to diverge

from π beyond $Pref(\pi)$. This understanding makes it easier to modify C' so that the missing behavior π can be added back to it.

4.4 Compatibility Analysis

The compatibility check is aimed at ensuring that the upgraded system satisfies global safety specifications. Our compatibility check procedure involves two key paradigms: dynamic regular-set learning and assume-guarantee reasoning. We first present these two techniques and then describe their use in the compatibility algorithm.

4.4.1 Dynamic Regular-Set Learning

Central to our compatibility check procedure is a new *dynamic* algorithm to learn regular languages. Our algorithm is based on the L^* algorithm described in Section 3. In this section we first present a dynamic version of the L^* learning algorithm and then describe how it can be applied for checking compatibility.

Dynamic L^*.

Normally L^* initializes with $S = E = \{\epsilon\}$. This can be a drawback in cases where a previously learned candidate (and hence a table) exists and we wish to restart learning using information from the previous table. In the following discussion, we show that if L^* begins with any non-empty valid table, it must terminate with the correct result (Theorem 5). In particular, this theorem allows us to perform our compatibility check dynamically by restarting L^* with any previously computed table by revalidating it instead of starting from an empty table.[1]

Definition 9 (Agreement) *An observation table $\mathcal{T} = (S, E, T)$ is said to agree with a*

[1]A similar idea was also proposed in the context of adaptive model checking [72].

regular language U iff:

$$\forall (s, e) \in (S \cup S \cdot \widehat{\Sigma}) \times E \ . \ T(s, e) = 1 \equiv s \cdot e \in U$$

Definition 10 (Validity) *Recall the notion of a well-formed observation table from Section 2.3.1. An observation table $\mathcal{T} = (S, E, T)$ is said to be valid for a language U iff $\mathcal{T}$ is well-formed and agrees with U. Moreover, we say that a candidate automaton derived from a table $\mathcal{T}$ is valid for a language U if $\mathcal{T}$ is valid for U.*

Theorem 5 *L^* terminates with a correct result for any unknown language U starting from any valid table for U.*

Proof. It was shown earlier (cf. Theorem 1) that for a given unknown language U, the L^* algorithm terminates if it is able to perform a finite number of candidate queries. Therefore, it remains to show that starting from a valid observation table, the algorithm must be able to perform a candidate query in a finite number of steps. Now, note that each iteration of the L^* algorithm involves executing the **CloseTable** and `MkDFA` procedures before making a candidate query (cf. Figure 2.1). Therefore, we need to show that the procedures **CloseTable** and `MkDFA` terminate in a finite number of steps starting from a valid table.

Let the valid observation table be $\mathcal{T}_1$. Since $\mathcal{T}_1$ agrees with U, the **CloseTable** procedure terminates in a finite number of steps with a closed table $\mathcal{T}_2$ (cf. Lemma 3). Moreover, $\mathcal{T}_2$ is well-formed since the initial table $\mathcal{T}_1$ is well-formed (cf. Lemma 3). Since $\mathcal{T}_2$ is well-formed and closed, the `MkDFA` algorithm is able to compute a DFA candidate D (cf. Lemma 4) from $\mathcal{T}_2$ and terminates. Therefore, after the execution of `MkDFA` finishes, L^* must perform a candidate query.

□

Suppose we have a table $\mathcal{T}$ that is valid for an unknown language U, and we have a new unknown language U' different from U. Suppose we want to learn U' by starting L^* with table $\mathcal{T}$. Note that since U and U' differ in general, $\mathcal{T}$ may not agree with U' and hence

may not be valid with respect to U'; hence, starting from $\mathcal{T}$ is not appropriate. Thus, we first *revalidate* $\mathcal{T}$ against U' and then start L^* from the valid $\mathcal{T}$. Theorem 5 provides the key insight behind the correctness of this procedure. As we shall see, this idea forms the backbone of our dynamic compatibility-check procedure (see Section 4.4.3).

In the context of assume-guarantee reasoning, U represents a weakest assumption language. When an upgrade occurs, U may change to a different language U'. However, since the change was caused by an upgrade, we expect that the language U' will differ from U only slightly. We will see that the efficiency of our revalidation procedure depends crucially on this hypothesis.

Revalidation Procedure. Suppose we have a table $\mathcal{T}$ which is valid for an unknown language U. Given a Teacher for a different unknown language U', the table revalidation procedure **Reval** (shown in Figure 4.2) makes $\mathcal{T}$ valid with respect to U' by executing the following two steps. In Step 1, **Reval** updates all the table entries in $\mathcal{T}$ by asking membership queries. The table $\mathcal{T}'$ obtained as a result may not be well-formed since the function T is updated. More precisely, for some $s_1, s_2 \in S$ where $s_1 \not\equiv s_2$ in $\mathcal{T}$, it may happen that $s_1 \equiv s_2$ in $\mathcal{T}'$. However, the construction of a candidate DFA requires that the observation table be well-formed (cf. Lemma 4). Therefore, in Step 2, **Reval** uses the procedure **MkWellFormed** to make $\mathcal{T}'$ well-formed. In order to describe **MkWellFormed**, we need the concepts of the *well-formed cover* and the *experiment cover* for an observation table $\mathcal{T}$.

Procedure Reval
Input: An observation table $\mathcal{T} = (S, E, T)$ and a teacher for a language U'.
Output: An observation table $\mathcal{T}'$ that is valid for U'.

1. **(Step 1)** For all $s \in S$ and $e \in E$, ask membership query for $s \cdot e$ with respect to U' and update T.
   Let the table obtained as a result be $\mathcal{T}'$.
2. **(Step 2)** Make $\mathcal{T}'$ well-formed (cf. Section 2.3.1) by using the procedure **MkWellFormed**.

Figure 4.2: The table revalidation procedure **Reval**.

Definition 11 (Well-formed Cover) *Given a prefix-closed set S, a well-formed subset of S is a set $S' \subseteq S$ such that (i) S' is prefix-closed, and (ii) for all $s_1, s_2 \in S'$, $s_1 \not\equiv s_2$ holds. A well-formed cover S' of S is a maximal well-formed subset of S.*

Given a prefix-closed set S, a well-formed cover S' of S can be obtained by performing a depth-first tree search on the tree representation of S in the following way: for each newly visited node in the tree, the corresponding string in S is added to S'. However, a node (with the corresponding string s) is visited only if for all s' in the current cover S', s and s' are non-equivalent, i.e., $s \not\equiv s'$. The search terminates when for every $s \in S$ there exists some $s' \in S'$ so that $s \equiv s'$. Note that the final S' obtained in this way is prefix-closed and no two elements of S' are equivalent. For example, let $S = \{a, a \cdot b, a \cdot c, d\}$ where $a \equiv a \cdot c$ and $d \equiv a \cdot b$. A well-formed cover of S is $S' = \{a, a \cdot b\}$. Note that S' is prefix-closed and $a \not\equiv a \cdot b$.

Definition 12 (Column Function) *Given an observation table $\mathcal{T} = (S, E, T)$, and some $e \in E$, $Col(e)$ is defined to be a function from $(S \cup S \cdot \widehat{\Sigma})$ to $\{0, 1\}$ such that $Col(e)(s) = T(s, e)$ for all $s \in (S \cup S \cdot \widehat{\Sigma})$. For $e_1, e_2 \in E$, we say that $Col(e_1) = Col(e_2)$ if for all $s \in (S \cup S \cdot \widehat{\Sigma})$, $T(s, e_1) = T(s, e_2)$.*

Intuitively, for an experiment $e \in E$, $Col(e)$ denotes the vector of Boolean values in the column corresponding to e in an observation table $\mathcal{T}$. Two elements e_1 and e_2 are equivalent under the Col function if the vector of Boolean values in the corresponding columns of the observation table are same.

Definition 13 (Experiment Cover) *An experiment cover of E is a set $E' \subseteq E$, such that (i) for all $e_1, e_2 \in E'$, $Col(e_1) \neq Col(e_2)$, and (ii) for each $e \in E$, there exists an $e' \in E'$, such that $Col(e) = Col(e')$.*

An experiment cover for E can be obtained by finding the set of elements equivalent under Col function and picking a representative element from each set. For example, consider the observation table in Figure 4.3(d). Here, $E = \{\epsilon, \alpha\}$. Note that $Col(\epsilon) \neq$

$Col(\alpha)$. Hence, the experiment cover E' for E is the same as E.

The **MkWellFormed** procedure is described by the pseudo-code in Figure 4.4. Intuitively, the procedure removes duplicate elements from S (which are equivalent under the $\equiv$ relation) and E (having the same value under the Col function).

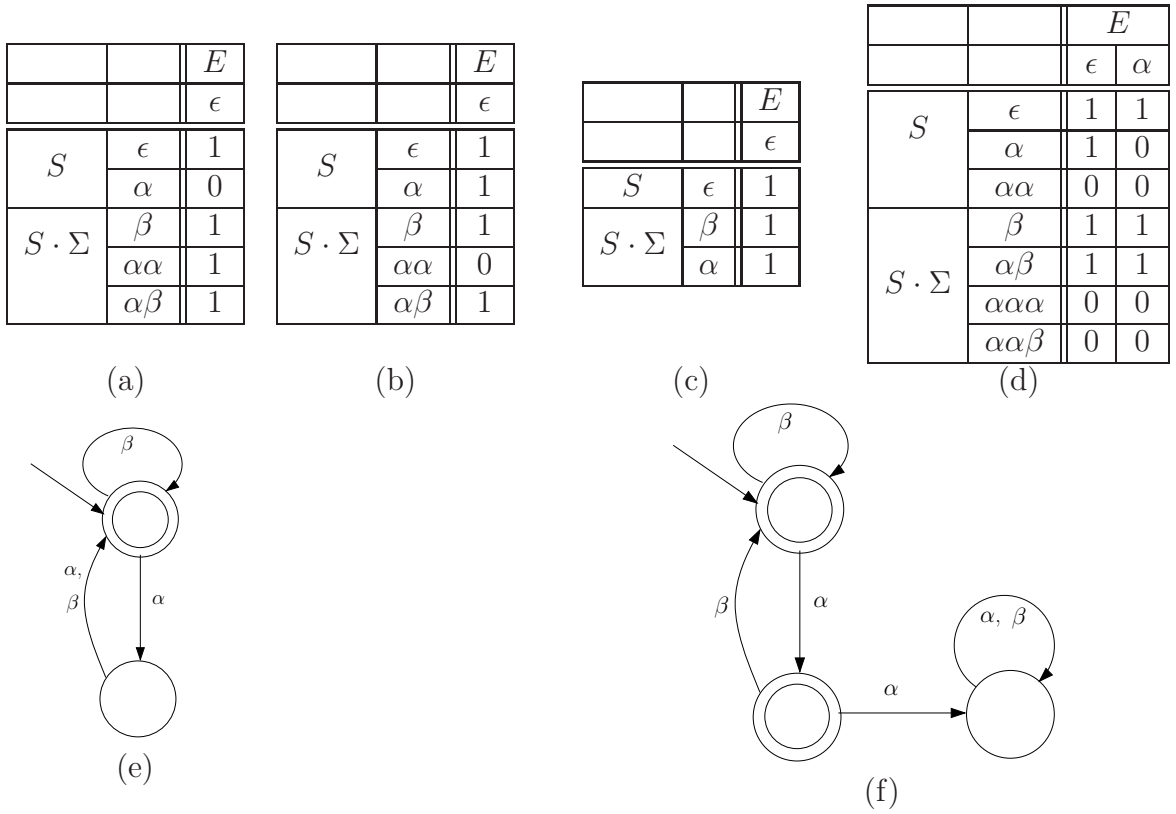


Figure 4.3: Illustration of the revalidation procedure described in Example below; (a) Observation table for original language $U = (\beta \mid (\alpha \cdot (\alpha|\beta)))^*$; (b) New observation table after recomputing the entries with respect to the new language $U' = ((\beta \mid \alpha \cdot \beta)^*) \mid ((\beta \mid \alpha \cdot \beta)^* \cdot \alpha)$; e.g., $\alpha \in U'$ implies $T(\alpha, \epsilon) = 1$ (c) Observation table after revalidating with respect to U' and (d) after an L^* learning iteration with respect to U'; (e) DFA for language U (corresponding to observation table in (a)); and (f) DFA for language U' (corresponding to table in (d)).

(Revalidation Example) Figure 4.3 shows an illustration of the revalidation procedure in the dynamic L^* algorithm. Let the initial unknown language (the weakest assumption language) $U = (\beta \mid (\alpha \cdot (\alpha|\beta)))^*$. The observation table $\mathcal{T}_1$ and the DFA for U are shown in Figure 4.3(a) and Figure 4.3(e) respectively. Suppose that an upgrade happens

and the new weakest assumption language $U' = ((\beta \mid \alpha \cdot \beta)^*) \mid ((\beta \mid \alpha \cdot \beta)^* \cdot \alpha)$. In particular, note that $\alpha \in U'$ but not in U and $\alpha \cdot \alpha \in U$ but not in U'. Our goal is to start learning with respect to U' from the observation table $\mathcal{T}_1$ computed for U previously. So, the **Reval** procedure is applied to $\mathcal{T}_1$. Figure 4.3(b) shows the table obtained after applying the Step 1 of the revalidation procedure with respect to the new language U'. Note that the entries for $T(\alpha, \epsilon)$ and $T(\alpha \cdot \alpha, \epsilon)$ are updated with respect to U'. This, in turn, results in $\alpha \equiv \epsilon$ (cf. Figure 4.3(b)). Now, the Step 2 of the **Reval** procedure is applied: since $\alpha \equiv \epsilon$ and $S = \{\epsilon, \alpha\}$, the well-formed cover $S' = \{\epsilon\}$. The experiment cover E' remains the same as E. Hence, α is removed from S during computation of the well-formed cover in this step (Note that the extensions $\alpha \cdot \alpha$ and $\alpha \cdot \beta$ are also in turn removed from $S \cdot \widehat{\Sigma}$). The resultant observation table (after making it closed) is shown in Figure 4.3(c). Since this table is closed, learning proceeds in the normal fashion from here by computing the next candidate and making a candidate query. Figure 4.3(d) shows the final observation table and Figure 4.3(f) shows the DFA obtained after learning completes with respect to U'.

Note that our example is small, and therefore the revalidation step gives rise to a trivial intermediate observation table (Figure 4.3(b)). However, as noted earlier, in the case when an upgrade causes the change from U to U', the languages U and U' may differ only slightly. Therefore, in this case, the **Reval** procedure may modify the observation table only slightly. In particular, during revalidation, the well-formed cover of S may remain very similar to S (i.e., a large number of elements of S may continue to remain non-equivalent after revalidation), leading to reuse of information about many traces ($S \cdot E$) in the observation table. In the experimental evaluation of our approach, we observed that the above expectation was true in most of the cases.

We now show that the output of **MkWellFormed** procedure is a well-formed table.

Lemma 9 *The* **MkWellFormed** *procedure returns a well-formed observation table.*

Procedure MkWellFormed
Input: Observation table $\mathcal{T} = (S, E, T)$
Output: Well-formed observation table $\mathcal{T}' = (S', E', T')$

1. Set S' to a well-formed cover (cf. Definition 11) of S.
2. Set E' to an experiment cover (cf. Definition 13) of E with respect to $(S' \cup S' \cdot \widehat{\Sigma}')$.
3. Obtain T' by restricting T to $(S' \cup S' \cdot \widehat{\Sigma}') \times E'$

Figure 4.4: Pseudo-code for the **MkWellFormed** procedure

Proof. Given an observation table $\mathcal{T} = (S, E, T)$, the **MkWellFormed** procedure restricts S to a well-formed cover (say S') and E to an experiment cover (say E'). Let the table obtained as a result be $\mathcal{T}'$. It follows from Definition 11 that for all s_1,$s_2 \in S'$, $s_1 \not\equiv s_2$. Using the definition of $\equiv$ (cf. Section 2.3.1), we know that for some $e \in E$, $T(s_1 \cdot e) \neq T(s_2 \cdot e)$. Now, consider the following two cases:

Case 1. If $e \in E'$, $s_1 \not\equiv s_2$ still holds in the result table since $T(s_1 \cdot e) \neq T(s_2 \cdot e)$.

Case 2. Otherwise, $e \notin E'$. However, by Definition 13, there exist some $e' \in E'$, so that $Col(e') = Col(e)$. By using the definition of Col (Definition 12), it follows that for all $s \in S$, $T(s \cdot e) = T(s \cdot e')$. Hence, $T(s_1 \cdot e') = T(s_1 \cdot e) \neq T(s_2 \cdot e) = T(s_2 \cdot e')$. Therefore, $s_1 \not\equiv s_2$ holds and so the output table $\mathcal{T}'$ is well-formed.

□

Lemma 10 *The* **Reval** *procedure always computes a valid observation table for the unknown language U' as an output.*

Proof. Refer to Figure 4.2 describing the **Reval** procedure. By construction, the table obtained at the end of Step 1 must *agree* with U'. In Step 2, the procedure **MkWellFormed** is applied. Therefore, it follows from Lemma 9 that the resultant table is *well-formed*. As a result, the final table both agrees with U' and is well-formed; hence, by Definition 10, it is valid.

□

It follows from Lemma 10 and Theorem 5 that starting from an observation table computed by the **Reval** procedure, the L^* algorithm must terminate with the correct minimum DFA for an unknown language U'.

4.4.2 Assume-Guarantee Reasoning

Along with dynamic L^*, we use assume-guarantee style compositional reasoning (cf. Chapter 3 to check compatibility. Given a set of component finite automata $M_1, \ldots, M_n$ and a specification automaton φ, the non-circular rule **NC** (cf. Chapter 3) can be used to verify $M_1 \parallel \cdots \parallel M_n \sqsubseteq \varphi$:

$$\frac{M_1 \parallel A_1 \sqsubseteq \varphi \qquad M_2 \parallel \cdots \parallel M_n \sqsubseteq A_1}{M_1 \parallel \cdots \parallel M_n \sqsubseteq \varphi}$$

As discussed in Chapter 3, the second premise is itself an instance of the top-level proof obligation with $n-1$ component finite automata. Hence, the rule **NC** can be instantiated in a recursive manner for n components [43] in the following way.

$$\frac{M_i \parallel A_i \sqsubseteq A_{i-1} (1 \leq i \leq n-1, A_0 = \varphi) \qquad M_n \sqsubseteq A_{n-1}}{M_1 \parallel .. \parallel M_n \sqsubseteq \varphi}$$

We will see later that our algorithm for checking compatibility uses this instantiation of rule **NC** for n components. We can show that this rule is *complete* using the notion of weakest assumptions. Recall that for any finite automaton M and a specification automaton φ, there must exist a weakest finite automaton assumption WA such that $M \parallel A \sqsubseteq \varphi$ iff $A \sqsubseteq WA$ and $M \parallel WA \sqsubseteq \varphi$. For the above instantiation of **NC** rule, we can define a set of weakest assumptions WA_i $(1 \leq i \leq n-1)$ as follows. It is clear that a weakest assumption WA_1 exists such that $M_1 \parallel WA_1 \sqsubseteq \varphi$. Given WA_1, it follows that WA_2 must exist so that $M_2 \parallel WA_2 \sqsubseteq WA_1$. Therefore, by induction on i, there must exist weakest assumptions WA_i for $1 \leq i \leq n-1$, such that $M_i \parallel WA_i \sqsubseteq WA_{i-1} (1 \leq i \leq n-1, WA_0 = \varphi)$

and $M_n \sqsubseteq A_{n-1}$.

4.4.3 Compatibility Check for C Components

The procedure for checking compatibility of new components in the context of the original component assembly is presented in Figure 4.5. Given an old component assembly $\mathcal{C} = \{C_1, \ldots, C_n\}$ and a set of new components $\mathcal{C}' = \{C'_i \mid i \in \mathcal{I}\}$ (where $\mathcal{I} \subseteq \{1, \ldots, n\}$), the compatibility-check procedure checks if a safety property φ holds in the new assembly. We first present an overview of the compatibility procedure and then discuss its implementation in detail. The procedure uses a **DynamicCheck** algorithm (cf. Section 4.4.3) and is done in an iterative abstraction-refinement style as follows:

1. Use predicate abstraction to obtain finite automaton models M_i, where M_i is constructed from C_i if $i \notin \mathcal{I}$ and from C'_i if $i \in \mathcal{I}$. The abstraction is carried out component-wise. Let $\mathcal{M} = \{M_1, \ldots, M_n\}$.
2. Apply **DynamicCheck** on $\mathcal{M}$. If the result is TRUE, the compatibility check terminates successfully. Otherwise, we obtain a counterexample CE.
3. Check if CE is a valid counterexample. Once again this is done component-wise. If CE is valid, the compatibility check terminates unsuccessfully with CE as a counterexample. Otherwise we go to the next step.
4. Refine a specific model, say M_k, such that the spurious CE is eliminated. Repeat the process from Step 2.

Overview of DynamicCheck.

We first present an overview of the algorithm for two finite automata and then generalize it to an arbitrary collection of finite automata. Suppose we have two old finite automata, M_1 and M_2, and a property finite automaton φ. We assume that we previously tried to verify $M_1 \parallel M_2 \sqsubseteq \varphi$ using **DynamicCheck**. The algorithm **DynamicCheck** uses dynamic L^*

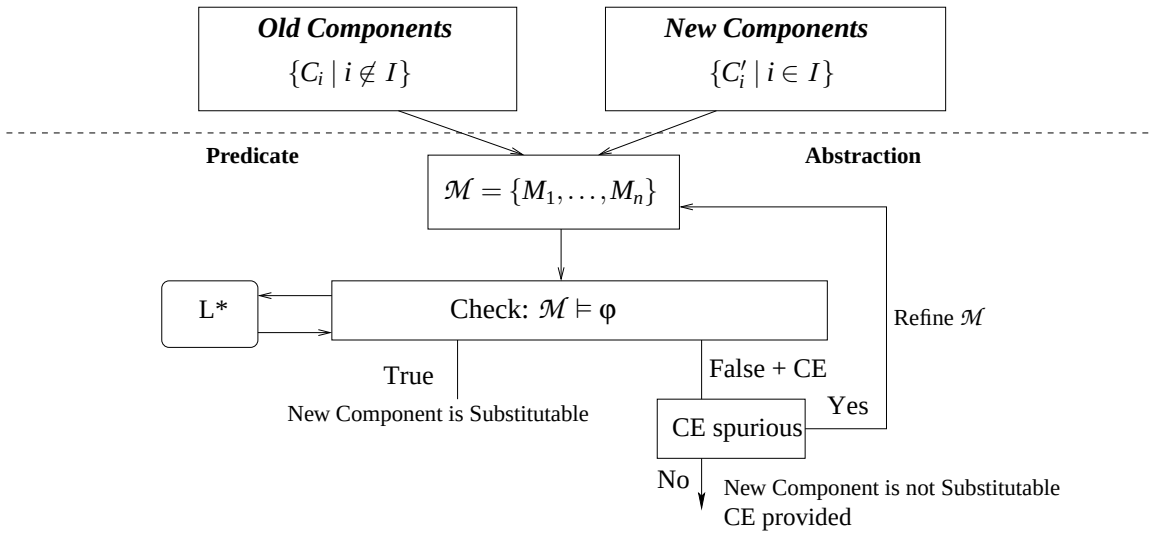


Figure 4.5: The Compatibility Phase of the Substitutability Framework

to learn appropriate assumptions that can discharge the premises of **NC**. In particular, suppose that while trying to verify $M_1 \parallel M_2 \sqsubseteq \varphi$, **DynamicCheck** had constructed an observation table $\mathcal{T}$.

Now suppose that we have new versions M_1' and M_2' for M_1 and M_2. Note that, in general, either M_1' or M_2' could be identical to its old version. **DynamicCheck** now reuses $\mathcal{T}$ and invokes the dynamic L^* algorithm to automatically learn an assumption A' such that (i) $M_1' \parallel A' \sqsubseteq \varphi$ and (ii) $M_2' \sqsubseteq A'$. More precisely, **DynamicCheck** proceeds iteratively as follows:

1. It checks if $M_1 = M_1'$. If so, it starts learning from the previous table $\mathcal{T}$ (i.e., it sets $\mathcal{T}' := \mathcal{T}$). Otherwise, it revalidates $\mathcal{T}$ against M_1' to obtain a new table $\mathcal{T}'$.

2. It derives a conjecture A' from $\mathcal{T}'$ and checks if $M_2' \sqsubseteq A'$. If this check passes, it terminates with TRUE and the new assumption A'. Otherwise, it obtains a counterexample CE.

3. It analyzes CE to see if CE corresponds to a real counterexample to $M_1' \parallel M_2' \sqsubseteq \varphi$. If so, it constructs such a counterexample and terminates with FALSE. Otherwise, it adds a new experiment to $\mathcal{T}'$ using CE. This is done via the algorithm by Rivest

and Schapire [117] as explained in Chapter 2. Therefore, once the new experiment is added, $\mathcal{T}'$ is no longer closed.

4. It makes $\mathcal{T}'$ closed by making membership queries and repeats the process from Step 2.

We now describe the key ideas that enable us to reuse the previous assumptions and then present the complete **DynamicCheck** algorithm for multiple finite automata. Due to its dynamic nature, the algorithm is able to locally identify the set of assumptions that must be modified to revalidate the system.

Incremental Changes Between Successive Assumptions. Recall that the L^* algorithm maintains an observation table (S, E, T) corresponding to an assumption A for every component M. During an initial compatibility check, this table stores the information about membership of the current set of traces $(S \cdot E)$ in an unknown language U. Upgrading the component M modifies this unknown language for the corresponding assumption from U to, say, U'. Therefore, checking compatibility after an upgrade requires that the learner must compute a new assumption A' corresponding to U'. As mentioned earlier, in most cases, the languages $L(A)$ and $L(A')$ may differ only slightly; hence, the information about the behaviors of A is reused in computing A'.

Table Revalidation. The original L^* algorithm computes A' starting from an empty table. However, as mentioned before, a more efficient algorithm would try to reuse the previously inferred set of elements of S and E to learn A'. The result in Section 4.4.1 (Theorem 5) precisely enables the L^* algorithm to achieve this goal. In particular, since L^* terminates starting from any *valid* table, the algorithm uses the **Reval** procedure to obtain a valid table by reusing traces in S and experiments in E. The valid table thereby obtained is subsequently made closed, and then learning proceeds in the normal fashion. Doing this allows the compatibility check to restart from any previous set of assumptions by revalidating them. The **RevalidateAssumption** module implements this feature (see

Figure 4.7).

Overall DynamicCheck Procedure.

The **DynamicCheck** procedure instantiates the **NC** rule for n components and enables checking multiple upgrades simultaneously by reusing previous assumptions and verification results. In the description, we denote the previous and new versions of a component finite automaton by M and M' and the previous and new versions of component assemblies by $\mathcal{M}$ and $\mathcal{M}'$, respectively. For ease of description, we always use a property, φ, to denote the right-hand side of the top-level proof obligation of the **NC** rule. We denote the modified property[2] at each recursion level of the algorithm by φ'. The old and new assumptions are denoted by A and A', respectively.

Figure 4.7 presents the pseudo-code of the **DynamicCheck** algorithm to perform the compatibility check. Lines 1-4 describe the case when $\mathcal{M}$ contains only one component. In Line 5-6, if the previous assumption is found to be not valid (using **IsValidAssumption** procedure) with respect to the weakest assumption corresponding to M' and φ', it is revalidated using the **RevalidateAssumption** procedure. Lines 8-10 describe recursive invocation of **DynamicCheck** on $\mathcal{M}' \setminus M'$ against property A'. Finally, Lines 11-16 show how the algorithm detects a counterexample CE and updates A' with it or terminates with a TRUE result or a counterexample. The salient features of this algorithm are the following:

- We assume that there exists a set of previously computed assumptions from the earlier verification check. Suppose we have a component automaton M and a property automaton φ, such that the corresponding weakest assumption is WA. In order to find out if a previously computed assumption (say A) is valid against $L(WA)$ (cf. Definition 10), the **IsValidAssumption** procedure is used. More precisely, the

[2]Under the recursive application of the compatibility-check procedure, the updated property φ' corresponds to an assumption from the previous recursion level.

GenerateAssumption (A, CE)

// Let (S,E,T) be the L^* observation table corresponding to an assumption A;

1: Obtain a *distinguishing suffix* e from CE;
2: $E := E \cup \{e\}$;
3: **forever do**
4: **CloseTable**();
5: $A' :=$ `MkDFA`($\mathcal{T}$);
6: **if** (**IsCandidate**(A')) **return**A';
7: **let** CE' be the counterexample returned by **IsCandidate**;
8: Obtain a *distinguishing suffix* e from CE';
9: $E := E \cup \{e\}$;

Figure 4.6: Pseudo-code for procedure **GenerateAssumption**.

IsValidAssumption procedure checks if the observation table (say $\mathcal{T}$) corresponding to A is valid with respect to $L(WA)$ by asking a membership query for each element of the table (cf. Lemma 7).

- The procedure **GenerateAssumption** (cf. Figure 4.6) essentially models the L^* algorithm. Given a counterexample CE, the procedure **GenerateAssumption** computes the next candidate assumption in a manner similar to the original L^* algorithm (cf. Chapter 2). The termination of the **GenerateAssumption** procedure directly follows from that of the L^* algorithm.
- Verification checks are repeated on a component M' (or a collection of components $\mathcal{M}' \setminus M'$) only if it is (or they are) found to be different from the previous version M ($\mathcal{M} \setminus M$) or if the corresponding property φ has changed (Lines 3, 8). Otherwise, the previously computed and cached result (returned by the procedure **CachedResult**) is reused (Lines 4, 9).

Note that for a component automaton M and a counterexample trace CE, we write $M \parallel CE$ to denote the composition of M with the automaton representation of the trace CE (where the last state is the only accepting state). In order to prove the correctness of **DynamicCheck**, we need the following lemma.

DynamicCheck ($\mathcal{M}'$, φ') **returns** counterexample or TRUE

1: **let** $M' =$ *first* element of $\mathcal{M}'$;
//M and φ denote the first element of $\mathcal{M}$ and the corresponding property before upgrade
//and A denotes the assumption computed previously for M and φ
2: **if** ($\mathcal{M}' = \{M'\}$)
3: **if** ($M \neq M'$ or $\varphi \neq \varphi'$) **return** ($M' \sqsubseteq \varphi'$);
4: **else return CachedResult**($M \sqsubseteq \varphi$);
//check if A is a valid assumption for M' and φ'
5: **if** ($\neg$ **IsValidAssumption**(A, M', φ'))
//make assumption A valid for M' and φ'
6: $A' :=$ **RevalidateAssumption**(A, M', φ');
7: **else** $A' := A$;
//Now check the rest of the system $\mathcal{M}' \setminus M'$ against A'
8: **if** ($A \neq A'$ or $\mathcal{M} \setminus M \neq \mathcal{M}' \setminus M'$)
9: $res :=$ **DynamicCheck**($\mathcal{M}' \setminus M'$, A');
10: **else** $res :=$ **CachedResult**($\mathcal{M} \setminus M \sqsubseteq A$);
11: **while**(res is not TRUE)
//Let CE be the counterexample obtained
12: **if** ($M' \parallel CE \sqsubseteq \varphi'$)
13: $A' :=$ **GenerateAssumption** (A',CE); *// Obtain A' so that $M' \parallel A' \sqsubseteq \varphi'$*
14: $res =$ **DynamicCheck** ($\mathcal{M}' \setminus M'$, A'); *// Check if $\mathcal{M}' \setminus M' \sqsubseteq A'$*
15: **else return** a witness counterexample CE' to $M' \parallel CE \not\sqsubseteq \varphi'$;
16: **return** TRUE;

Figure 4.7: Pseudo-Code for Compatibility Checking on an upgrade. The procedure returns TRUE if $\mathcal{M}' \sqsubseteq \varphi'$ holds, otherwise returns a counterexample witness CE.

Lemma 11 *Suppose $\mathcal{M}$ is a set of component automata (with $M \in \mathcal{M}$) and φ be a specification automaton. Let $\mathcal{M} \setminus M \not\sqsubseteq \varphi$ hold and CE be a witness to it. Moreover, suppose $M \parallel CE \not\sqsubseteq \varphi$ holds, and CE' is a witness to it. Then $\mathcal{M} \not\sqsubseteq \varphi$ holds and CE' is a witness to it.*

Proof. Let $M_2 = \mathcal{M} \setminus M$. Since CE is a witness to $M_2 \not\sqsubseteq \varphi$, we know that $CE \in L(M_2)$. Also, since $M \parallel CE \not\sqsubseteq \varphi$ holds and CE' is a witness to it, there is a $CE'' \in L(M)$ such that $CE' = (CE'' \parallel CE)$ (using the automaton representation of both CE and CE''). Also, $CE' \notin L(\varphi)$. Since $CE'' \in L(M)$ and $CE \in L(M_2)$, it follows that $CE' = (CE'' \parallel CE)$ is in $L(M \parallel M_2) = L(\mathcal{M})$. Hence, CE' is in $L(\mathcal{M})$ but not in $L(\varphi)$. Therefore, CE' is a witness to $\mathcal{M} \not\sqsubseteq \varphi$.

Theorem 6 shows the correctness of **DynamicCheck**. The proof relies on the fact that the rule **NC** for a system of n component automata is complete due to the existence of an unique set of weakest assumptions (cf. Section 4.4.2). Note that we never construct the weakest assumptions directly; they are only used to show that the procedure **DynamicCheck** terminates with the correct result.

Theorem 6 *Given modified $\mathcal{M}'$ and φ', the* **DynamicCheck** *algorithm always terminates with either* TRUE *or a counterexample CE to $\mathcal{M}' \sqsubseteq \varphi'$.*

Proof. We assume that for the earlier system $\mathcal{M}$, a set of previously computed assumption automata $A_1 \ldots A_{n-1}$ exist. Now, suppose one or more components in $\mathcal{M}$ are upgraded resulting in the system $\mathcal{M}'$.

The proof proceeds by induction over the number of components k in $\mathcal{M}'$. In the base case $\mathcal{M}'$ consists of a single component automaton M'; hence we need to model check M' against φ' only if either M or φ changed. This is done in Lines 3-4. Hence, **DynamicCheck** returns the correct result in this case.

Assume for the inductive case that **DynamicCheck**$(\mathcal{M}' \setminus M', A')$ terminates with either *true* or a counterexample CE. If Line 8 holds (i.e., $A' \neq A$ or $\mathcal{M} \setminus M \neq \mathcal{M}' \setminus M'$),

then, by the inductive hypothesis, execution of Line 9 terminates with the correct result: either *true* or a counterexample CE. Otherwise, the previously computed correct result *res* is used (Line 10). Based on this result, Lines 11-16 update the current assumption in an iterative manner. Therefore, it remains to be shown that Lines 11-16 compute the correct return value based on this result.

If the result in Line 9 or Line 10 is *true*, it follows from the soundness of the assume-guarantee rule that $\mathcal{M}' \sqsubseteq \varphi'$ and **DynamicCheck** returns *true* (Line 16). Otherwise, a counterexample CE is found which is a witness to $\mathcal{M} \setminus M \not\sqsubseteq \varphi'$. This counterexample is used in Line 12 to check if $M' \parallel CE \sqsubseteq \varphi'$. If this holds, then CE is used to improve the current assumption in Lines 13-14. Otherwise, the procedure returns a suitable witness CE' (Line 15). In order to show that Lines 11-16 compute the correct result, we need to show that (i) the counterexample CE' is indeed a witness to $\mathcal{M}' \not\sqsubseteq \varphi'$ and, (ii) the loop in Lines 11-15 can execute only a finite number of times.

Using the fact that CE is a witness to $\mathcal{M}' \setminus M' \not\sqsubseteq \varphi'$ (from Line 9-10) and $M' \parallel CE \not\sqsubseteq \varphi'$ (Line 12), it follows from Lemma 11 that $\mathcal{M}' \not\sqsubseteq \varphi'$ and CE' is a suitable witness to $\mathcal{M}' \not\sqsubseteq \varphi'$.

It remains to show that Lines 11-15 can execute only a finite number of times. Note that in Line 13, A' is valid since it was computed by **RevalidateAssumption** (Line 6). Hence, **GenerateAssumption** (Line 13) must terminate (cf. Theorem 5) by learning a new assumption, say A'', such that $M' \parallel A'' \sqsubseteq \varphi'$. Note that by Lemma 2, the number of states of A' or A'' cannot exceed that of the corresponding weakest assumption WA'. Also, it follows from the proof of correctness of L^* (cf. Theorem 1) that $|A'| < |A''|$. Moreover, by the inductive hypothesis, Line 14 must terminate with the correct result. Hence, each iteration of Lines 11-14 of the **while** loop will lead to increase in the number of states of the assumption candidates until $|A''| = |WA'|$. In this case, the loop terminates. If no counterexample is generated at Line 14, then the loop terminates with a true result at Line 16. Otherwise, if a counterexample CE is generated at Line 14 (with $A'' = WA'$), then it follows that $CE \in L(\mathcal{M}' \setminus M')$ and $CE \notin L(WA')$. Therefore it follows from Lemma 7

that $M' \parallel CE \sqsubseteq \varphi'$ does not hold. Hence, by Lemma 11, CE is an actual witness to $\mathcal{M}' \not\sqsubseteq \varphi'$. Therefore, the procedure returns by generating the correct witness CE' at Line 15.

□

4.5 Implementation and Experimental Evaluation

The procedures for checking, in a dynamic manner, the substitutability of components, were implemented in the ComFoRT reasoning framework [85]. The tool includes a front end for parsing and constructing control-flow graphs from C programs. Further, it is capable of model checking properties on programs based on automated may-abstraction (existential abstraction), and it allows compositional verification by employing learning-based, automated assume-guarantee reasoning. We reused the above features of ComFoRT in the implementation of the substitutability check. The tool interface was modified so a collection of components and corresponding upgrades could be specified. We extended the learning-based, automated assume-guarantee to obtain its dynamic version, as required in the compatibility check. Doing this involved keeping multiple learner instances across calls to the verification engine and implementing algorithms to validate multiple, previous observation tables in an efficient way during learning. We also implemented the under-approximation generation algorithms for performing the containment check on small program examples. Doing this involved procedures for implementing must-abstractions from C code using predicates obtained from C components [75].

We performed the compatibility check while verifying upgrades of a benchmark provided to us by our industrial partner, ABB Inc. [5]. The benchmarks consist of seven components which together implement an inter-process communication (IPC) protocol. The combined state space is over 10^6.

We used a set of properties describing the functionality of the verified portion of the

IPC protocol. We used upgrades of the *write-queue* (ipc_1) and the *ipc-queue* (ipc_2 and ipc_3) components. The upgrades had both missing and extra behaviors compared to their original versions. We verified two properties (P_1 and P_2) before and after the upgrades. We also verified the properties on a simultaneous upgrade (ipc_4) of both the components. P_1 specifies that a process may write data into the *ipc-queue* only after it obtains a lock for the corresponding critical section. P_2 specifies an order in which data may be written into the *ipc-queue*. Figure 4.8 shows the comparison between the time required for initial verification of the IPC system, and the time taken by **DynamicCheck** for verifying the upgrades. In Figure 4.8, $\#Mem.\ Queries$ denotes the total number of membership queries made during verification of the original assembly, T_{orig} denotes the time required for the verification of the original assembly, and T_{ug} denotes the time required for the verification of the upgraded assembly.

Upgrade # (Prop.)	**# Mem. Queries**	**T$_{orig}$ (msec)**	**T$_{ug}$ (msec)**
$ipc_1(P_1)$	279	2260	13
$ipc_1(P_2)$	308	1694	14
$ipc_2(P_1)$	358	3286	17
$ipc_2(P_2)$	232	805	10
$ipc_3(P_1)$	363	3624	17
$ipc_3(P_2)$	258	1649	14
$ipc_4(P_1)$	355	1102	24

Figure 4.8: Summary of Results for **DynamicCheck**

We observed that the previously generated assumptions in all the cases were also sufficient to prove the properties on the upgraded system. Hence, the compatibility check succeeded in a small fraction of time (T_{ug}) as compared to the time for compositional verification (T_{orig}) of the original system.

4.6 Related Work

A number of approaches to check substitutability (also referred to as compatibility check) have been proposed previously. e.g,. based on behavioral types [92] and automated invariant generation for components [96]. Most of the previous approaches are based on a notion of refinement: all behaviors of the replaced component must be present in original one, in which case, the new component is said to refine the old component. In contrast, our substitutability check is more general and allows both loss and addition of behaviors with evolution.

For instance, de Alfaro et al. [34, 49] define a notion of interface automaton for modeling component interfaces and show compatibility between components via refinement and consistency between interfaces. However, automated techniques for constructing interface automata from component implementations are not presented. In contrast, our approach automatically extracts conservative finite state automaton models from component implementations. Moreover, we do not require refinement among the old components and their new versions.

McCamant and Ernst [96] suggest a technique for checking compatibility of multi-component upgrades. They derive consistency criteria by focusing on input/output component behavior only and abstract away the temporal information. Even though they state that their abstractions are unsound in general, they report success in detecting important errors. In contrast, our abstractions preserve temporal information about component behavior and are always sound. They also use a refinement-based notion on the generated consistency criteria for showing compatibility.

Another approach to preserve behavioral properties of a component across an upgrade is based on the principle of behavioral sub-typing [92]: type T' is a subtype of type T if for every property $\phi(t)$ provable about objects t of type T, $\phi(t')$ is provable about objects t' of type T'. The notion of subtypes is extended to system behaviors by augmenting object

types with invariants and constraints and showing that these constraints are maintained for objects of the subtype. However, this approach focuses only on the given behavior specification of a single component and does not take into account the way it is used in the component assembly. In contrast, the assumptions in our approach reflect the behavior of environment components. Therefore, although the upgraded component may not satisfy a property ϕ in all possible environments, it may continue to satisfy ϕ in context of the current environment components. In other words, the new component may not be a behavioral subtype of the earlier one, but still be compatible with its environment.

4.7 Conclusions

We proposed a solution to the critical and vital problem of component substitutability consisting of two phases: *containment* and *compatibility*. The compatibility check performs compositional reasoning with help of a *dynamic* regular language inference algorithm and a model checker. Our experiments confirm that the dynamic approach is more effective than complete re-validation of the system after an upgrade. The containment check detects behaviors which were present in each component before but not after the upgrade. These behaviors are used to construct useful feedback to the developers.

Chapter 5

Checking Simulation Conformance Compositionally

This chapter extends the automated AGR paradigm to branching time setting. Recall that the initial approach was proposed for a linear time setting (cf. Chapter 3). We describe an algorithm for learning tree languages into a tree automaton called L^T and then show how it can be used to checking simulation conformance in an automated AGR framework.

5.1 Preliminaries

In this section, we introduce basic notation and definitions that will be used in the rest of this chapter.

Definition 14 (Labeled Transition System) *A labeled transition system (LTS) is a 4-tuple $(S, Init, \Sigma, T)$ where (i) S is a finite set of states, (ii) $Init \subseteq S$ is the set of initial states, (iii) Σ is a finite alphabet, and (iv) $T \subseteq S \times \Sigma \times S$ is the transition relation. We write $s \xrightarrow{\alpha} s'$ as a shorthand for $(s, \alpha, s') \in T$.*

Definition 15 (Simulation) *Let $M_1 = (S_1, Init_1, \Sigma_1, T_1)$ and $M_2 = (S_2, Init_2, \Sigma_2, T_2)$ be LTSs such that $\Sigma_1 = \Sigma_2 = \Sigma$ say. A relation $\mathcal{R} \subseteq S_1 \times S_2$ is said to be a simulation*

relation if:

$$\forall s_1, s_1' \in S_1 \,.\, \forall a \in \Sigma \,.\, \forall s_2 \in S_2 \,.\, s_1 \mathcal{R} s_2 \wedge s_1 \xrightarrow{a} s_1' \Rightarrow \exists s_2' \in S_2 \,.\, s_2 \xrightarrow{a} s_2' \wedge s_1' \mathcal{R} s_2'$$

We say M_1 is simulated by M_2, and denote this by $M_1 \preccurlyeq M_2$, if there is a simulation relation $\mathcal{R}$ such that $\forall s_1 \in I_1 \,.\, \exists s_2 \in I_2 \,.\, s_1 \mathcal{R} s_2$. We say M_1 and M_2 are simulation equivalent if $M_1 \preccurlyeq M_2$ and $M_2 \preccurlyeq M_1$.

Definition 16 (Tree) *Let ϵ denote the empty tree and Σ be an alphabet. The set of trees over Σ is defined by the grammar: $T := \epsilon \mid \Sigma \cdot T \mid T + T$. The set of all trees over the alphabet Σ is denote by Σ^T, and we let t range over it.*

Definition 17 (Context) *The set of contexts over an alphabet Σ can be defined by the grammar: $C := \Box \mid \Sigma \cdot C \mid C + T \mid T + C$. We let c range over the set of contexts.*

A context is like a tree except that it has exactly one hole denoted by $\Box$ at one of its nodes. When we plug in a tree t in a context c, we essentially replace the single $\Box$ in c by t. The resulting tree is denoted by $c[t]$. A tree t can naturally be seen as an LTS. Specifically, the states of the LTS are the nodes of t, the only initial state is the root node of t, and there is a labeled transition from node t_1 to t_2 labeled with α if $t_1 = \alpha \cdot t_2$ or $t_1 = \alpha \cdot t_2 + t_3$ or $t_1 = t_2 + \alpha \cdot t_3$.

Definition 18 (Tree Language of an LTS) *An LTS M induces a tree language, which is denoted by $\mathcal{T}(M)$ and is defined as: $\mathcal{T}(M) = \{t \mid t \preccurlyeq M\}$. In other words, the tree language of an LTS contains all the trees that can be simulated by the LTS.*

For example, the language of M (Figure 5.1(a)) contains the trees ϵ, $\alpha \cdot \lambda$, $\alpha \cdot (\lambda + \lambda)$, $\alpha \cdot \lambda + \beta \cdot \lambda$, $\beta \cdot \lambda + \beta \cdot \lambda$ and so on. The notion of tree languages of LTSs and simulation between LTSs are fundamentally connected. Specifically, it follows from the definition of simulation between LTSs that for any two LTSs M_1 and M_2, the following holds:

$$M_1 \preccurlyeq M_2 \iff \mathcal{T}(M_1) \subseteq \mathcal{T}(M_2) \tag{5.1}$$

Definition 19 (Tree Automaton) *A (bottom-up) tree automaton (TA) is a 6-tuple* $A = (S, Init, \Sigma, \delta, \otimes, F)$ *where: (i)* S *is a set of states, (ii)* $Init \subseteq S$ *is a set of initial states, (iii)* Σ *is an alphabet, (iv)* $\delta \subseteq S \times \Sigma \times S$ *is a forward transition relation, (v)* $\otimes \subseteq S \times S \times S$ *is a cross transition relation, and (vi)* $F \subseteq S$ *is a set of accepting states*[1].

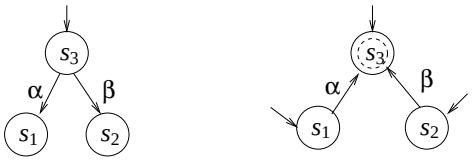
Tree automata accept trees and can be viewed as two-dimensional extensions of finite automata. Since trees can be extended either forward (via the $\cdot$ operator) and across (via the $+$ operator), a TA must have transitions defined when either of these two kinds of extensions of its input tree are encountered. This is achieved via the forward and cross transitions respectively. The automaton starts at each leaf of the input tree at some initial state, and then runs bottom-up in accordance with its forward and cross transition relations. The forward transition is applied when a tree of the form $\alpha \cdot T$ is encountered. The cross transition is applied when a tree of the form $T_1 + T_2$ is found. The tree is accepted if the run ends at the root of the tree in some accepting state of A.

Before we formally define the notions of runs and acceptance, we introduce a few notational conventions. We may sometimes write $s \xrightarrow{\alpha} s'$ or $s' \in \delta(s, \alpha)$ as a shorthand for $(s, \alpha, s') \in \delta$, and $s_1 \otimes s_2 \longrightarrow s$ as a shorthand for $(s_1, s_2, s) \in \otimes$. Similarly, for sets of states S_1, S_2, we use the following shorthand notations:

$$\begin{aligned} \delta(S_1, \alpha) &= \{s' \mid \exists s \in S_1 \centerdot s \xrightarrow{\alpha} s'\} \\ S_1 \otimes S_2 &= \{s \mid \exists s_1 \in S_1 \centerdot \exists s_2 \in S_2 \centerdot (s_1, s_2, s) \in \otimes\} \end{aligned}$$

Definition 20 (Run/Acceptance) *Let* $A = (S, Init, \Sigma, \delta, \otimes, F)$ *be a TA. The run of* A *is a function* $r : \Sigma^T \to 2^S$ *from trees to sets of states of* A *that satisfies the following conditions: (i)* $r(\epsilon) = Init$*, (ii)* $r(\alpha \cdot t) = \delta(r(t), \alpha)$*, and (iii)* $r(t_1 + t_2) = r(t_1) \otimes r(t_2)$*. A*

[1]We use the above specialized definition of TA instead of the conventional one [47] since we are only interested in the term signature given by the grammar in Definition 16. Specifically, the forward transitions correspond to the unary function symbol $a\cdot$ (one for each $a \in \Sigma$), the cross transitions correspond to the binary function symbol $+$, and the initial states correspond to the constant symbol λ.



$\otimes$	s_1	s_2	s_3
s_1	s_1		
s_2		s_2	
s_3			s_3

(a) (b)

Figure 5.1: (a-left) an LTS M with initial state s_3; (a-right) forward transitions of a tree automaton A accepting $\mathcal{T}(M)$; all states are initial; (b) table showing cross transition relation $\otimes$ of A. Note that some table entries are absent since the relation $\otimes$ is not total.

tree T is accepted by A iff $r(T) \cap F \neq \emptyset$. The set of trees accepted by A is known as the language of A and is denoted by $\mathcal{L}(A)$.

[Runs/acceptance of tree automata] Consider the trees $t_1 = \epsilon$, $t_2 = \alpha \cdot \epsilon$, $t_3 = \beta \cdot \epsilon$, $t_4 = t_2 + t_3$ and $t_5 = \alpha \cdot t_2$. Let r be the run function of the TA A in Figure 5.1. Then we have the following:

- $r(t_1) = r(\epsilon) = \{s_1, s_2, s_3\}$
- $r(t_2) = r(\alpha \cdot \epsilon) = \delta(r(\epsilon), \alpha) = \delta(\{s_1, s_2, s_3\}, \alpha) = \{s_3\}$
- $r(t_3) = r(\beta \cdot \epsilon) = \delta(r(\epsilon), \beta) = \delta(\{s_1, s_2, s_3\}, \beta) = \{s_3\}$
- $r(t_4) = r(t_2 + t_3) = r(t_2) \otimes r(t_3) = \{s_3\} \otimes \{s_3\} = \{s_3\}$
- $r(t_5) = r(\alpha \cdot t_2) = \delta(r(t_2), \alpha) = \delta(\{s_3\}, \alpha) = \emptyset$

Since s_3 is the only accepting state of A, we find that A accepts t_1, t_2, t_3 and t_4 but does not accept t_5. The following lemma will be useful later on.

A *deterministic* tree automaton (DTA) is one which has a single initial state and where the forward and cross transition relations are *functions* $\delta : S \times \Sigma \to S$ and $\otimes : S \times S \to S$ respectively. If $A = (S, Init, \Sigma, \delta, \otimes, F)$ is a DTA then $Init$ refers to the single initial state, and $\delta(s, \alpha)$ and $s_1 \otimes s_2$ refer to the unique state s' such that $s \xrightarrow{\alpha} s'$ and $s_1 \otimes s_2 \longrightarrow s'$ respectively. Note that if A is deterministic then for every tree t the set $r(t)$ is a singleton, i.e., the run of A on any tree t ends at a unique state of A. Further, we recall [47] the

following facts about tree-automata. The set of languages recognized by TA (referred to as *regular tree languages* henceforth) is closed under union, intersection and complementation. For every TA A there is a DTA A' such that $\mathcal{L}(A) = \mathcal{L}(A')$. Given any regular tree language L there is always a *unique* (up to isomorphism) *smallest* DTA A such that $\mathcal{L}(A) = L$.

The following lemma asserts that for any LTS M, the set $\mathcal{T}(M)$ is a regular tree language. Thus, using (5.1), the simulation problem between LTSs can also be viewed as the language containment problem between tree automata.

Lemma 12 *For any LTS M there is a TA A such that $\mathcal{L}(A) = \mathcal{T}(M)$.*

Proof. Let $M = (S, Init, \Sigma, T)$. Construct a TA $A = (S', Init', \Sigma, \delta', \otimes', F')$ such that

- $S' = Init' = S$
- $F' = Init$
- $(s, \alpha, s') \in \delta'$ iff $(s', \alpha, s) \in T$,
- $\forall s' \in S' \centerdot (s', s', s') \in \otimes'$.

Let r denote the run of A on trees and t be any tree. We will prove the claim that $s \in r(t)$ iff t can be simulated by M starting from state s. Then, $t \in \mathcal{L}(A)$ iff $r(t) \cap Init \neq \emptyset$ iff M simulates t starting from one of its initial states, i.e., $t \in \mathcal{T}(M)$. Therefore we will have $\mathcal{L}(A) = \mathcal{T}(M)$ as required.

The proof of the claim is by structural induction on t. We will use the shorthand notation $t \preccurlyeq M[s]$ to denote that fact that t can be simulated by M starting from state s. For the base case $t = \lambda$, we have $r(\lambda) = Init' = S$, and $\lambda \preccurlyeq M[s]$ for all $s \in S$, and therefore the claim holds. For the induction step there are two cases:

- $t = \alpha \cdot t'$: Then

$$\begin{aligned} t \preccurlyeq M[s] &\Leftrightarrow \exists s' \centerdot s \xrightarrow{\alpha} s' \in T \text{ and } t' \preccurlyeq M[s'] \\ &\Leftrightarrow s' \in r(t') && \text{(by induction hypothesis)} \\ &\Leftrightarrow s \in r(\alpha \cdot t') && \text{(since } s' \xrightarrow{\alpha} s \in \delta) \end{aligned}$$

- $t = t_1 + t_2$: Then

$$\begin{aligned} t \preccurlyeq M[s] &\Leftrightarrow t_1 \preccurlyeq M[s] \text{ and } t_2 \preccurlyeq M[s] \\ &\Leftrightarrow s \in r(t_1) \text{ and } s \in r(t_2) && \text{(by induction hypothesis)} \\ &\Leftrightarrow s \in r(t_1 + t_2) && \text{(since } (s,s,s) \in \otimes) \end{aligned}$$

Consider the LTS M and TA A as shown in Figure 5.1. A is obtained from M by the construction in lemma 12 and hence $\mathcal{L}(A) = \mathcal{T}(M)$.

We now provide the standard notion of parallel composition between LTSs, where components synchronize on shared actions and proceed asynchronously on local actions.

Definition 21 (Parallel Composition of LTSs) *Given LTSs $M_1 = (S_1, Init_1, \Sigma_1, T_1)$ and $M_2 = (S_2, Init_2, \Sigma_2, T_2)$, their parallel composition $M_1 \parallel M_2$ is an LTS $M = (S, Init, \Sigma, T)$ where $S = S_1 \times S_2$, $Init = Init_1 \times Init_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$, and the transition relation T is defined as follows: $((s_1, s_2), \alpha, (s_1', s_2')) \in T$ iff for $i \in \{1, 2\}$ the following holds:*

$$(\alpha \in \Sigma_i) \wedge (s_i, \alpha, s_i') \in T_i \quad \bigvee \quad (\alpha \notin \Sigma_i) \wedge (s_i = s_i')$$

Working with different alphabets for each component would needlessly complicate the exposition in Section 5.3. For this reason, without loss of generality, we make the simplifying assumption that $\Sigma_1 = \Sigma_2$. This is justified because we can construct LTSs M_1' and M_2', each with the same alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$ such that $M_1' \parallel M_2'$ is simulation equivalent (in fact bisimilar) to $M_1 \parallel M_2$. Specifically, $M_1' = (S_1, Init_1, \Sigma, T_1')$ and $M_2' = (S_2, Init_2, \Sigma, T_2')$ where

$$\begin{aligned} T_1' &= T_1 \cup \{(s, \alpha, s) \mid s \in S_1 \text{ and } \alpha \in \Sigma_2 \setminus \Sigma_1\} \\ T_2' &= T_2 \cup \{(s, \alpha, s) \mid s \in S_2 \text{ and } \alpha \in \Sigma_1 \setminus \Sigma_2\} \end{aligned}$$

Finally, the reader can check that if M_1 and M_2 are LTSs with the same alphabet then $\mathcal{T}(M_1 \parallel M_2) = \mathcal{T}(M_1) \cap \mathcal{T}(M_2)$.

Lemma 13 *Let A be any TA and r be its run on trees. Let t_1 and t_2 be two trees such that $r(t_1) = r(t_2)$. Then for any context c, $r(c[t_1]) = r(c[t_2])$.*

Proof. The proof is by structural induction on the context c. For the base case where $c = \Box$, we have $r(c[t_1]) = r(t_1) = r(t_2) = r(c[t_2])$. For the induction step, there are two possible cases.

- $c = \alpha \cdot c'$: Then

$$\begin{aligned} r(c[t_1]) &= r(\alpha \cdot c'[t_1]) \\ &= \delta(r(c'[t_1]), \alpha) \\ &= \delta(r(c'[t_2]), \alpha) \qquad \text{(by induction hypothesis)} \\ &= r(\alpha \cdot c'[t_2]) \\ &= r(c[t_2]) \end{aligned}$$

- $c = c' + t$: Then

$$\begin{aligned} r(c[t_1]) &= r(c'[t_1] + t) \\ &= r(c'[t_1]) \otimes r[t] \\ &= r(c'[t_2]) \otimes r[t] \qquad \text{(by induction hypothesis)} \\ &= r(c'[t_2] + t) \\ &= r(c[t_2]) \end{aligned}$$

- $c = t + c'$: This case is similar to the one above.

5.2 Learning Minimal DTA

We now present the algorithm L^T that learns the minimal DTA for an unknown regular language U. It is assumed that the alphabet Σ of U is fixed, and that the language U is presented by a Teacher that answers two kinds of queries:

1. *Membership.* Given a tree t, is t an element of U, i.e., $t \in U$?
2. *Candidate.* Given a DTA A does A accept U, i.e., $\mathcal{L}(A) = U$? If $\mathcal{L}(A) = U$ the teacher returns TRUE, else it returns FALSE along with a counterexample tree CE that is in the symmetric difference of $\mathcal{L}(A)$ and U.

We will use the following notation. Given any sets of trees S_1, S_2 and an alphabet Σ we denote by $\Sigma \cdot S_1$ the set of trees $\Sigma \cdot S_1 = \{\alpha \cdot t \mid \alpha \in \Sigma \wedge t \in S_1\}$, and by $S_1 + S_2$ the set $S_1 + S_2 = \{t_1 + t_2 \mid t_1 \in S_1 \wedge t_2 \in S_2\}$, and by $\widehat{\mathcal{S}}$ the set $\mathcal{S} \cup (\Sigma \cdot \mathcal{S}) \cup (\mathcal{S} + \mathcal{S})$.

Observation Table : The algorithm L^T maintains an observation table $\tau = (\mathcal{S}, \mathcal{E}, \mathcal{R})$ where (i) $\mathcal{S}$ is a set of trees such that $\epsilon \in \mathcal{S}$, (ii) $\mathcal{E}$ is a set of contexts such that $\Box \in \mathcal{E}$, and (iii) $\mathcal{R}$ is a function from $\widehat{\mathcal{S}} \times \mathcal{E}$ to $\{0, 1\}$ that is defined as follows: $\mathcal{R}(t, c) = 1$ if $c[t] \in U$ and 0 otherwise. Note that given $\mathcal{S}$ and $\mathcal{E}$ we can compute $\mathcal{R}$ using membership queries. The information in the table is eventually used to construct a candidate DTA A_τ. Intuitively, the elements of $\mathcal{S}$ will serve as states of A_τ, and the contexts in $\mathcal{E}$ will play the role of *experiments* that distinguish the states in $\mathcal{S}$. Henceforth, the term experiment will essentially mean a context. The function $\mathcal{R}$ and the elements in $\widehat{\mathcal{S}} \setminus \mathcal{S}$ will be used to construct the forward and cross transitions between the states.

For any tree $t \in \widehat{\mathcal{S}}$, we denote by $row(t)$ the function from the set of experiments $\mathcal{E}$ to $\{0, 1\}$ defined as: $\forall c \in \mathcal{E} \,.\, row(t)(c) = \mathcal{R}(t, c)$.

Definition 22 (Well-formed) *An observation table $(\mathcal{S}, \mathcal{E}, \mathcal{R})$ is said to be well-formed if: $\forall t, t' \in \mathcal{S} \,.\, t \neq t' \implies row(t) \neq row(t')$. From the definition of $row(t)$ above, this boils down to: $\forall t, t' \in \mathcal{S} \,.\, t \neq t' \implies \exists c \in \mathcal{E} \,.\, \mathcal{R}(t, c) \neq \mathcal{R}(t', c)$.*

In other words, any two different row entries of a well-formed observation table must be distinguishable by at least one experiment in $\mathcal{E}$. The following crucial lemma imposes an upper-bound on the size of any well-formed observation table corresponding to a given regular tree language U.

Lemma 14 *Let $(\mathcal{S}, \mathcal{E}, \mathcal{R})$ be any well-formed observation table for a regular tree language U. Then $|\mathcal{S}| \leq n$, where n is the number of states of the smallest DTA which accepts U. In other words, the number of rows in any well-formed observation table for U cannot exceed the number of states in the smallest DTA that accepts U.*

	$\square$
ϵ	$1 \quad (s_0)$
$\alpha \cdot \epsilon$	1
$\beta \cdot \epsilon$	1
$\epsilon + \epsilon$	1

(a)

δ	α	β
s_0	s_0	s_0

(b)

$\otimes$	s_0
s_0	s_0

(c)

Figure 5.2: (a) A well-formed and closed observation table τ; (b) forward transition relation of the candidate A^1_τ constructed from τ; (c) cross transition relation of A^1_τ.

Proof. The proof is by contradiction. Let A be the smallest DTA accepting U and let $(\mathcal{S}, \mathcal{E}, \mathcal{R})$ be a well-formed observation table such that $|\mathcal{S}| > n$. Then there are two distinct trees t_1 and t_2 in $\mathcal{S}$ such that the runs of A on both t_1 and t_2 end on the same state of A. Then for any context c, the runs of A on $c[t_1]$ and $c[t_2]$ both end on the same state. But on the other hand, since the observation table is well-formed, there exists an experiment $c \in \mathcal{E}$ such that $\mathcal{R}(t_1, c) \neq \mathcal{R}(t_2, c)$, which implies that the runs of A on $c[t_1]$ and $c[t_2]$ end on different states of A. Contradiction.

Definition 23 (Closed) *An observation table $(\mathcal{S}, \mathcal{E}, \mathcal{R})$ is said to be closed if*

$$\forall t \in \widehat{\mathcal{S}} \setminus \mathcal{S} \,.\, \exists t' \in \mathcal{S} \,.\, row(t') = row(t)$$

Note that, given any well-formed observation table $(\mathcal{S}, \mathcal{E}, \mathcal{R})$, one can always construct a well-formed and closed observation table $(\mathcal{S}', \mathcal{E}, \mathcal{R}')$ such that $\mathcal{S} \subseteq \mathcal{S}'$. Specifically, we repeatedly try to find an element t in $\widehat{\mathcal{S}} \setminus \mathcal{S}$ such that $\forall t' \in \mathcal{S} \,.\, row(t') \neq row(t)$. If no such t can be found then the table is already closed and we stop. Otherwise, we add t to $\mathcal{S}$ and repeat the process. Note that, the table always stays well-formed. Then by Lemma 14, the size of $\mathcal{S}$ cannot exceed the number of states of the smallest DTA that accepts U. Hence this process always terminates.

Figure 5.2a shows a well-formed and closed table with $\mathcal{S} = \{\epsilon\}$, $\mathcal{E} = \{\square\}$, $\Sigma = \{\alpha, \beta\}$, and for the regular tree language defined by the TA in Figure 5.1. Note that $row(t) =$

$row(\lambda)$ for every $t \in \{\alpha \cdot \lambda, \beta \cdot \lambda, \lambda + \lambda\}$, and hence the table is closed.

Conjecture Construction: From a well-formed and closed observation table $\tau = (\mathcal{S}, \mathcal{E}, \mathcal{R})$, the learner constructs a candidate DTA $A_\tau = (S, Init, \Sigma, \delta, \otimes, F)$ where (i) $S = \mathcal{S}$, (ii) $Init = \epsilon$, (iii) $F = \{t \in \mathcal{S} \mid \mathcal{R}(t, \square) = 1\}$, (iv) $\delta(t, \alpha) := t'$ such that $row(t') = row(\alpha \cdot t)$, and (v) $t_1 \otimes t_2 := t'$ such that $row(t') = row(t_1 + t_2)$. Note that in (iv) and (v) above there is guaranteed to be a unique such t' since τ is closed and well-formed, hence A_τ is well-defined.

Consider again the closed table in Figure 5.2a. The learner extracts a conjecture A_τ from it with a single state s_0, which is both initial and final. Figures 5.2b and 5.2c show the forward and cross transitions of A_τ.

The Learning Algorithm: The algorithm L^T is iterative and always maintains a well-formed observation table $\tau = (\mathcal{S}, \mathcal{E}, \mathcal{R})$. Initially, $\mathcal{S} = \{\epsilon\}$ and $\mathcal{E} = \{\square\}$. In each iteration, L^T proceeds as follows:

1. Make τ closed as described previously.
2. Construct a conjecture DTA A_τ from τ, and make a candidate query with A_τ. If A_τ is a correct conjecture, then L^T terminates with A_τ as the answer. Otherwise, let CE be the counterexample returned by the teacher.
3. Extract a context c from CE, add it to $\mathcal{E}$, and proceed with the next iteration from step 1. The newly added c is such that when we make τ closed in the next iteration, the size of $\mathcal{S}$ is guaranteed to increase.

Extracting an Experiment From CE: Let r be the run function of the failed candidate A_τ. For any tree t, let $\tau(t) = r(t)$, i.e., $\tau(t)$ is the state at which the run of A_τ on t ends. Note that since states of A_τ are elements in $\mathcal{S}$, $\tau(t)$ is itself a tree. The unknown language U induces a natural equivalence relation $\approx$ on the set of trees as follows: $t_1 \approx t_2$ iff $t_1 \in U \iff t_2 \in U$.

```
1   procedure ExpGen(c,t)
2     case
3       t = α · t'  :  if (c[τ(t)] ≉ c[α · τ(t)]) then
4                        add c to E
5                      else ExpGen(c[α · □], t');

6       t = t1 + t2:  if (c[τ(t)] ≉ c[τ(t1) + τ(t2)]) then
7                        add c to E
8                      else if (c[τ(t)] ≈ c[τ(t1) + t2]) then
9                        ExpGen(c[τ(t1) + □], t2);
10                     else ExpGen(c[□ + t2], t1)
11    end case
12  end procedure
```

Figure 5.3: Pseudocode for extracting an experiment from a counterexample.

The procedure **ExpGen** for extracting a new experiment from the counterexample is iterative (see Figure 5.3). It maintains a context c and a tree t that satisfy the following condition: **(INV)** $c[t] \not\approx c[\tau(t)]$. Initially $c = \Box$ and $t = CE$. Note that this satisfies **INV** because $CE \in U \iff CE \notin \mathcal{L}(A_\tau)$. In each iteration, **ExpGen** either generates an appropriate experiment or updates c and t such that **INV** is maintained and the size of t strictly decreases. Note that t cannot become ϵ since at that point **INV** can no longer be maintained; this is because if $t = \epsilon$ then $\tau(t) = \epsilon$ and therefore $c[t] \approx c[\tau(t)]$, which would contradict **INV**. Hence, **ExpGen** must terminate at some stage by generating an appropriate experiment. Now, there are two possible cases:

Case 1: $(t = \alpha \cdot t')$ Let $c' = c[\alpha \cdot \Box]$. We consider two sub-cases. Suppose that $c[\tau(t)] \approx c'[\tau(t')]$. From **INV** we know that $c[t] \not\approx c[\tau(t)]$. Hence $c'[\tau(t')] \not\approx c[t] \approx c'[t']$. Hence, **ExpGen** proceeds to the next iteration with $c = c'$ and $t = t'$. Note that **INV** is preserved and the size of t strictly decreases.

Otherwise, suppose that $c[\tau(t)] \not\approx c'[\tau(t')]$. In this case, **ExpGen** terminates by adding the experiment c to $\mathcal{E}$. Note that A_τ has the transition $\tau(t') \xrightarrow{\alpha} \tau(t)$, i.e., $row(\tau(t)) = row(\alpha \cdot \tau(t'))$. But now, since $c[\tau(t)] \not\approx c'[\tau(t')] \approx c[\alpha \cdot \tau(t')]$, the experiment c is guaranteed

to distinguish between $\tau(t)$ and $\alpha \cdot \tau(t')$. Therefore, the size of $\mathcal{S}$ is guaranteed to increase when we attempt to close τ in the next iteration.

Case 2: $(t = t_1 + t_2)$ There are two sub-cases. Suppose that $c[\tau(t)] \not\approx c[\tau(t_1) + \tau(t_2)]$. In this case, **ExpGen** terminates by adding the experiment c to $\mathcal{E}$. The experiment c is guaranteed to distinguish between $\tau(t)$ and $\tau(t_1) + \tau(t_2)$ and therefore strictly increase the size of $\mathcal{S}$ when we attempt to close τ in the next iteration.

Otherwise, suppose that $c[\tau(t)] \approx c[\tau(t_1) + \tau(t_2)]$. We again consider two sub-cases. Suppose that $c[\tau(t_1) + \tau(t_2)] \not\approx c[\tau(t_1) + t_2]$. In this case, **ExpGen** proceeds to the next iteration with $c = c[\tau(t_1) + \Box]$ and $t = t_2$. Note that **INV** is preserved and the size of t strictly decreases.

Otherwise, we have $c[\tau(t_1) + t_2] \approx c[\tau(t_1) + \tau(t_2)] \approx c[\tau(t)]$, and by **INV** we know that $c[\tau(t)] \not\approx c[t] \approx c[t_1 + t_2]$. Hence, it must be the case that $c[\tau(t_1) + t_2] \not\approx c[t_1 + t_2]$. In this case, **ExpGen** proceeds to the next iteration with $c = c[\Box + t_2]$ and $t = t_1$. Note that, once again **INV** is preserved and the size of t strictly decreases. This completes the argument for all cases.

We show how L^T learns the minimal DTA corresponding to the language U of TA A of Figure 5.1. L^T starts with an observation table τ with $\mathcal{S} = \{\epsilon\}$ and $\mathcal{E} = \{\Box\}$. The table is then made closed by asking membership queries, first for ϵ and then for its (forward and cross) extensions $\{\alpha \cdot \epsilon, \beta \cdot \epsilon, \epsilon + \epsilon\}$. The resulting closed table τ_1 is shown in Figure 5.2a. L^T then extracts a candidate A^1_τ from τ_1, which is shown in Figure 5.2b.

When the conjecture A^1_τ is presented to the teacher, it checks if $\mathcal{L}(A^1_\tau) = U$. In our case, it detects otherwise and returns a counterexample CE from the symmetric difference of $\mathcal{L}(A^1_\tau)$ and U. For the purpose of illustration, let us assume CE to be $\alpha \cdot \beta \cdot \epsilon$. Note that $CE \in \mathcal{L}(A^1_\tau) \setminus U$. The algorithm **ExpGen** extracts the context $\alpha \cdot \Box$ from CE and adds it to the set of experiments $\mathcal{E}$. L^T now asks membership queries corresponding to the new experiment and checks if the new table τ is closed. It finds that $row(\alpha \cdot \epsilon) \neq row(t)$

	$\square$	$\alpha \cdot \square$	
ϵ	1	1	(s_0)
$\alpha \cdot \epsilon$	1	0	(s_1)
$\alpha \cdot \alpha \cdot \epsilon$	0	0	(s_2)
$\beta \cdot \epsilon$	1	0	
$\beta \cdot \alpha \cdot \epsilon$	0	0	
$\alpha \cdot \alpha \cdot \alpha \cdot \epsilon$	0	0	
$\beta \cdot \alpha \cdot \alpha \cdot \epsilon$	0	0	
$\epsilon + \epsilon$	1	1	
$\epsilon + \alpha \cdot \epsilon$	1	0	
$\alpha \cdot \epsilon + \alpha \cdot \epsilon$	1	0	
$\epsilon + \alpha \cdot \alpha \cdot \epsilon$	0	0	
$\alpha \cdot \epsilon + \alpha \cdot \alpha \cdot \epsilon$	0	0	
$\alpha \cdot \alpha \cdot \epsilon + \alpha \cdot \alpha \cdot \epsilon$	0	0	

(a)

δ	α	β
s_0	s_1	s_1
s_1	s_2	s_2
s_2	s_2	s_2

(b)

$\otimes$	s_0	s_1	s_2
s_0	s_0	s_1	s_2
s_1	s_1	s_1	s_2
s_2	s_2	s_2	s_2

(c)

Figure 5.4: (a) observation table τ and (b) transitions for the second conjecture A^2_τ.

for all $t \in \mathcal{S}$, and hence it moves $\alpha \cdot \lambda$ from $\widehat{\mathcal{S}} \setminus \mathcal{S}$ to $\mathcal{S}$ in order to make τ closed. Again, membership queries for all possible forward and cross extensions of $\alpha \cdot \epsilon$ are asked. This process is repeated till τ becomes closed. Figure 5.4a shows the final closed τ. As an optimization, we omit rows for the trees $t_1 + t_2$ whenever there is already a row for $t_2 + t_1$; we know that the rows for both these trees will have the same markings. The corresponding conjecture A^2_τ contains three states s_0, s_1 and s_2 and its forward and cross transitions are shown in Figure 5.4b and Figure 5.4c. s_0 is the initial state and both s_0 and s_1 are final states. The candidate query with A^2_τ returns TRUE since $\mathcal{L}(A^2_\tau) = U$, and L^T terminates with A^2_τ as the output.

Correctness and Complexity:

Theorem 7 *Algorithm L^T terminates and outputs the minimal DTA that accepts the unknown regular language U.*

Proof. Termination is guaranteed by the facts that each iteration of L^T terminates, and in each iteration $|\mathcal{S}|$ must strictly increase, and, by Lemma 14, $|\mathcal{S}|$ cannot exceed the number

of states of the smallest DTA that accepts U. Further, since L^T terminates only after a correct conjecture, if the DTA A_τ is its output then $\mathcal{L}(A_\tau) = U$. Finally, since the number of states in A_τ equals $|S|$, by Lemma 14 it also follows that A_τ is the minimal DTA for U.

To keep the space consumption of L^T within polynomial bounds, the trees and contexts in $\widehat{\mathcal{S}}$ and $\mathcal{E}$ are kept in a DAG form, where common subtrees between different elements in $\widehat{\mathcal{S}}$ and $\mathcal{E}$ are shared. Without this optimization, the space consumption can be exponential in the worst case. The other point to note is that the time taken by L^T depends on the counterexamples returned by the teacher; this is because the teacher can return counterexamples of any size in response to a failed candidate query.

To analyze the complexity of L^T, we make the following standard assumption: every query to the teacher, whether a membership query or a candidate query, takes unit time and space. Further, since the alphabet Σ of the unknown language U is fixed, we assume that the size of Σ is a constant. Then the following theorem summarizes the complexity of L^T.

Theorem 8 *The algorithm L^T takes $O(mn + n^3)$ time and space where n is the number of states in the minimal DTA for the unknown language U and m is the size of the largest counterexample returned by the teacher.*

Proof. By Lemma 14, we have $|\mathcal{S}| \leq n$. Then the number of rows in the table, which is $|\widehat{\mathcal{S}}| = |\mathcal{S} \cup (\Sigma \cdot \mathcal{S}) \cup (\mathcal{S} + \mathcal{S})|$, is of $O(n^2)$. Further, recall that every time a new experiment is added to $\mathcal{E}$, $|\mathcal{S}|$ increases by one. Hence the number of table columns $|\mathcal{E}| \leq n$, and the number of table entries $|\widehat{\mathcal{S}}||\mathcal{E}|$ is of $O(n^3)$.

The trees and contexts in $\widehat{\mathcal{S}}$ and $\mathcal{E}$ are kept in a DAG form, where common subtrees between different elements in $\widehat{\mathcal{S}}$ and $\mathcal{E}$ are shared in order to keep the space consumption within polynomial bounds. Specifically, recall that whenever a tree t is moved from $\widehat{\mathcal{S}} \setminus \mathcal{S}$ to $\mathcal{S}$, all trees of the form $\alpha \cdot t$ for each $\alpha \in \Sigma$ and $t + t'$ for each $t' \in \mathcal{S}$ (which are $O(|\mathcal{S}|)$ in number) are to be added to $\widehat{\mathcal{S}}$. Adding the tree $\alpha \cdot t$ to $\widehat{\mathcal{S}}$ only needs constant space

since t is already in $\widehat{\mathcal{S}}$ and hence is shared in the DAG representation. Similarly adding a tree of form $t + t'$ takes only constant space, since both t and t' are already in $\widehat{\mathcal{S}}$. Thus, each time $\mathcal{S}$ is expanded, a total of $O(|\mathcal{S}|)$ space is required to add all the new trees to $\widehat{\mathcal{S}}$. Since at most n trees can be added $\mathcal{S}$ in all, it follows that the total space consumed by elements in $\widehat{\mathcal{S}}$ is $O(n^2)$.

Now, we compute the total space consumed by the contexts in $\mathcal{E}$. Note that the teacher can return counterexamples of arbitrary size in response to a wrong conjecture. Suppose m is the size of the largest counterexample. Observe that an experiment is extracted from CE (procedure **ExpGen** in Figure 5.3) essentially by replacing some of the subtrees of CE with trees in $\mathcal{S}$, and exactly one subtree of CE with $\Box$. But, since in the DAG form, common subtrees are shared between trees and contexts in $\mathcal{S}$ and $\mathcal{E}$, none of the above replacements consume any extra space. Hence, the size of the experiment extracted from CE is utmost the size of CE. Since there are at most n contexts in $\mathcal{E}$, the total space consumed by contexts in $\mathcal{E}$ is $O(mn)$. Putting together all observations so far, we get that the total space consumed by L^T is $O(mn + n^3)$.

Now, we compute the time consumed by L^T. It takes $O(n^3)$ membership queries to fill in the $O(n^3)$ table entries. Since each query is assumed to take $O(1)$ time, this takes a total of $O(n^3)$ time. The time taken to extract an experiment from a counterexample CE is linear on the size of CE. This is because, as seen in Figure 5.3, the procedure **ExpGen** involves making a constant number of membership queries for each node of CE (branch conditions in lines 3, 6, and 8) as CE is processed in a top down fashion. Thus, the time taken to extract an experiment from CE is at most $O(m)$. Since there can be at most n wrong conjectures, the total time spent on processing counterexamples is $O(mn)$. Putting these observations together we conclude that L^T takes $O(mn + n^3)$ time. We thus have the following theorem.

5.3 Automating Assume-Guarantee for Simulation

For M_1, M_2 and M_S, suppose we are to check if $M_1 \parallel M_2 \preccurlyeq M_S$. Recall from Section 5.1 that $M_1 \parallel M_2 \preccurlyeq M_S$ if and only if $\mathcal{T}(M_1 \parallel M_2) \subseteq \mathcal{T}(M_S)$, and $\mathcal{T}(M_1 \parallel M_2) = \mathcal{T}(M_1) \cap \mathcal{T}(M_2)$. Therefore, the verification problem is equivalent to checking if $\mathcal{T}(M_1) \cap \mathcal{T}(M_2) \subseteq \mathcal{T}(M_S)$. Now, define $\mathcal{T}_{max} = \overline{\mathcal{T}(M_1) \cap \overline{\mathcal{T}(M_S)}}$. Then

$$\mathcal{T}(M_1) \cap \mathcal{T}(M_2) \;\subseteq\; \mathcal{T}(M_S) \iff \mathcal{T}(M_2) \subseteq \mathcal{T}_{max}$$

Thus, $\mathcal{T}_{max}$ represents the maximal environment under which M_1 satisfies M_S, and

$$M_1 \parallel M_2 \preccurlyeq M_S \;\Leftrightarrow\; \mathcal{T}(M_2) \subseteq \mathcal{T}_{max}$$

Checking $\mathcal{T}(M_2) \subseteq \mathcal{T}_{max}$ is as expensive as directly checking $M_1 \parallel M_2 \preccurlyeq M_S$ since it involves both M_1 and M_2. In the following, we show how the L^T algorithm can be used for a more efficient solution.

Since regular tree languages are closed under intersection and complementation, $\mathcal{T}_{max}$ is a regular tree language. We therefore use the L^T algorithm to learn the canonical DTA for $\mathcal{T}_{max}$ in an incremental fashion. The key idea is that when a candidate query is made by L^T, the teacher checks if the **NC** proof rule can be discharged by using the proposed candidate as the assumption. Empirical evidence (see Section 5.4) suggests that this often succeeds well before $\mathcal{T}_{max}$ is learnt, leading to substantial savings in time and memory consumption.

We now elaborate on how the teacher assumed by L^T is implemented. Specifically, the membership and candidate queries of L^T are processed as follows.

Membership Query. For a given tree t we are to check if $t \in \mathcal{T}_{max}$. This is equivalent to checking if $t \notin \mathcal{T}(M_1)$ or $t \in \mathcal{T}(M_S)$. In our implementation, both $\mathcal{T}(M_1)$ and $\mathcal{T}(M_S)$

are maintained as tree automata, and the above check amounts to membership queries on these automata.

Candidate Query. Given a DTA D we are to check if $\mathcal{L}(D) = \mathcal{T}_{max}$. We proceed in three steps as follows.

1. Check if **(C1)** $\mathcal{L}(D) \subseteq \mathcal{T}_{max} = \overline{\mathcal{T}(M_1) \cap \overline{\mathcal{L}(M_S)}}$. This is implemented using the complementation, intersection and emptyness checking operations on tree automata. If **C1** holds, then we proceed to step 2. Otherwise, we return some $t \in \mathcal{T}_{max} \setminus \mathcal{L}(D)$ as a counterexample to the candidate query D.
2. Check if **(C2)** $\mathcal{T}(M_2) \subseteq \mathcal{L}(D)$. If this is true, then **(C1)** and **(C2)** together imply that $\mathcal{T}(M_2) \subseteq \mathcal{T}_{max}$, and thus our overall verification procedure terminates concluding that $M_1 \parallel M_2 \preccurlyeq M_S$. Note that even though the procedure terminates $\mathcal{L}(D)$ may not be equal to $\mathcal{T}_{max}$. On the other hand, if **(C2)** does not hold, we proceed to step 3 with some $t \in \mathcal{T}(M_2) \setminus \mathcal{L}(D)$.
3. Check if $t \in \mathcal{T}_{max}$, which is handled as in the membership query above. If this is true, then it follows that $t \in \mathcal{T}_{max} \setminus \mathcal{L}(D)$, and hence we return t as a counterexample to the candidate query D. Otherwise, if $t \notin \mathcal{T}_{max}$ then $\mathcal{T}(M_2) \not\subseteq \mathcal{T}_{max}$, and therefore we conclude that $M_1 \parallel M_2 \not\preccurlyeq M_S$.

Thus, the procedure for processing the candidate query can either answer the query or terminate the entire verification procedure with a positive or negative outcome. Further, the reader may note that M_1 and M_2 are never considered together in any of the above steps. For instance, the candidate D is used instead of M_1 in step 1, and instead of M_2 in step 2. Since D is typically very small in size, we achieve significant savings in time and memory consumption, as reported in Section 5.4.

5.4 Experimental Results

Our primary target has been the analysis of concurrent message-passing C programs. Specifically, we have experimented with a set of benchmarks derived from the OpenSSL-0.9.6c source code. We analyzed the source code that implements the critical handshake that occurs when an SSL server and client establish a secure communication channel between them. The server and client source code contained roughly 2500 LOC each. Since these programs have an infinite state space, we constructed finite conservative labeled transition system (LTS) models from them using various abstraction techniques [30][2]. The abstraction process was carried out component-wise.

We designed a set of eight LTS specifications on the basis of the SSL documentation. We verified these specifications on a system composed of one server (M_1) and one client (M_2) using both the brute-force composition ($M_1 \parallel M_2$), and our proposed automated AG approach. All experiments were carried out on a 1800+ XP AMD machine with 3 GB of RAM running RedHat 9.0. Our results are summarized in Table 5.5. The learning based approach shows superior performance in all cases in terms of **memory** consumption (up to a **factor of 12.8**). An important reason behind such improvement is that the sizes of the (automatically learnt) assumptions that suffice to prove or disprove the specification (shown in column labeled $|A|$) are much smaller than the size of the second (client) component (3136 states).

5.5 Conclusion and Related Work

We have presented an automated AG-style framework for checking simulation conformance between LTSs. Our approach uses a learning algorithm L^T to incrementally construct the weakest assumption that can discharge the premises of a non-circular AG proof rule.

[2]Spurious counterexamples arising due to abstraction are handled by iterative counterexample guided abstraction refinement.

Name		Direct		AG		Gain			
	Result	T_1	M_1	T_2	M_2	M_1/M_2	$\|A\|$	MQ	CQ
SSL-1	*Invalid*	*	2146	**325**	**207**	10.4	8	265	3
SSL-2	*Valid*	*	2080	**309**	**163**	12.8	8	279	3
SSL-3	*Valid*	*	2077	**309**	**163**	12.7	8	279	3
SSL-4	*Valid*	*	2076	**976**	**167**	12.4	16	770	4
SSL-5	*Valid*	*	2075	**969**	**167**	12.4	16	767	4
SSL-6	*Invalid*	*	2074	**3009**	**234**	8.9	24	1514	5
SSL-7	*Invalid*	*	2075	**3059**	**234**	8.9	24	1514	5
SSL-8	*Invalid*	*	2072	**3048**	**234**	8.9	24	1514	5

Figure 5.5: Experimental results. Result = specification valid/invalid; T_1 and T_2 are times in seconds; M_1 and M_2 are memory in mega bytes; $|A|$ is the assumption size that sufficed to prove/disprove specification; MQ is the number of membership queries; CQ is the number of candidate queries. A * indicates out of memory (2 GB limit). Best figures are in bold.

Although we have focused on an instantiation of the non-circular rule **NC**, our approach can be directly extended to the circular rule **C** in the manner described in Chapter 3.

We have implemented this framework in the ComFoRT [85] toolkit and experimented with a set of benchmarks based on the OpenSSL source code and the SSL specification. Our experiments indicate that in practice, extremely small assumptions often suffice to discharge the AG premises. This can lead to orders of magnitude improvement in the memory and time required for verification.

The L^T algorithm may be viewed as a branching time analogue of L^* where the Teacher must be capable of answering queries on trees and tree automata (as opposed to traces and finite state machines in L^*). Unlike the algorithms in [20, 62] which learn tree languages offline from a training set, L^T learns actively by querying a teacher. Another algorithm for learning tree languages [53] is closely related to L^T. However, L^T has a better the worst-case complexity of $O(n^3)$ as compared to $O(n^5)$ of the previous algorithm. We note that learning from derivation trees was investigated initially in the context of context-free grammars [119] and forms the basis of several inference algorithms for tree languages [20, 53, 62] including ours.

Chapter 6

Efficient AGR using SAT and Lazy Learning

6.1 Introduction

The automated AGR approach (cf. Chapter 3) using the L^* algorithm is effective for small systems. In order to make it scalable, there are two main challenges:

- *Efficient Teacher Implementation:* The teacher, i.e., the model checker, must be able to answer membership and candidate queries efficiently. More precisely, each query may itself involve exploration of a large state space making explicit-state model checking infeasible.
- *Alphabet explosion:* Suppose the system to be verified consists of two components M_1 and M_2. If M_1 and M_2 interact using a set X of global shared communication variables, the alphabet of the assumption A consists of all the valuations of X and is exponential in size of X. The learning algorithm (cf. Chapter 2) explicitly enumerates the alphabet set at each iteration and performs membership queries for enumeration step. Therefore, it is prohibitively expensive to apply L^* directly to shared memory

systems with a large number of shared communication variables. Indeed, it is sometimes impossible to enumerate the full alphabet set, let alone learning an assumption hypothesis. We refer to this problem as the *alphabet explosion* problem.

In this chapter, we address the above problems by (i) efficiently implementing the teacher using SAT-based model checking; and (ii) a *lazy* learning approach for mitigating the alphabet explosion problem.

6.2 Notation and Preliminaries

We first define the notions of symbolic transition systems, automata, and composition which we will use in the rest of the chapter. Our formalism borrows notation from [94, 108].

Let $X = \{x_1, \ldots, x_n\}$ be a finite set of typed variables defined over a non-empty finite domain of values $\mathcal{D}$. We define a *label* a as a total map from X to $\mathcal{D}$ which maps each variable x_i to value d_i. An X-*trace* ρ is a finite sequence of labels on X. The next-time label is $a' = a\langle X/X'\rangle$ is obtained from a by replacing each $x_i \in dom(a)$ by x'_i. Given a label a over X and $Y \subseteq X$, we define the label projection $a\,|_Y = a'$ where $dom(a') = Y$ and $a(y) = a'(y)$ for each $y \in Y$. Given variables X and the corresponding next-time variables X', let us denote the (finite) set of all predicates on $X \cup X'$ by Φ_X ($true$ and $false$ denote the boolean constants). Given labels a and b on X, we say that a label pair (a, b') satisfies a predicate $\phi \in \Phi_X$, denoted $\phi(a, b')$, if ϕ evaluates to $true$ under the variable assignment given by a and b'.

6.2.1 Communicating Finite Automata

A *communicating finite automata* (CFA) C on a set of variables X (called the support set) is a tuple $\langle X, Q, q0, \delta, F\rangle$; Q denotes a finite set of states, $q0$ is the initial state, $\delta \subseteq Q \times \Phi_X \times Q$ is the transition relation and F is the set of final states. For states $q, q' \in Q$ and $\phi \in \Phi_X$, if $\delta(q, \phi, q')$ holds, then we say that ϕ is a transition predicate

between q and q'. For each state q, we define its follow set $fol(q)$ to be the set of outgoing transition predicates, i.e., $fol(q) = \{\phi | \exists q' \in Q.\ \delta(q, \phi, q')\}$. We say that $fol(q)$ is complete iff $\bigvee\{\phi \in fol(q)\} = true$ and disjoint iff for all $\phi_i, \phi_j \in fol(q)$, $\phi_i \wedge \phi_j = false$. Also, we say that δ is complete (deterministic) iff for each $q \in Q$, $fol(q)$ is complete (disjoint). The alphabet Σ of C is defined to be the set of label pairs (a, a') on variables X and X'. The above definition of transitions (on current and next-time variables) allows compact representation of CFAs and direct composition with STSs below.

A *run* of C is defined to be a sequence $(q_0, \ldots, q_n)$ of states in Q such that $q_0 = q0$. A run is said to be accepting if $q_n \in F$. Given a W-trace ($X \subseteq W$), $\rho = a_0, \ldots, a_n$, is said to be a trace of C if there exists an accepting run $(q_0, \ldots, q_n)$ of C, such that for all $j < n$, there exists a predicate ϕ, such that $\delta(q_j, \phi, q_{j+1})$ and $\phi(a_j, a'_{j+1})$ holds. In other words, the labels a_j and a_{j+1} must satisfy some transition predicate between q_j and q_{j+1}. The W-trace language $\mathbb{L}_W(C)$ is the set of all W-traces of C. Note that this definition of W-trace allows a sequence of labels on X to be *extended* by all possible valuations of variables in $W \setminus X$ and eases the definition of the composition operation below. In general, we assume W is the universal set of variables and write $\mathbb{L}(C)$ to denote the language of C.

A CFA can be viewed as an ordinary finite automaton with alphabet Σ which accepts a regular language over Σ. While the states are represented explicitly, the *follow* function allows clustering a set of alphabet symbols into one transition symbolically. The common automata-theoretic operations, viz., union, intersection, complementation and determinization via subset-construction can be directly extended to CFAs. The complement of C is denoted by $\overline{C}$, where $\mathbb{L}(\overline{C}) = \overline{\mathbb{L}(C)}$. Note that the constraint **LC1** (cf. Chapter 3) holds for CFA.

Definition 24 (Product of CFAs.) *Given CFAs* $C_1 = \langle X_1, Q_1, q0_1, \delta_1, F_1\rangle$ *and* $C_2 = \langle X_2, Q_2, q0_2, \delta_2, F_2\rangle$*, their product* $C = C_1 \times C_2$ *is a tuple* $\langle X, Q, q0, \delta, F\rangle$ *where* $X = X_1 \cup X_2$, $Q = Q_1 \times Q_2$, $q0 = (q0_1, q0_2)$, $F = F_1 \times F_2$ *and for a label* c *over* $X \cup X'$, $q_1, q'_1 \in Q_1$

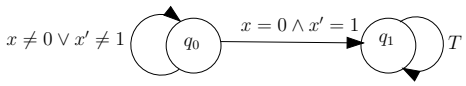


Figure 6.1: A CFA on support $X = \{x\}$; x is a boolean. $\Sigma = \{(x = 0, x' = 0), (x = 0, x' = 1), (x = 1, x' = 0), (x = 1, x' = 1)\}$. $fol(q_0) = \{(x = 0 \wedge x' = 1), (x \neq 0 \vee x' \neq 1)\}$. $fol(q_1) = \{true\}$. Note that the first element of $fol(q_0)$ corresponds to an alphabet symbol while the second element is an alphabet cluster. Also, both $fol(q_0)$ and $fol(q_1)$ are disjoint and complete.

and $q_2, q_2' \in Q_2$, $(q_1', q_2') \in \delta((q_1, q_2), c)$ *iff* $q_1' \in \delta_1(q_1, c|_{X_1 \cup X_1'})$ *and* $q_2' \in \delta_2(q_2, c|_{X_2 \cup X_2'})$.

Lemma 15 *For CFAs* C_1 *and* C_2, $\mathbb{L}(C_1 \times C_2) = \mathbb{L}(C_1) \cap \mathbb{L}(C_2)$.

Definition 25 (Support set of a Language) *We define the support* $Spt(L)$ *of a regular language* L *recursively as follows:*

- *If* $L = \mathbb{L}(C)$ *for a CFA* C *with support set* X, *then* $Spt(L) = X$.
- *If* $L = L_1 \cap L_2$ *for languages* L_1 *and* L_2, *then* $Spt(L) = Spt(L_1) \cup Spt(L_2)$.
- *If* $L = \overline{L_1}$, *for a language* L_1, *then* $Spt(L) = Spt(L_1)$.

It follows that for $L = L_1 \cup L_2 = \overline{\overline{L_1} \cap \overline{L_2}}$, $Spt(L) = Spt(L_1) \cup Spt(L_2)$.

Lemma 16 *A regular language with support* X *is accepted by a CFA with support* X.

Proof. We prove by structural induction over the definition of $Spt(L)$. The base case holds trivially since $L = \mathbb{L}(C)$ for a C with support set X. If $L = L_1 \cap L_2$, by inductive hypothesis, there must exist CFAs C_1 and C_2 where $L_1 = \mathbb{L}(C_1)$ and $L_2 = \mathbb{L}(C_2)$, with support sets X_1 and X_2 respectively, so that $X = X_1 \cup X_2$. Let $C = C_1 \times C_2$. Now, $\mathbb{L}(C) = \mathbb{L}(C_1) \cap \mathbb{L}(C_2) = L$. Therefore, L is accepted by the CFA C whose support set is X. Again, if $L = \overline{L_1}$ on support set X, there exist a CFA C_1 on support set X, so that $\mathbb{L}(C_1) = L_1$. Let C denote the CFA obtained by determinizing and complementing C_1. Note that C has support X and $\mathbb{L}(C) = \overline{\mathbb{L}(C_1)} = L$.

6.2.2 Symbolic Transition Systems

A *symbolic transition system* (STS) M is a tuple $\langle X, S, I, R, F\rangle$, defined over a set of variables X called its *support*, where S consists of all labels over X, $I(X)$ is the initial state predicate, $R(X, X')$ is the transition predicate and $F(X)$ is the final state predicate. Given a variable set W ($X \subseteq W$), a W-trace $\rho = a_0, \ldots, a_n$ is said to be a trace of M if $I(a_0)$ and $F(a_n)$ hold and for all $j < n$, $R(a_j, a'_{j+1})$ holds. The trace language $\mathbb{L}(M)$ of M is the set of all traces of M.[1]

CFA as an STS. Given a CFA $C = \langle X_C, Q_C, q0_C, \delta_C, F_C\rangle$, there exists an STS $M = \langle X, S, I, R, F\rangle$ such that $\mathbb{L}(C) = \mathbb{L}(M)$. We construct M as follows: (i) $X = X_C \cup \{q\}$ where q is a fresh variable which ranges over Q_C, (ii) $I(X) = (q = q0)$, (iii) $F(X) = \exists q_i \in F_C.(q = q_i)$, and (iv) $R(X, X') =$

$$(\exists q_1, q_2 \in Q_C, \phi \in \Phi.\ (q = q_1 \wedge q' = q_2 \wedge \delta_C(q_1, \phi, q_2) \wedge \phi(X_C, X'_C))$$

Synchronous Composition of STSs. Suppose we are given two STSs $M_1 = \langle X_1, S_1, I_1, R_1, F_1\rangle$ and $M_2 = \langle X_2, S_2, I_2, R_2, F_2\rangle$. We define the composition $M_1 \parallel M_2$ to be a STS $M = \langle X, S, I, R, F\rangle$ where: (i) $X = X_1 \cup X_2$, (ii) S consists of all labels over X, (iii) $I = I_1 \wedge I_2$, (iv) $R = R_1 \wedge R_2$, and (v) $F = F_1 \wedge F_2$.

Lemma 17 (LC2 for STSs) *(cf. Chapter 3) Given two STSs M_1 and M_2, $\mathbb{L}(M_1 \parallel M_2) = \mathbb{L}(M_1) \cap \mathbb{L}(M_2)$.*

We use STSs to represent system components and CFA on shared variables to represent assumptions computed in the various AGR sub-tasks, respectively. We assume that all STSs have total transition predicates. We define the composition of an STS M with a CFA C, denoted by $M \parallel C$, to be $M \parallel M_C$, where M_C is the STS obtained from C. Although we use a synchronous notion of composition in this chapter, our work can be directly extended to asynchronous composition also.

[1]We overload the symbol $\mathbb{L}()$ to describe the trace language of both CFAs and STSs.

Lemma 18 (LC2 for STSs and CFAs) *The constraint* **LC2** *holds for STS and CFA models together, i.e., for STS M and CFA C, $\mathbb{L}(M \parallel C) = \mathbb{L}(M) \cap \mathbb{L}(C)$.*

Proof. From definition $\mathbb{L}(M \parallel C) = \mathbb{L}(M \parallel M_C) = \mathbb{L}(M) \cap \mathbb{L}(M_C) = \mathbb{L}(M) \cap \mathbb{L}(C)$ since $\mathbb{L}(M_C) = \mathbb{L}(C)$.

Definition 26 (Model Checking STSs) *Given an STS M and a property CFA P, the model checking question is to determine if $M \vDash P$ where $\vDash$ denotes a conformance relation. Using the trace semantics for STSs and CFAs and set containment as the conformance relation, the problem can be reduced to checking if $\mathbb{L}(M) \subseteq \mathbb{L}(P)$.*

Since CFAs are closed under negation and there is a language-equivalent STS for each CFA, we can further reduce the model checking question to checking if $\mathbb{L}(M \parallel M_{\overline{P}})$ is empty, where the STS $M_{\overline{P}}$ is obtained by complementing P to form $\overline{P}$ and then converting it into an STS. Let STS $\mathcal{M} = M \parallel M_{\overline{P}}$. In other words, we are interested in checking if there is an *accepting* trace in $\mathcal{M}$, i.e., a trace that ends in a state that satisfies $F_{\mathcal{M}}$.

6.2.3 SAT-based Model Checking

It is possible to check for existence of an accepting trace in an STS $\mathcal{M}$ using satisfiability checking. A particular instance of this problem is bounded model checking [21] where we check for existence of an accepting trace of length k using a SAT solver.

Bounded Model Checking(BMC). Given an integer bound k, the BMC problem can be formulated in terms of checking satisfiability of the following formula [21]:

$$BMC(\mathcal{M}, k) \;:=\; I_{\mathcal{M}}(s_0) \wedge \bigwedge_{0 \le j \le k-1} R_{\mathcal{M}}(s_j, s_{j+1}) \wedge \bigvee_{0 \le j \le k} F_{\mathcal{M}}(s_j) \tag{6.1}$$

Here s_j $(0 \le j \le k)$ represents the set of variables $X_{\mathcal{M}}$ at depth j. The transition relation of $\mathcal{M}$ is unfolded up to k steps, conjuncted with the initial and the final state predicates at the first and the last steps respectively, and finally encoded as a propositional formula

that can be solved by a SAT solver. If the formula is SAT then the satisfying assignment corresponds to an accepting trace of length k (a counterexample to $M \vDash P$). Otherwise, no accepting trace exists of length k or less. It is possible to check for accepting traces of longer lengths by increasing k and checking iteratively.

Unbounded Model Checking(UMC). The unbounded model checking problem involves checking for an accepting trace of any length. Several SAT-based approaches have been proposed to solve this problem [113]. Here, we consider two approaches, one based on k-induction [14, 56, 120] and the other based on interpolation [102].

The k-induction technique [120] tries to show that there are no accepting traces of any length with the help of two SAT checks corresponding to the base and induction cases of the UMC problem. In the base case, it shows that no accepting trace of length k or less exists. This exactly corresponds to the BMC formula (Eq. 6.1) being UNSAT. In the induction step, it shows that if no accepting trace of length k or less exists, then there cannot be an accepting trace of length $k+1$ in $\mathcal{M}$, and is represented by the following formula:

$$Step(\mathcal{M}, k) := \bigwedge_{0 \le j \le k} R_{\mathcal{M}}(s_j, s_{j+1}) \wedge \bigwedge_{0 \le j \le k} \neg F_{\mathcal{M}}(s_j) \wedge F_{\mathcal{M}}(s_{k+1}) \wedge \bigwedge_{0 \le i \le j \le k} s_i \neq s_{j+1} \quad (6.2)$$

The induction step succeeds if $Step(\mathcal{M}, k)$ is UNSAT. Otherwise, the depth k is increased iteratively until it succeeds or the base step is SAT (a counterexample is found). The set of constraints of form $s_i \neq s_{j+1}$ in (Eq. 6.2) (also known as simple path or uniqueness constraints) are necessary for completeness of the method and impose the condition that all states in the accepting trace must be unique. The method can be implemented efficiently using an incremental SAT solver [56], which allows reuse of recorded conflict clauses in the SAT solver across iterations of increasing depths. The k-induction technique has the drawback that it may require as many iterations as the length of the longest simple path between any two states in $\mathcal{M}$ (also known as recurrence diameter [21]), which may be

exponentially larger than the longest of all the shortest paths (or the diameter) between any two states.

Another approach to SAT-based UMC is based on using interpolants [102]. The method computes an over-approximation $\mathcal{I}$ of the reachable set of states in $\mathcal{M}$, which is also an inductive invariant for $\mathcal{M}$, by using the UNSAT proof of the BMC instance (Eq. 6.1). If $\mathcal{I}$ does not overlap with the set of final states, then it follows that there exists no accepting trace in $\mathcal{M}$. An important feature of this approach is that it does not require unfolding the transition relation beyond the diameter of the state space of $\mathcal{M}$, and, in practice, often succeeds with shorter unfoldings. We do not present the details of this approach here; they can be found in [9, 102].

In order to use a SAT solver, the above formula instances have to be translated into propositional logic. A lot of structural information is lost (e.g., relation between bits of an encoded variable) due to this translation and may lead to useless computation by the SAT solver. We can avoid this translation by using an SMT solver [3, 125]. Besides allowing propositional constraints, an SMT solver also supports input formulas in one or more (ground) first order theories, e.g., the quantifier-free fragment of linear arithmetic over integers. Therefore, both BMC and UMC based on k-induction can be carried out using a SMT solver, provided it supports the theories over which the above formulas are defined. A particular mixed boolean/integer encoding of hardware RTL constructs can be found in [24]. Similarly, interpolation-based UMC may be carried out using an interpolating prover provided it can generate interpolants in the required theories.

6.3 Assume-Guarantee Reasoning using Learning

We now present the automated AGR framework for STSs and CFAs using learning. Here, we will be concerned mainly with the instantiation of the non-circular AGR rule presented in Chapter 3 for STSs and CFAs.

Definition 27 Non-circular AGR (NC) *Given STSs M_1, M_2 and CFA P, show that $M_1 \parallel M_2 \vDash P$, by picking an assumption CFA A, such that both* **(n1)** $M_1 \parallel A \vDash P$ *and* **(n2)** $M_2 \vDash A$ *hold.*

Although we focus our presentation on **NC** rule, our results can be applied to an instantiation of the circular rule **C** (cf. Chapter 3) in a straightforward way. We implement and experiment with both the rules (cf. Section 6.5).

Lemma 19 (Soundness and Completeness) *Both instances of* **NC** *and* **C** *rules are sound and complete for STSs and CFAs.*

Proof. It follows from the proof of soundness and completeness of the abstract rule in Chapter 3 and the fact that **CL1** holds for CFAs (see Section 6.2.1) and **CL2** holds for STSs and CFAs (from Lemma 18).

Moreover, both the rules can be extended to a system of n STSs $M_1 \dots M_n$ by picking a set of assumptions (represented as a tuple) $\langle A_1 \dots A_{n-1} \rangle$ for **NC** and $\langle A_1 \dots A_n \rangle$ for **C** respectively. The proofs of completeness for both these rules rely on the notion of weakest assumptions (cf. Chapter 3). The following lemma shows that CFAs are suitable for representing weakest assumptions.

Lemma 20 (Weakest Assumptions) *Given a finite STS M with support set X_M and a CFA P with support set X_P, there exists a unique weakest assumption CFA, WA, such that (i) $M \parallel WA \vDash P$ holds, and (ii) for all CFA A where $M \parallel A \vDash P$, $\mathbb{L}(A) \subseteq \mathbb{L}(WA)$ holds. Moreover, $\mathbb{L}(WA)$ is regular and the support variable set of WA is $X_M \cup X_P$.*

Proof. It follows from the Lemma 17 and Lemma 18 that the weakest assumption language $L_W = \overline{\mathbb{L}(M)} \cup \mathbb{L}(P)$. Also, L_W is regular since $\mathbb{L}(M)$ and $\mathbb{L}(P)$ are regular. The model class of deterministic CFAs uniquely covers the regular language set. Hence, by Lemma 8, there exists an unique deterministic CFA WA, so that $\mathbb{L}(WA) = L_W$. Since, the support set of $\overline{\mathbb{L}(M)}$ and $\mathbb{L}(P)$ is X_M and X_P respectively, the support of $\mathbb{L}(WA)$ and therefore WA is $X_M \cup X_P$.

As mentioned earlier (cf. Chapter 3), a learning algorithm for regular languages, L^*, assisted by a model checker based Teacher, can be used to automatically generate the assumptions. However, there are problems in scaling this approach to large shared memory systems. Firstly, the Teacher must be able to discharge the queries efficiently even if it involves exploring a large state space. Secondly, the alphabet Σ of an assumption A is exponential in its support set of variables. Since L^* explicitly enumerates Σ during learning, we need a technique to curb this alphabet explosion. We address these problems by proposing a SAT-based implementation of the Teacher and a lazy algorithm based on alphabet clustering and iterative partitioning (Section 6.4).

6.3.1 SAT-based Assume-Guarantee Reasoning

We now show how the Teacher can be implemented using SAT-based model checking. The Teacher needs to answer membership and candidate queries.

Membership Query. Given a trace t, we need to check if $t \in \mathbb{L}(WA)$ which corresponds to checking if $M_1 \parallel \{t\} \models P$ holds. To this end, we first convert t into a language-equivalent STS M_t, obtain $M = M_1 \parallel M_t$ and perform a single BMC check $BMC(M, k)$ (cf. Section 6.2.3) where k is the length of trace t. Note that since M_t accepts only at the depth k, we can remove the final state constraints at all depths except k. The Teacher replies with a TRUE answer if the above formula instance is UNSAT; otherwise a FALSE answer is returned.

Candidate Query. Given a deterministic CFA A, the candidate query involves checking the two premises of **NC**, i.e., whether both $M_1 \parallel A \models P$ and $M_2 \models A$ hold. The latter check maps to SAT-based UMC (cf. Section 6.2.3) in a straightforward way. Note that since A is deterministic, complementation does not involve a blowup. For the previous check, we first obtain an STS $M = M_1 \parallel M_A$ where the STS M_A is language-equivalent to A (cf. Section 6.2) and then use SAT-based UMC for checking $M \models P$.

In our implementation, we employ both induction and interpolation for SAT-based UMC. Although the interpolation approach requires a small number of iterations, computing interpolants, in many cases, takes more time in our implementation. The induction-based approach, in contrast, is faster if it converges within small number of iterations.

Now, automated AGR is carried out in the standard way (cf. Chapter 3) based on the above queries. The learner sets the support variable set for the assumption A to the support of the weakest assumption ($X_{wa} = X_{M_1} \cup X_P$) and iteratively computes hypotheses assumptions by asking membership and candidate queries until **n1** holds. The last assumption is then presented in a candidate query which checks if **n2** holds. If **n2** holds, then the procedure terminates. Otherwise, a counterexample ce is returned. ce may be spurious; a membership query on ce is used to check if it is spurious. In that case, ce is projected to X_{wa} to obtain ce' and learning continues with the ce'. Otherwise, ce is returned as an actual counterexample to $M_1 \parallel M_2 \models P$. The termination of this procedure is guaranteed by the existence of a unique weakest assumption WA. However, it is important to note that we seldom need to compute WA. In practice, this procedure terminates with any assumption A that satisfies **n1** and **n2** and the size of A is much smaller than that of WA.

6.4 Lazy Learning

This section presents our new lazy learning approach to address the alphabet explosion problem. In contrast to the eager BDD-based learning algorithm [115], the lazy approach (i) avoids use of quantifier elimination to compute the set of edges and (ii) introduces new states and transitions lazily only when necessitated by a counterexample. We first propose a generalization of the L^* algorithm (cf. Chapter 2) and then present the lazy l^* algorithm based on it.

6.4.1 Generalized L^* Algorithm

Recall that given an unknown language $\mathbb{L}_U$ defined over alphabet Σ, L^* maintains an observation table $\mathcal{T} = (U, UA, V, T)$ consisting of trace *samples* from $\mathbb{L}_U$, where $U \subseteq \Sigma^*$ is a prefix-closed set, $V \subseteq \Sigma^*$ is a set of suffixes, UA contains extensions of elements in U and T is a map so that $T(u, v) = [\![u \cdot v]\!]$ for some $u \in U \cup UA$ and $v \in V$. We generalize the L^* using the notion of *follow* sets for each $u \in U$.

Definition 28 (Follow Sets) *For each $u \in \Sigma^*$, we define a follow function $follow : \Sigma^* \to 2^\Sigma$, where $follow(u)$ consists of the set of alphabet symbols $a \in \Sigma$ that u is extended by in order to form $u \cdot a$. We say that $follow(u)$ is the follow set for u.*

We also assume that there exists a *cluster mapping* function $cmap : \Sigma^* \times \Sigma \to 2^\Sigma$, which maps each $u \in \Sigma^*$ and $a \in follow(u)$ to the alphabet cluster $cmap(u, a)$.

The basis of our generalization of L^* is the *follow* function; instead of allowing each $u \in \Sigma^*$ to be extended by the full alphabet Σ as in original L^*, we only allow u to be extended by the elements in $follow(u)$. More precisely, the generalized algorithm differs from the original algorithm in the following ways:

- We initialize $follow(u) = \Sigma$ for each $u \in \Sigma^*$.
- We redefine the `FillAllSuccs` procedure; instead of iterating over all $a \in \Sigma$, now `FillAllSuccs` only iterates over $follow(u)$ for a given u.
- In the procedure `MkDFA`, the transition relation is determined using the follow set, i.e., $\delta([u], a) = [u \cdot a]$ for all $u \in U$ and $a \in follow(u)$.

Note that with $follow(u) = \Sigma$ (for each u) the generalized algorithm reduces to the original algorithm presented in Chapter 2.

6.4.2 Lazy l^* Algorithm

The main bottleneck in generalized L^* algorithm is due to alphabet explosion, i.e., it enumerates and asks membership queries on all extensions of an element $u \in U$ on the (exponential-sized) Σ explicitly. The lazy approach avoids this as follows. Initially, the follow set for each u contains a singleton element, the alphabet cluster $true$, which requires only a single enumeration step. This cluster may then be partitioned into smaller clusters in the later learning iterations, if necessitated by a counterexample. In essence, the lazy algorithm not only determines the states of the unknown CFA, but also computes the set of distinct alphabet clusters outgoing from each state lazily.

More formally, l^* performs queries on trace sets, wherein each transition corresponds to an alphabet cluster. We therefore augment our learning setup to handle sets of traces. Let $\hat{\Sigma}$ denote the set 2^{Σ} and concatenation operator $\cdot$ be extended to sets of traces S_1 and S_2 by concatenating each pair of elements from S_1 and S_2 respectively. The follow function is redefined as $follow : \hat{\Sigma}^* \rightarrow 2^{\hat{\Sigma}}$ whose range now consists of alphabet cluster elements (or alphabet predicates). The observation table $\mathcal{T}$ is a tuple (U, UA, V, T) where $U \subseteq \hat{\Sigma}^*$ is prefix-closed, $V \subseteq \hat{\Sigma}^*$ and UA contains all extensions of elements in U on elements in their follow sets. $T(u, v)$ is defined on a sets of traces u and v, so that $T(u, v) = [\![u \cdot v]\!]$ where the membership function $[\![\cdot]\!]$ is extended to a set of traces as follows: given a trace set S, $[\![S]\!] = 1$ iff $\forall t \in S.\ [\![t]\!] = 1$. In other words, a $[\![S]\!] = 1$ iff $S \subseteq \mathbb{L}_U$. This definition is advantageous in two ways. Firstly, the SAT-based Teacher (cf. Section 6.3.1) can answer membership queries in the same way as before by converting a single trace set into the corresponding SAT formula instance. Secondly, in contrast to a more discriminating 3-valued interpretation of $[\![S]\!]$ in terms of 0, 1 and *undefined* values, this definition enables l^* to be more lazy with respect to state partitioning.

Figure 6.2 shows the pseudocode for the procedure `LearnCE`, which learns from a counterexample ce and improves the current hypothesis CFA C. `LearnCE` calls the `LearnCE_0`

and `LearnCE_1` procedures to handle negative and positive counterexamples respectively. `LearnCE_0` is the same as `LearnCE` in generalized L^*: it finds a split of ce at position i (say, $ce = u_i \cdot v_i = u_i \cdot o_i \cdot v_{i+1}$), so that $\alpha_i \neq \alpha_{i+1}$ and adds a new distinguishing suffix v_{i+1} (which must exist by Lemma 21 below) to V to partition the state corresponding to $[u_i \cdot o_i]$. The procedure `LearnCE_1`, in contrast, may either partition a state or partition an alphabet cluster. The case when v_{i+1} is not in V is handled as above and leads to a state partition. Otherwise, if v_{i+1} is already in V, `LearnCE_1` first identifies states in the current hypothesis CFA C corresponding to $[u_i]$ and $[u_i \cdot o_i]$, say, q and q' respectively, and the transition predicate ϕ corresponding to the transtion on symbol o_i from q to q'. Let $u_r = [u_i]^r$. Note that ϕ is also an alphabet cluster in $follow(u_r)$ and if $o_i = (a_i, b_i')$, then $\phi(a_i, b_i')$ holds (cf. Section 6.2).

The procedure `PartitionTable` is then used to partition ϕ using o_i (into $\phi_1 = \phi \wedge o_i$ and $\phi_2 = \phi \wedge \neg o_i$) and update the follow set of u_r by removing ϕ and adding ϕ_1 and ϕ_2. Note that U and UA may also contain extensions of $u_r \cdot \phi$, given by $Uext$ and $UAext$ respectively. In order to keep U prefix-closed and have only extensions of U in UA, the procedure removes $Uext$ and $UAext$ from U and UA respectively. Finally, it adds the extensions of u_r on the new follow set elements ϕ_1 and ϕ_2 to UA and performs the corresponding membership queries.

Note that since all the follow sets are disjoint and complete at each iteration, the hypothesis CFA obtained from a closed table $\mathcal{T}$ is always deterministic and complete (cf. Section 6.2).

Example. Figure 6.3 illustrates the l^* algorithm for the unknown language $\mathbb{L}_U = (a|b|c|d) \cdot (a|b)^*$. Recall that the labels a, b, c and d are, in fact, predicates over program variables. The upper and lower parts of the table represent U and UA respectively, while the columns contain elements from V. The Boolean table entries correspond to the membership query $[\![u \cdot v]\!]$ where u and v are the row and column entries respectively. The algorithm initializes both U and V with element ϵ and fills the corresponding table entry

Init: $\forall u \in \Sigma^*$, set $follow(u) = \{true\}$

`LearnCE`(ce)
if ($[\![ce]\!] = 0$)
 `LearnCE_0(ce)`
else `LearnCE_1(ce)`

`LearnCE_1`(ce)
Find i so that $\alpha_i = 1$ and $\alpha_{i+1} = 0$
if $v_{i+1} \notin V$
 $V := V \cup \{v_{i+1}\}$
 For all $u \in U \cup UA$: `Fill(u, v`$_{i+1}$`)`
else
 Let $ce = u_i \cdot o_i \cdot v_{i+1}$
 Let $q = [u_i]$ and $q' = [u_i \cdot o_i]$
 Suppose $R_C(q, \phi, q')$ and $o_i \in \phi$
 `PartitionTable(`$[u_i]^r$`, `ϕ`, `o_i`)`

`LearnCE_0`(ce)
Find i so that $\alpha_i = 0$ and $\alpha_{i+1} = 1$
$V := V \cup \{v_{i+1}\}$
For all $u \in U \cup UA$: `Fill(u, v`$_{i+1}$`)`

`PartitionTable`(u_r, ϕ, a)
$\phi_1 := \phi \wedge a$, $\phi_2 := \phi \wedge \neg a$
$follow(u_r) := follow(u_r) \cup \{\phi_1, \phi_2\} \setminus \{\phi\}$
Let $Uext = \{u \in U \mid \exists v \in \hat{\Sigma}^*.\ u = u_r \cdot \phi \cdot v\}$
Let $UAext = \{u \cdot \phi_f \mid u \in Uext \wedge \phi_f \in follow(u)\}$
$U := U \setminus Uext$
$UA := UA \setminus UAext$
For $u \in \{u_r \cdot \phi_1, u_r \cdot \phi_2\}$
 $UA := UA \cup \{u\}$
 For all $v \in V$: `Fill`(u, v)

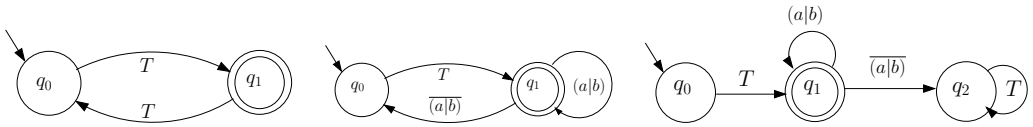
Figure 6.2: Pseudocode for the lazy l^* algorithm (mainly the procedure `LearnCE`).

by asking a membership query. Then, it asks query for a single extension of ϵ on cluster T (the L^* algorithm will instead asks queries on each alphabet element explicitly). Since $\epsilon \not\equiv T$, in order to make the table closed, the algorithm further needs to query on the trace $T \cdot T$. Now, it constructs the first hypothesis (Figure 6.3(i)) and asks a candidate query with it. The teacher replies with a counterexample $a \cdot a$, which is then used to partition the follow set of T into elements a and $\bar{a}$. The table is updated and the algorithm continues iteratively. The algorithm converges to the final CFA using four candidate queries; the figure shows the hypotheses CFAs for first, third and last queries. The first three queries are unsuccessful and return counterexamples $a \cdot a$ (positive), $a \cdot b$ (positive), $a \cdot d \cdot c$ (negative). The first two counterexamples lead to cluster partitioning (by a and b respectively) and the third one leads to state partitioning. Note that the algorithm avoids explicitly enumerating the alphabet set for computing extensions of elements in Σ. Also, note that the algorithm is insensitive to the size of alphabet set to some extent: if $\mathbb{L}_U$ is of the form $\Sigma \cdot (a|b)^*$, the algorithm always converges in the same number of iterations since only two cluster partitions from state q_1 need to be made. The drawback of this lazy approach is

	ϵ	
ϵ	0	(q_0)
T	1	(q_1)
T$\cdot$ T	0	

	ϵ	
ϵ	0	(q_0)
T	1	(q_1)
T$\cdot a$	1	
T$\cdot b$	1	
T$\cdot\overline{(a\|b)}$	0	

	ϵ	c	
ϵ	0	1	(q_0)
T	1	0	(q_1)
T$\cdot\overline{(a\|b)}$	0	0	(q_2)
T$\cdot a$	1	0	
T$\cdot b$	1	0	
T$\cdot\overline{(a\|b)}\cdot$ T	0	0	



(i) (ii) (iii)

Figure 6.3: Illustration of the l^* algorithm for $\mathbb{L}_U = (a|b|c|d)(a|b)^*$. Rows and column represent elements of $U \cup UA$ and V respectively. Alphabets are represented symbolically: T $= (a|b|c|d)$, $\overline{(a|b)} = (c|d)$.

that it may require more candidate queries as compared to the generalized L^* in order to converge. This is because the algorithm is lazy in obtaining information on the extensions of elements in U and therefore builds candidates using less information, e.g., it needs two candidate queries to be able to partition the cluster T on both a and b (note that the corresponding counterexamples $a \cdot a$ and $a \cdot b$ differ only in the last transition). We have developed a SAT-based method (presented below) that accelerates learning in such cases by generalizing a counterexample ce to include a set of similar counterexamples (ce') and then using ce' to perform a *coarser* cluster artition.

Lemma 21 *The procedure* `LearnCE_0` *must lead to addition of at least one new state in the next hypothesis CFA.*

Proof. We first show that $v_i \notin V$. Suppose $v_i \in V$. We know that $\alpha_i = [\![[u_i]^r \cdot o_i \cdot v_i]\!] = 0$ and $\alpha_{i+1} = [\![[u_i \cdot o_i]^r \cdot v_i]\!] = 1$. Also there must exist $\phi \in follow([u_i]^r)$ so that $o_i \in \phi$ and $T([u_i]^r \cdot A, v_i) = T([u_i \cdot o_i]^r, v_i) = 1$. Therefore, by definition, $\forall a \in A$, $[\![[u_i]^r \cdot a \cdot v_i]\!] = 1$. But, $o_i \in \phi$ and $[\![[u_i]^r \cdot o_i \cdot v_i]\!] = 0$. Contradiction.

Let $ua = ([u_i]^r \cdot \phi)$ and $u' = ([[u_i]^r \cdot \phi]^r)$. Adding v_i to V makes $ua \not\equiv u'$ which were equivalent earlier. Moreover, since u' must be in U already, both ua and u' must be

inequivalent to all other $u \in U$. Therefore, `CloseTable` must add ua to U and therefore `MkDFA` will add at least one new state in the next hypothesis.

Lemma 22 *The procedure* `LearnCE_1` *either leads to addition of at least one new state or one transition in the next hypothesis CFA.*

Proof. If $v_i \notin V$, we can argue that at least one state will be added in a way similar to the previous lemma. If $v_i \in V$, then we know that $[\![[u_i]^r \cdot o_i \cdot v_i]\!] = 1$ and there exists $\phi \in follow([u_i]^r)$ so that $o_i \in \phi$ and $[\![[u_i]^r \cdot \phi \cdot v_i]\!] = 0$. In this case, `LearnCE_1` splits the cluster ϕ into $\phi_1 = \phi \wedge o_i$ and $\phi_2 = \phi \wedge \neg o_i$. It follows from definition of $[\![\cdot]\!]$ that $[\![[u_i]^r \cdot \phi_1 \cdot v_i]\!] = 1$ and $[\![[u_i]^r \cdot \phi_2 \cdot v_i]\!] = 0$. Hence, ϕ_1 and ϕ_2 must go to different states, causing addition of at least one transition.

Remark. Although `LearnCE_1` may add a transition, the number of states in the next hypothesis may decrease. This is because partitioning a cluster may also cause a state partition causing $[u_i]^r$ to split into two previously existing states, i.e., the new partitioned traces may become equivalent to some previously existing elements of U.

Theorem 9 *l^* terminates in $O(k \cdot 2^n)$ iterations where k is the alphabet size and n is the number of states in the minimum deterministic CFA C_m corresponding to $\mathbb{L}_U$.*

Proof. Consider the prefix tree PT obtained from the prefix-closed set of elements in U. Note that each node in PT corresponds to a different state (equivalence class) in a hypothesis CFA C. Also, consider computation tree CT obtained by unrolling the transition structure of C_m. Note that PT of depth d can be embedded into CT where different nodes in PT at a given depth k ($k \leq d$) correspond to different (possibly overlapping) subset of states in CT at depth k. `LearnCE_0` introduces a new node in PT while `LearnCE_1` partitions an alphabet cluster outgoing from some node in PT, so that the size of each of the new clusters is smaller. It is sufficient (with respect to adding and removing states) to consider an PT_f of depth $d = 2^n$ since each node in PT_f corresponds to (i) an element $u \in U$ where $T(u)$ is unique for each u and also (ii)to a subset of states reachable at depth

$|u|$ in C_m. Note that a node may be removed from PT only if an outgoing cluster of one of its ancestor nodes can be partitioned. Now since `LearnCE_1` always partitions some cluster in the prefix tree into smaller ones, this can happen only k number of times for the nodes at a given depth in PT_f until each transition corresponds to a single alphabet symbol. Using induction on depth of PT_f, it follows that the clusters at all nodes in PT_f will be fully partitioned in at most $k \cdot 2^n$ iterations. Therefore, the algorithm will make at most $(k \cdot 2^n)$ calls to `LearnCE` (or candidate queries) before terminating.

6.4.3 Optimizing l^*

Although the complexity is bad (mainly due to the reason that l^* may introduce a state corresponding to each subset of states reachable at a given depth in C_m), our experimental results show that the algorithm is effective in computing small size assumptions on real-life examples. Moreover, in context of AGR, we seldom need to learn C_m completely; often, an approximation obtained at an intermediate learning step is sufficient. We now propose several optimizations to the basic l^* algorithm outlined above.

Coarser Cluster partitioning using *ce* generalization. Recall that $ce = u_i \cdot a \cdot v_{i+1}$ where a is a label on $X \cup X'$. Let $u_r = [u_i]^r$. Cluster partitioning occurs in the `LearnCE_1` procedure where $[\![u_r \cdot a \cdot v_i]\!] = 1$ and $[\![u_r \cdot \phi \cdot v_i]\!] = 0$. The `PartitionTable` procedure uses the symbol a (called the *refining predicate*) to partition the cluster ϕ in $follow(u_r)$ into ϕ_1 and ϕ_2. Since a is an alphabet symbol, this leads to a fine-grained partitioning of $follow(u_r)$. Moreover, note that multiple *similar* counterexamples may cause repeated partitioning of $follow(u_r)$, which may lead to explicit enumeration of Σ in the worst case. For example, there may be several positive counterexamples of form $u_i \cdot a' \cdot v_{i+1}$ where $a' \in \phi$ and a' differs from a only in a few variable assignments. Therefore, we propose a SAT-based technique that performs a *coarser* partitioning of ϕ by first *enlarging* the refining predicate a to a new predicate, say, A, and then using A to partition ϕ.

Recall that the value of $[\![u_r \cdot a \cdot v_{i+1}]\!]$ is computed using a BMC instance. Given a predicate p over $X \cup X'$, let $E(p)$ represent the BMC formula corresponding to the evaluating $[\![u_r \cdot p \cdot v_{i+1}]\!]$. We know that the formula $E(a)$ is UNSAT while $E(\phi)$ is SAT (cf. Section 6.3.1). We say that a predicate A is an enlargement of a if $a \Rightarrow A$. We are interested in computing the maximum enlargement A of a so that $E(A)$ is UNSAT. This is equivalent to solving an All-SAT problem [113] and is computationally expensive with SAT. Instead, we propose a greedy approach to compute a *maximal* enlargement of a by using a variable lifting technique [116] in the following way. Since a may be viewed as a conjunction of variable assignment constraints, we iteratively remove these variable assignments to obtain larger enlargements A as long as the formula $E(A)$ remains UNSAT. The procedure `Enlarge` shows the pseudocode for this technique. It can be implemented efficiently using an incremental SAT solver and made efficient by observing the UNSAT core obtained at each iteration [121].

`Enlarge` (E, a)

$A = a$

// A is a set of constraints of form ($x_i = d_i$)

Loop:

Pick a *new* constraint $x_i = d_i$ in A; If impossible, return A

$A := A \setminus \{(x_i = d_i)\}$

if ($E(A)$ is SAT)

$A := A \cup \{(x_i = d_i)\}$

Lemma 23 *The procedure* `Enlarge` *finds a maximal enlargement A_m of a when it terminates. Also, A_m must partition the cluster ϕ into two disjoint clusters.*

Proof. Note that $E(p)$ can be written as $F \wedge p$ for some formula F. We know that $E(a)$ is UNSAT and $E(\phi)$ is SAT. `Enlarge` must terminate with at least one constraint in A, since $E(true)$ is SAT. ($E(true) = F \wedge true = F \wedge (\phi \vee \neg\phi) = E(\phi) \vee f'$ for some formula

f'). It is clear from the pseudocode that A_m computed on termination is maximal.

Since $a \Rightarrow \phi$ and $a \Rightarrow A_m$, so $\phi \wedge A_m \neq false$. Hence A_m must split ϕ into two disjoint clusters $\phi_1 = \phi \wedge A_m$ and $\phi_2 = \phi \wedge \neg A_m$.

Follow transfer on partitioning. Recall that `PartitionTable` procedure partitions the cluster ϕ in follow set of u_r into ϕ_1 and ϕ_2 and removes all extensions $Uext$ of u_r. However, this may lead to loss of information about follow sets of elements of $Uext$ which may be useful later. We therefore copy the follow set information for each $u \in Uext$ ($u = u_r \cdot \phi \cdot v$ for some v) to the corresponding partitioned traces, $u_r \cdot \phi_1 \cdot v$ and $u_r \cdot \phi_2 \cdot v$.

Reusing $v \in V$. In the `LearnCE_1` algorithm, it is possible that $v_{i+1} \notin V$. Instead of eagerly adding v_{i+1} to set V, we check if some $v \in V$ can act as a substitute for v_{i+1}, i.e., $\alpha_i = 1$ and $\alpha_{i+1} = 0$ with v substituted for v_{i+1}. If we find such v, we use the other case in `LearnCE_1` which performs cluster partitioning. Intuitively, adding an element to V may cause unnecessary state partitions corresponding to other elements in U, while reusing a previous element in V will lead to a cluster partition whose effect will be local to $u_i \cdot \phi$ and its successors.

Membership Cache and Counterexample History The results of all membership queries are stored in a *membership cache* so that multiple queries on the same trace are not brought to the Teacher each time, but instead looked up in the cache. We also keep a *counterexample history* set, which stores all the counterexamples provided by the Teacher. Before making a candidate query, we check that the new hypothesis agrees with the unknown language on all the previous counterexamples in the counterexample history set. This is useful because of the lazy nature of the algorithm: it may become necessary to learn again from an older counterexample since the previous learning step only extracted partial information from it.

6.4.4 Another Lazy Learning Algorithm: l_r^*

An algorithm for learning parameterized systems was presented in [19]. In such systems, the alphabet is *parameterized*, i.e., it consists of a small set of basis symbols, each of which is parameterized by a set of boolean variables. Note that the alphabet in CFAs can be viewed as being parameterized by the set of support variables (with a single basis alphabet, that may be omitted). Although the above algorithm can be adapted to learn CFAs, it is not efficient in practice because the counterexample analysis is inefficient and it may also enumerate the exponential alphabet set in the worst case. In this section, we first reformulate the algorithm (called l_r^*) based on the generalized L^* presented in Section 6.4.1. We then describe the details of the learning procedure `LearnCE` for l_r^*. We compare l^* (Lazy-AGR) with l_r^* (P-AGR) in the next section.

Recall that the generalized L^* algorithm maintains a follow set of alphabet symbols for each $u \in \Sigma^*$. In the original L^* algorithm, the follow set equals Σ for each u, which ensures that the obtained DFA is complete. In contrast, the l_r^* algorithm only keeps a set of representative alphabet symbols in the follow set for each u. Each of these symbols in mapped to an alphabet cluster by a *cluster mapping* function *cmap*. Each $u \in U$ and $a \in follow(u)$ is mapped to the cluster $cmap(u, a)$. In other words, each element $a \in follow(u)$ represents a different outgoing transition cluster $cmap(u, a)$ from the state $[u]$. The procedure `MkDFA` obtains the transition relation as follows: $\delta([u], cmap(u, a)) = [u \cdot a]$ for $a \in follow(u)$. Note that if $\cup_{a \in follow(u)}(cmap(u, a)) = \Sigma$, then the obtained deterministic CFA will be complete. Note that the elements of the follow set in l^* are alphabet clusters while in l_r^*, they are individual alphabet symbols.

The l_r^* algorithm learns from a given counterexample (procedure `LearnCE`) by partitioning a state in the current hypothesis either by adding a distinguishing suffix to the suffix set V or by adding a new representative element to the follow set for some $u \in U$. In the latter case, the function *cmap* is recomputed for u and leads to a cluster partition and

may also lead to addition of a new state. Figure 6.4 shows the pseudocode for the `LearnCE` procedure in the l_r^* algorithm. The procedure finds the representative follow set element f for the mis-classified transition label o_i in the counterexample. Then, it checks if f and o_i belong to the same cluster class by asking membership queries. If they don't, then a distinguishing trace v_{i+1} is added to V and the observation table is updated. Otherwise, o_i has been wrongly classified to the cluster containing f by the cluster mapping function *cmap*. Therefore, o_i is added to the follow set for u_r (that also contains f) and the corresponding *cmap* function is updated. In order to avoid enumerating the alphabet set as the result of the partitioning, we can use the counterexample generalization optimization to enlarge o_i before partitioning, in a way similar to that presented for the l^* algorithm.

`LearnCE`(ce)
Init: $\forall u \in \Sigma^*$, set $follow(u) = \{a\}$, for some fixed $a \in \Sigma$

Find i so that $\alpha_i \neq \alpha_{i+1}$
Let $ce = u_i \cdot o_i \cdot v_{i+1}$
Let $u_r = [u_i]^r$, $q = [u_i]$ and $q' = [u_i \cdot o_i]$
Suppose $R_C(q, \phi, q')$ and $o_i \in \phi$
Obtain f so that $cmap(u_r, f) = \phi$
Let $b_1 = [\![u_r \cdot o_i \cdot v_{i+1}]\!]$
Let $b_2 = [\![u_r \cdot f \cdot v_{i+1}]\!]$

if ($b_1 \neq b_2$) // o_i and f can be distinguished by v_{i+1}, add v_{i+1} to V
  $V := V \cup \{v_{i+1}\}$
  For all $u \in U \cup UA$: `Fill`($\mathtt{u}, \mathtt{v_{i+1}}$)
else // update $follow(u_r)$, partition ϕ cluster
  $\phi_1 := \phi \wedge o_i$, $\phi_2 := \phi \wedge \neg o_i$
  $follow(u_r) := follow(u_r) \cup \{o_i\}$
  $cmap(u_r, o_i) := \phi_1$, $cmap(u_r, f) = \phi_2$
  For all $v \in V$: `Fill`($\mathtt{u_r \cdot o_i}, \mathtt{v}$)

Figure 6.4: Pseudocode for the `LearnCE` procedure in the l_r^* algorithm

6.5 Implementation and Experiments

We have implemented our SAT-based AGR approach based on **NC** and **C** rules in a tool called SYMODA, written in C++. The l^* algorithm is implemented together with related optimizations. The input language of the tool is a simple intermediate language (SIL), which allows specification of a set of concurrent modules which execute synchronously. Each module may have its internal variables and communicates with other modules using global variables. Variables are of boolean, enumerated and bit-vector types and sequential logic are specified in terms of control flow based on guarded commands. Each module may also have a block of combinational logic in terms of boolean equations. In order to evaluate our approach, we translate both SMV and Verilog programs into SIL. Translator from Verilog to SIL is implemented using the ICARUS Verilog parser. Translation from SMV is done using a python script. The encoding of programs into formula is done as follows. We translate enumerated types to integers with bound constraints. Bit-vector variables are "bit-blasted" currently. We check the correctness of the translation by monolithic SAT-based model checking on the translated models. We use the incremental SMT solver YICES [3, 55] as the main decision procedure. Interpolants are obtained using the library interface to the FOCI tool [1]. We represent states of a CFA explicitly while BDDs are used to represent transitions compactly and avoid redundancy.

Experiments. All experiments were performed on a 1.4GHz AMD machine with 3GB of memory running Linux. Table 6.5 compares three algorithms for automated AGR: a BDD-based approach [107, 115] (BDD-AGR), our SAT-based approach using l^* (Lazy-AGR) and (P-AGR), which uses a learning algorithm for parameterized systems [19]. The last algorithm was not presented in context of AGR earlier; we have implemented it using a SAT-based Teacher and other optimizations for comparison purposes. The BDD-AGR approach automatically partitions the given model before learning assumptions while we manually assign each top-level module to a different partition. Benchmarks *s1a*, *s1b*,

guidance, *msi* and *syncarb* are derived from the NuSMV tool set and used in the previous BDD-based approach [107] while *peterson* and *CC* are obtained from the VIS and Texas97 benchmark sets [2]. All examples except *guidance* and *CC* can be proved using monolithic SAT-based UMC in small amount of time. Note that in some of these benchmarks, the size of the assumption alphabet is too large to be even enumerated in a short amount of time.

The SAT-based Lazy-AGR approach performs better than the BDD-based approach on $s1a$ and $s2a$ (cf. Table 6.5); although they are difficult for BDD-based model checking [115], SAT-based UMC quickly verifies them. On the *msi* example, the Lazy-AGR approach scales more uniformly compared to BDD-AGR. BDD-AGR is able to compute an assumption with 67 states on the *syncarb* benchmark while our SAT-based approaches with interpolation timeout with assumption sizes of around 30. The bottleneck is SAT-based UMC in the candidate query checks; the k-induction approach keeps unfolding transition relations to increasing depths while the interpolants are either large or take too much time to compute. On the *peterson* benchmark, BDD-AGR finishes earlier but with larger assumptions of size up to 34 (for two partitions) and 13 (for four partitions). In contrast, Lazy-AGR computes assumptions of size up to 6 while P-AGR computes assumptions of size up to 8. This shows that it is possible to generate much smaller assumptions using the lazy approach as compared to the eager BDD-based approach. Both the *guidance* and *syncarb* examples require interpolation-based UMC and timeout inside a candidate query with the k-induction based approach. P-AGR timeouts in many cases where Lazy-AGR finishes since the former performs state partitions more eagerly and introduces unnecessary states in the assumptions.

Example	TV	GV	T/F	BDD-AGR				P-AGR				Lazy-AGR			
				NC		**C**		**NC**		**C**		**NC**		**C**	
				#A	Time	#A	Time	#A	Time	#A	Time	#A	Time	#A	Time
s1a	86	5	T	2	754	2	223	3	3	3	3	3	3.5	3	1.3
s1b	94	5	T	2	TO	2	1527	3	3.3	3	3.3	3	3.9	3	2
guidance	122	22	T	2	196	2	6.6	1	31.5^i	5	146^i	1	40^i	3	55^i
msi(3)	57	22	T	2	2.1	2	0.3	1	8	*	TO	1	8	3	17
msi(5)	70	25	T	2	1183	2	32	1	16	*	TO	1	15	3	43
syncarb	21	15	T	-	-	67	30	*	TO^i	*	TO^i	*	TO^i	*	TO^i
peterson	13	7	T	-	-	34	2	6	53^i	8	210^i	6	13	6	88^i
CC(2a)	78	30	T	-	-	-	-	1	8	*	TO	1	8	4	26
CC(3a)	115	44	T	-	-	-	-	1	8	*	TO	1	7	4	20
$CC(2b)^i$	78	30	T	-	-	-	-	*	TO	*	TO	10	1878	5	87
$CC(3b)^i$	115	44	T	-	-	-	-	*	TO	*	TO	6	2037	11	2143

Table 6.1: Comparison of BDD-based and Lazy AGR schemes. P-AGR uses a learning algorithm for parameterized systems [19] while Lazy-AGR uses l^*. TV and GV represent the number of total and global boolean variables respectively. All times are in seconds. TO denotes a timeout of 3600 seconds.#A denotes states of the largest assumption. '-' denotes that data could not be obtained due to the lack of tool support (The tool does not support the **NC** rule or Verilog programs as input). The superscript ^{i} denotes that interpolant-based UMC was used.

Example	T/F	with CE Gen	w/o CE Gen
s1a	T	1.3	1.1
s1b	T	2	1.87
s2a	F	26	TO
s2b	T	36	TO
msi(5)	T	43	86
guidance	T	55	57
Peterson	T	13	175
CC(3b)	T	2143	TO

Table 6.2: Effect of the counterexample generalization optimization on the l^* algorithm.

6.6 Conclusions and Related Work

We have presented a new SAT-based approach to automated AGR for shared memory systems based on lazy learning of assumptions: alphabet explosion during learning is avoided by representing alphabet clusters symbolically and performing on-demand cluster partitioning during learning. Experimental results demonstrate the effectiveness of our approach on hardware benchmarks. Since we employ an off-the-shelf SMT solver, we can directly leverage future improvements in SAT/SMT technology. Future work includes investigating techniques to exploit incremental SAT solving for answering queries for a particular AGR premise, e.g., since we need to check $M \parallel A \vDash P$ repeatedly for many different assumptions A, we could add and remove constraints corresponding to A at each iteration while retaining the rest of the constraints corresponding to M and P. Finally, the problem of finding good system decompositions for allowing small assumptions needs to be investigated. Although presented for the case of finite-state systems, our technique can be extended to infinite-state systems, where the weakest assumption has a finite bisimulation quotient. It can also be applied to compositional verification of concurrent software by first obtaining a finite state abstraction based on a set of predicate variables and then learning assumptions based on these predicate variables. We also plan to use interpolants to improve coarse cluster partitioning.

SAT-based bounded model checking for LTL properties was proposed by Biere et al. [21] and several improvements, including techniques for making it complete have been proposed [9, 113]. All the previous approaches are non-compositional, i.e., they build a monolithic transition relation for the whole system. To the best of our knowledge, our work in the first to address automated compositional verification in the setting of SAT-based model checking.

The symbolic BDD-based AGR approach [115] for shared memory systems and its extension using automated system decomposition [107] is closely related to ours. The

technique uses a BDD-based model checker and avoids alphabet explosion by using eager state-partitioning to introduce all possible new states in the next assumption, and by computing the transition relation (edges) using BDD-based quantifier elimination. In contrast, we use a SAT-based model checker and our lazy learning approach does not require a quantifier elimination step, which is expensive with SAT. Moreover, due to its eager state-partitioning, the BDD-based approach may introduce unnecessary states in the assumptions.

Two approaches for improved learning based on alphabet under-approximation and iterative enlargement [33, 63] have been proposed. Our lazy approach is complementary: while the above techniques try to reduce the overall alphabet by under-approximation, our technique tries to compactly represent a large alphabet set symbolically and performs localized partitioning. In cases where a small alphabet set is not sufficient, the previous techniques may not be effective. We also note that both the above approaches can be combined with our approach by removing assumption variables during learning and adding them back iteratively. A learning algorithm for parameterized systems (alphabet consists of a small set of basis symbols, each of which is parameterized by a set of boolean variables) was proposed in [19]. Our lazy learning algorithm is different: we reason about a set of traces directly using a SAT-based model checker and perform more efficient counterexample analysis by differentiating positive and negative counterexamples (cf. Section 6.4).

Similar to a lazy approach for CEGAR [80], the lazy learning algorithm localizes the cluster partitioning to the follow set of a particular state and adds only a single cluster to the follow sets at each iteration.

Chapter 7

Checking Deadlock Compositionally

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. In this chapter, 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 Teacher. We show how L^F can be used for compositional regular failure language containment, and in particular, 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.

7.1 Problem Formulation and Contributions

Recall that 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 [44] (cf. Chapter 3), weakest assumptions are

naturally represented as deterministic finite automata, and this leads to the use of the L^* learning algorithm. Similarly, in the case of simulation (cf. Chapter 5), 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* [82], 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. The failure automata can be viewed as an instance of lattice automata [90], where the state labels are drawn from a lattice. 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 **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: (i) $L_1 \parallel L_W \subseteq L_S$ and (ii) 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 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 **NC** and L^F. We also define a circular AG proof rule **C** 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 [29] reasoning framework. We present encouraging results on several non-trivial benchmarks, including an embedded OS, and Linux device drivers.

7.2 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 [118] definitions.

Definition 29 (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 $t_1 \cdot t_2$. For any LTS $M = (S, Init, \Sigma, \delta)$, we define the function $\widehat{\delta} : 2^S \times \Sigma^* \rightarrow 2^S$ as follows: $\widehat{\delta}(X, \epsilon) = X$ and $\widehat{\delta}(X, t \cdot \alpha) = \{s' | \exists s \in \widehat{\delta}(X, t) \mathbf{.} s \xrightarrow{\alpha} s'\}$. M is said to be deterministic if $|Init| = 1$ and $\forall s \in S \mathbf{.} \forall \alpha \in \Sigma \mathbf{.} |\widehat{\delta}(\{s\}, \alpha)| \leq 1$, and complete if $\forall s \in S \mathbf{.} \forall \alpha \in \Sigma \mathbf{.} |\widehat{\delta}(\{s\}, \alpha)| \geq 1$. Thus if M is both deterministic and complete then $|Init| = 1$ and $\forall s \in S \mathbf{.} \forall t \in \Sigma^* \mathbf{.} |\widehat{\delta}(\{s\}, t)| = 1$. In such a case, we write $\widehat{\delta}(s, t)$ to mean the only element of $\widehat{\delta}(\{s\}, t)$.

Definition 30 (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 31 (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 $\forall s' \in S \mathbf{.} (s, \alpha, s') \notin \delta$. 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) $\forall s \mathbf{.} Ref(s, \emptyset)$, and (ii) $\forall s, R, R' \mathbf{.} Ref(s, R) \wedge R' \subseteq R \implies Ref(s, R')$, i.e., refusals are* downward-closed.

Definition 32 (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 33 (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: $Max(P) = \{R \in P \mid \forall R' \in P \,.\, R \not\subset R'\}$

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 34 (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 $\mu : F \to 2^{\widehat{\Sigma}}$ 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). Fig. 7.1(a) shows an LTS over Σ=$\{a,b,c\}$. Fig. 7.1(b) and (c) show the corresponding FLA and its deterministic

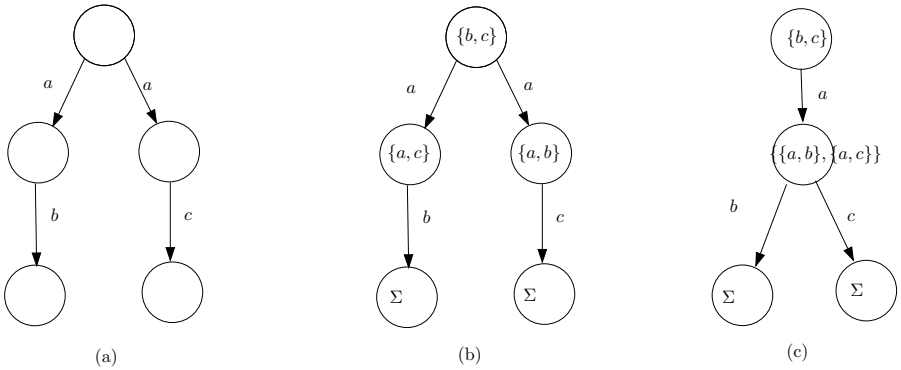


Figure 7.1: (a) LTS M on $\Sigma = \{a, b, c\}$, (b) its FLA (c) its deterministic FLA. All states of FLAs are accepting.

version, respectively.

Definition 35 (Language of a FLA) *Let $A = (M, F, \mu)$ be a FLA such that $M = (S, Init, \Sigma, \delta)$. Then a failure (t, R) is accepted by A iff $\exists s \in F \,.\, \exists R' \in \mu(s) \,.\, s \in \widehat{\delta}(Init, t) \wedge R \subseteq R'$. The language of A, denoted by $\mathbb{L}(A)$, is the set of all failures accepted by A.*

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

Lemma 24 *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 1 *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 by $\Psi(s')$ the set of states of M from which s' was derived by the subset construction. To define μ' 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

$\mathbb{L}(A') = L$, *consider any failure* (t, R). *Then:*

$$(t, R) \in \mathbb{L}(A')$$

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

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

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

$$\iff (t, R) \in \mathbb{L}(A) = L$$

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[1]. 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 25 *Every regular failure language(RFL) is accepted by a unique (up to isomorphism) minimal deterministic finite failure automaton.*

Proof 2 *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* Σ^* *as follows:*

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

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)* $\forall u \in \Sigma^* \,.\, \forall \alpha \in \Sigma \,.\, [u] \xrightarrow{\alpha} [u \cdot \alpha]$, *and (iv)* $F = \{[u] \mid \exists R \in \widehat{\Sigma} \,.\, (u, R) \in L\}$.

[1]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.

Also, let us define a function μ 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, μ).

We first show, by contradiction, that A is deterministic. First, note that $|Init| = 1$. Next, suppose that A is non-deterministic. Then there exists two traces $u \in \Sigma^*$ and $v \in \Sigma^*$ and an action $\alpha \in \Sigma$ such that $u \equiv v$ but $u \cdot \alpha \not\equiv v \cdot \alpha$. Then there exists a failure (t, R) such that $(u \cdot \alpha \cdot t, R) \in L \iff (v \cdot \alpha \cdot t, R) \notin L$. But then there exists a failure $(t', R) = (\alpha \cdot t, R)$ such that $(u \cdot t', R) \in L \iff (v \cdot t', R) \notin L$. 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 $t = t' \cdot \alpha$ for some trace t' and action α. By the inductive hypothesis, $\widehat{\delta}(Init, t') = [t']$. Also, from the definition of A we know that $[t'] \xrightarrow{\alpha} [t' \cdot \alpha]$. Hence $\widehat{\delta}(Init, t) = \widehat{\delta}(Init, t' \cdot \alpha) = [t' \cdot \alpha] = [t]$. This completes the proof.*

Now, consider any DFLA $A' = (M', F', \mu')$ where $M' = (S', Init', \Sigma, \delta')$ such that $\mathbb{L}(A') = L$. 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 Ω 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 $(u \cdot t, R)$ iff it also accepts $(v \cdot t, R)$. Since A' accepts L, we find that $u \equiv v$. Combining this 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 Ω is well-defined. In addition, Ω 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 $\mathbb{L}(A) = L$ we observe that for any trace t and any refusal R, $(t, R) \in L \iff [t] \in F \wedge \exists R' \in \mu([t]) \,.\, R \subseteq R' \iff (t, R) \in \mathbb{L}(A)$.

Next, recall that Ω 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, Ω must be a bijection. We show that Ω is also an isomorphism.

Let us write Ω^{-1} to mean the inverse of Ω. Note that Ω^{-1} is also a bijection, and more specifically, $\forall t \in \Sigma^* \centerdot \Omega^{-1}([t]) = \Omega^{-1}(\widehat{\delta}(Init, t)) = \widehat{\delta}(Init', t)$. We will now prove the following statements: **(C2)** $\Omega^{-1}(Init) = Init'$, **(C3)** $\forall u \in \Sigma^* \centerdot \forall v \in \Sigma^* \centerdot \forall \alpha \in \Sigma \centerdot [u] \xrightarrow{\alpha} [v] \iff \Omega^{-1}([u]) \xrightarrow{\alpha} \Omega^{-1}([v])$, **(C4)** $\forall s \in S \centerdot s \in F \iff \Omega^{-1}(s) \in F'$, *and* **(C5)** $\forall s \in F \centerdot \mu(s) = \mu'(\Omega^{-1}(s))$.

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 $[v] = \widehat{\delta}(Init, u \cdot \alpha)$. Hence $\Omega^{-1}([u]) = \widehat{\delta}(Init', u)$ and $\Omega^{-1}([v]) = \widehat{\delta}(Init', u \cdot \alpha)$. 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 $\Omega^{-1}([v]) = \widehat{\delta}(Init', u \cdot \alpha)$. Therefore, $[u] = \widehat{\delta}(Init, u)$ and $[v] = \widehat{\delta}(Init, u \cdot \alpha)$, 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 $s \in F \iff [u] \in F \iff \exists R \centerdot (u, R) \in L \iff \widehat{\delta}(Init', u) \in F' \iff \Omega^{-1}(s) \in F'$. 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 $\exists R' \in \mu(s) \centerdot R \subseteq R'$ and $\forall R' \in \mu'(\Omega^{-1}(s)) \centerdot R \not\subseteq R'$. Now we also know that $s = \widehat{\delta}(Init, u)$ and $\Omega^{-1}(s) = \widehat{\delta}(Init', u)$. Therefore $(u, R) \in \mathbb{L}(A) \setminus \mathbb{L}(A')$, which implies that $\mathbb{L}(A) = L \neq L = \mathbb{L}(A')$, a contradiction.*

Note that for any LTS M, $\mathcal{F}(M)$ is regular[2]. Indeed, the failure automaton corresponding to $M = (S, Init, \Sigma, \delta)$ is $A = (M, S, \mu)$ such that $\forall s \in S . \mu(s) = Max(\{R \mid Ref(s, R)\})$.

[2]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.

7.3 Assume-Guarantee Reasoning for Deadlock

We now present an assume-guarantee (cf. Chapter 3) proof rule for deadlock detection for systems composed of two components. We use the notion of parallel composition proposed in the theory of CSP [82] and define it formally.

Definition 36 (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:* $\forall i \in \{1, 2\} \bullet (\alpha \in \Sigma_i \wedge (s_i, \alpha, s_i') \in \delta_i) \vee (\alpha \notin \Sigma_i \wedge s_i = s_i')$.

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 [42] 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.

Definition 37 (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 26 *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 3 *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 37 we have* $(t, R) \in (L_1' \parallel L_2')$, *which completes the proof.*

Definition 38 (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* $A_1 \amalg A_2$[3], *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 $\|$ 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.

Lemma 27 $\mathcal{F}(M_1 \amalg M_2) = \mathcal{F}(M_1) \parallel \mathcal{F}(M_2)$.

Proof 4 *For any LTSs* M_1 *and* M_2 *over the same alphabet* Σ, *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 37.

Lemma 28 $\mathbb{L}(A_1 \amalg A_2) = \mathbb{L}(A_1) \parallel \mathbb{L}(A_2)$.

Proof 5 *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* $\mathbb{L}(A_1 \amalg A_2)$. *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 μ *we find that:*

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

[3]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.

Therefore, $(t, R_1) \in \mathbb{L}(A_1)$, $(t, R_2) \in \mathbb{L}(A_2)$, *and* $(t, R) \in \mathbb{L}(A_1) \parallel \mathbb{L}(A_2)$. *This proves that* $\mathbb{L}(A_1 \amalg A_2) \subseteq \mathbb{L}(A_1) \parallel \mathbb{L}(A_2)$. *Now let* (t, R) *be any element of* $\mathbb{L}(A_1) \parallel \mathbb{L}(A_2)$. *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* $(t, R) \in \mathbb{L}(A_1 \amalg A_2)$. *This show that* $\mathbb{L}(A_1) \parallel \mathbb{L}(A_2) \subseteq \mathbb{L}(A_1 \amalg A_2)$ *and completes the proof.*

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 29 *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* $\mathbb{L}(A_1) \subseteq \mathbb{L}(A_2)$ *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 6 *First, we note that* $\mathbb{L}(A_1) = \mathbb{L}(A_1')$ *and* $\mathbb{L}(A_2) = \mathbb{L}(A_2')$. *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 $\forall R_2 \in \mu_2(s_2) \,.\, R_1 \not\subseteq R_2$. Now it follows that $(t, R) \in \mathbb{L}(A_1') \setminus \mathbb{L}(A_2')$. Hence $\mathbb{L}(A_1') \not\subseteq \mathbb{L}(A_2')$ and therefore $\mathbb{L}(A_1) \not\subseteq \mathbb{L}(A_2)$.

For the reverse implication we also prove the contrapositive. Suppose $\mathbb{L}(A_1) \not\subseteq \mathbb{L}(A_2)$ and let (t, R) be any element of $\mathbb{L}(A_1) \setminus \mathbb{L}(A_2) = \mathbb{L}(A_1') \setminus \mathbb{L}(A_2')$. Let $s_1 = \widehat{\delta}(Init_1, t)$ and $s_2 = \widehat{\delta}(Init_2, t)$. But then we know that $\exists R_1 \in \mu_1(s_1) \,.\, R \subseteq R_1$ and either $s_2 \notin F_2$ or $\forall R_2 \in \mu_2(s_2) \,.\, R \not\subseteq R_2$. However, this implies that $s_1 \in F_1$ and either $s_2 \notin F_2$ or $\exists R_1 \in \mu_1(s_1) \,.\, \forall R_2 \in \mu_2(s_2) \,.\, R_1 \not\subseteq R_2$. 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 $\mathbb{L}(A_1) \subseteq \mathbb{L}(A_2)$ by determinizing A_1 and A_2, constructing the *product* of the underlying LTSs and checking if the condition in Lemma 29 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 $\mathbb{L}(A_1)$ is obtained by composing two RFLs L_1 and L_2, i.e., $\mathbb{L}(A_1) = L_1 \parallel L_2$ and let $\mathbb{L}(A_2) = L_S$, the specification language. In order to check RFLC between $L_1 \parallel L_2$ and L_S, the approach presented in lemma 29 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 II M_2 is deadlock free iff $\mathcal{F}(M_1 \text{ II } M_2) \subseteq \overline{L_{Dlk}}$. Let $L_1 = \mathcal{F}(M_1)$, $L_2 = \mathcal{F}(M_2)$ and $L_S = \overline{L_{Dlk}}$. Using Lemma 27, 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) $\delta = \{s \xrightarrow{\alpha} s \mid \alpha \in \Sigma\}$, (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 II A_2 (cf. Lemma 38) and then check if $\mathbb{L}(A_1 \text{ II } A_2) \subseteq \mathbb{L}(A_S)$ (cf. Lemma 28 and 29). 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.

7.3.1 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, which we call **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 7 *The completeness of* **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 26, we have $L_1 \parallel L_2 \subseteq L_1 \parallel L_A$. Combining this with the first premise we get $L_1 \parallel L_2 \subseteq L_S$ which is the desired conclusion.*

In principle, **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 **NC** does not hold, then **NC** does not help in obtaining a counterexample to $L_1 \parallel L_2 \subseteq L_S$.

In this chapter, 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 **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 **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.

7.3.2 Weakest Assumption

Consider the proof rule **NC**. For any L_1 and L_S, let $\widehat{L}$ be the set of all languages that can discharge the first premise of **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 10 *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 8 *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 26. 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 a FLA* A_W *such that* $\mathbb{L}(A_W) = L_W$. *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* $A_W = (M_1 \amalg M_S, F_W, \mu_W)$. *In order to define the set of final states* F_W *and the labeling function* μ_W *of* A_W *we define the extended labeling function* $\widehat{\mu} : S \rightarrow 2^{\widehat{\Sigma}}$ *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:*

$$\widehat{\mu}(s_1, s_S) =$$

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

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* μ_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* $\mathbb{L}(A_W) = L_W$. *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 $[(t, R) \in \mathbb{L}(A_W)]$**..** *Then we know that* $R \in \widehat{\mu}(s_1, s_S)$*. Now consider the language* $L = L_1 \parallel \{(t, R)\}$*. By Definition 37, 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* $\exists R_S \in \widehat{\mu}(s_S) \,.\, (R_1 \cup R) \subseteq R_S$*. 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 sub-case.*

Case 2 $[(t, R) \notin \mathbb{L}(A_W)]$**..** *In this case* $R \notin \widehat{\mu}(s_1, s_S)$*. Then, from the definition of* $\widehat{\mu}$ *above we have* $\exists R_1 \in \widehat{\mu}(s_1) \,.\, \forall R_S \in \widehat{\mu}(s_S) \,.\, (R_1 \cup R) \not\subseteq R_S$*. Now consider the language* $L = L_1 \parallel \{(t, R)\}$*. By Definition 37,* $(t, R_1 \cup R) \in L$*. But from* $\forall R_S \in \widehat{\mu}(s_S) . (R_1 \cup R) \not\subseteq R_S$*, 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 sub-case 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 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 Teacher and learning from them. In the next section, we present L^F. Subsequently, in Section 7.5, 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 7.5 at this point.

7.4 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 Teacher that can answer two kinds of queries regarding U: **(1) Membership query:** Given a failure e the Teacher returns TRUE if $e \in U$ and FALSE otherwise. **(2) Candidate query:** Given a deterministic FLA C, the Teacher returns TRUE if $\mathbb{L}(C) = U$. Otherwise it returns FALSE along with a

counterexample failure $CE \in (\mathbb{L}(C) \setminus U) \bigcup (U \setminus \mathbb{L}(C))$.

Observation Table.. L^F uses an observation table to record the information it obtains by querying the Teacher. 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 $\widehat{\mathbb{T}} = \mathbb{T} \cup (\mathbb{T} \cdot \Sigma)$.

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

An observation table $\mathcal{T} = (\mathbb{T}, \mathbb{E}, \mathbb{R})$ is said to be well-formed iff: $\forall t_1 \in \mathbb{T} \centerdot \forall t_2 \in \mathbb{T} \centerdot t_1 \neq t_2 \implies \mathbb{R}(t_1) \neq \mathbb{R}(t_2)$. 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 30 *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 9 *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, $(t_1 \cdot t, p) \in U$ iff $(t_2 \cdot t, p) \notin U$. This is impossible since A would reach the same state on inputs $t_1 \cdot t$ and $t_2 \cdot t$.*

Closed observation table.. An observation table $\mathcal{T} = (\mathbb{T}, \mathbb{E}, \mathbb{R})$ is said to be *closed* iff it satisfies the following: $\forall t \in \mathbb{T} \centerdot \forall \alpha \in \Sigma \centerdot \exists t' \in \mathbb{T} \centerdot \mathbb{R}(t \cdot \alpha) = \mathbb{R}(t')$. Intuitively, this means that if we extend any trace $t \in \mathbb{T}$ by any action α then the result is indistinguishable from

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

Figure 7.2: Algorithm **MakeClosed** extends an input well-formed table $\mathcal{T}$ so that the resulting table is both well-formed and closed.

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 7.2. Observe that at every step of **MakeClosed**, the table $\mathcal{T}$ remains well-formed and hence, by Lemma 30, 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 $\mathbb{E}$.

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) $\delta = \{t \xrightarrow{\alpha} t' \mid \mathbb{R}(t \cdot \alpha) = \mathbb{R}(t')\}$, (iv) $F = \{t \mid \exists e = (\epsilon, R) \in \mathbb{E} \centerdot \mathbb{R}(t, e) = 1\}$, 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, $CE \in \mathbb{L}(C) \iff CE \notin U$. The algorithm **ExpGen** adds a single new failure to $\mathcal{T}$ as follows. Let $t = \alpha_1 \cdot \ldots \cdot \alpha_k$. 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 $t_i \cdot t^i = t$.

Additionally, for $0 \leq i \leq k$, let s_i be the state of C reached by executing t_i. In other words, $s_i = \widehat{\delta}(t_i)$. 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 $(s_i \cdot t^i, R) \in U$ and 0 otherwise. Note that we can compute b_i by evaluating s_i and then making a membership query with $(s_i \cdot t^i, R)$. In particular, $s_0 = \epsilon$, and hence $b_0 = 1$ if $CE \in U$ and 0 otherwise. We now consider two cases.

Case 1: [$b_0 = 0$] This means $CE \notin U$ and hence $CE \in \mathbb{L}(C)$. Recall that $CE = (t, R)$ and $t = \alpha_1 \cdot \ldots \cdot \alpha_k$. Consider the state $s_k = \widehat{\delta}(t)$ as described above. Since $CE \in \mathbb{L}(C)$ we know that $s_k \in F$ and $\exists R' \in \mu(s_k) \centerdot R \subseteq R'$.

Also, since C was constructed (cf. Section 7.4) 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 (t^{j+1}, R) to $\mathbb{E}$.

We now show that the failure $e = (t^{j+1}, R)$ has a special property.

Since C contained a transition $s_j \xrightarrow{\alpha_{j+1}} s_{j+1}$, it must be the case that $\mathbb{R}(s_j \cdot \alpha_{j+1}) =$

$\mathbb{R}(s_{j+1})$. However, $\mathbb{R}(s_j \cdot \alpha_{j+1}, e) = b_j \neq b_{j+1} = \mathbb{R}(s_{j+1}, e)$. 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: [$b_0 = 1$] This means $CE \in U$ and hence $CE \notin \mathbb{L}(C)$. We consider two sub-cases. 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 (t^{j+1}, R) 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 $CE \notin \mathbb{L}(C)$ we know that either s_k is not a final state of C or $\forall R' \in \mu(s_k) . R \not\subseteq R'$. 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 (ϵ, R_{max}) to $\mathbb{E}$.

The addition of (ϵ, 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 hence the number of 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, viz., R_{max}.

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

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 Teacher 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 Teacher 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.

7.5 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 Teacher 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 $\mathbb{L}(A_1) = L_1$, $\mathbb{L}(A_2) = L_2$ and $\mathbb{L}(A_S) = L_S$.

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 $\mathbb{L}(A_1 \amalg C) \subseteq \mathbb{L}(A_S)$. If not, let $(t, R_1 \cup R)$ be the counterexample obtained.

Note that $(t, R) \in \mathbb{L}(C) \setminus U$. We return FALSE to L^F along with the counterexample (t, R). If $\mathbb{L}(A_1 \amalg C) \subseteq \mathbb{L}(A_S)$, we proceed to step 2.

2. Check if $\mathbb{L}(A_2) \subseteq \mathbb{L}(C)$. If so, we have obtained an assumption, viz., $\mathbb{L}(C)$, that discharges both premises of **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 $(t', R'_1) \in \mathbb{L}(A_1)$ such that $(t', R'_1 \cup R') \notin \mathbb{L}(A_S)$. If so, then $(t', R'_1 \cup R') \in \mathbb{L}(A_1 \amalg A_2) \setminus \mathbb{L}(A_S)$ and the overall language containment algorithm terminates with FALSE and the counterexample $(t', R'_1 \cup R')$. Otherwise $(t', R') \in U \setminus \mathbb{L}(C)$ 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 $\mathbb{L}(C) = L_W$. In this scenario, termination is guaranteed to occur due to Theorem 10.

7.6 Arbitrary Components and Circularity

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

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

Exp	LOC	C	St	No Deadlock							
				Plain		**NC**			**C**		
				T	M	T	M	A	T	M	A
MC	7272	2	2874	-	*	308	**903**	**5**	**307**	**903**	6
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
ide	18905	4	716	972	*	*	63	-	**195**	**55**	**24**
ide	18905	5	760	1082	*	*	84	-	**639**	**85**	**48**
syn	17262	4	117	733	*	1547	**19**	**21**	**58**	21	24
syn	17262	5	127	713	*	*	19	-	**224**	**47**	**48**
syn	17262	6	137	767	*	*	27	-	**1815**	**189**	**96**
mx	15717	3	1995	1154	*	2079	140	**11**	**639**	**123**	12
mx	15717	4	2058	1545	*	-	168	-	**713**	**139**	24
mx	15717	5	2121	1660	*	-	179	-	**2131**	**185**	48
tg3	36774	3	1653	971	*	1568	118	**11**	**406**	**111**	12
tg3	36774	4	1673	927	*	-	149	-	**486**	**131**	24
tg3	36774	5	1693	1086	*	-	158	-	**1338**	**165**	48
tg3	36774	6	1713	1252	*	-	157	-	**3406**	**313**	96
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	-

Exp	LOC	C	St	Deadlock							
				Plain		**NC**			**C**		
				T	M	T	M	A	T	M	A
MC	7272	2	2874	372	β	386	980	**13**	**313**	**979**	16
MC	7272	3	2874	-	-	-	-	-	-	-	-
MC	7272	4	2874	-	-	-	-	-	-	-	-
ide	18905	3	672	755	*	*	80	-	**557**	**551**	**125**
ide	18905	4	716	978	*	*	84	-	2913	*	-
ide	18905	5	760	1082	*	*	89	-	*	498	-
syn	17262	4	117	864	*	**127**	**181**	**2**	133	**181**	6
syn	17262	5	127	1088	*	844	*	-	867	*	-
syn	17262	6	137	-	*	1188	*	-	-	*	-
mx	15717	3	1995	1182	*	657	**364**	**2**	**630**	**364**	5
mx	15717	4	2058	1309	*	1627	*	-	1206	*	-
mx	15717	5	2121	-	*	3368	*	-	2276	*	-
tg3	36774	3	1653	894	*	**486**	**393**	**2**	499	**393**	5
tg3	36774	4	1673	1096	*	1036	*	-	1037	*	-
tg3	36774	5	1693	-	*	2186	*	-	1668	*	-
tg3	36774	6	1713	1278	*	*	-	-	1954	*	-

Table 7.1: Experimental results for AGR for checking deadlock. 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.

At the top-level, we apply **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 **NC**. In terms of the implementation of the Teacher, the only difference is in step 2 of the candidate query (cf. Section 7.5). More specifically, we now invoke the language containment procedure recursively with $\mathbb{L}(A_2)$, $\mathbb{L}(A_3)$ and $\mathbb{L}(C)$ instead of checking directly for $\mathbb{L}(A_2) \subseteq \mathbb{L}(C)$. This technique can be extended to any finite number of components.

7.6.1 Circular AG Rule

We also explored a circular AG rule. Unlike **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 11 *Consider any two RFLs L_1 and L_2. Then the following proof rule, which we call* **C***, 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 10 *We first prove soundness by contradiction. Assume that three premises hold but the conclusion does not. Then 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 very definition of* $W(L_1)$ *and* $W(L_2)$*. We prove the third premise by contradiction. Suppose that 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* $(t, R) \notin W(L_2)$*. But this means that* $(t, R) \in L_1$ *and* $(t, \Sigma \setminus R) \in L_2$ *which implies that* $(t, \Sigma) \in L_1 \parallel L_2$ *and contradicts the conclusion.*

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 **NC** (cf. Section 7.5). Similarly, using the second premise, construct another candidate C_2. Construction of C_1 and C_2 proceeds exactly as in the case of **NC**.
2. Check if $W(\mathbb{L}(C_1)) \parallel W(\mathbb{L}(C_2)) \subseteq \overline{L_{Dlk}}$. This is done either directly or via a compositional language containment using **NC**. We compute the automata for $W(\mathbb{L}(C_1))$ and $W(\mathbb{L}(C_2))$ using the procedure described in the proof of Theorem 10. 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 $(t, R) \in W(\mathbb{L}(C_1))$ and $(t, \Sigma \backslash R) \in W(\mathbb{L}(C_2))$. Check if $(t, R) \in L_1$ and $(t, \Sigma \backslash R) \in L_2$. 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 $(t, R) \in W(\mathbb{L}(C_1))$, $(t, \Sigma \setminus R) \notin \mathbb{L}(C_1)$. 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 **C** in the context of only two components, it generalizes to an arbitrary, but finite, number of components.

7.7 Experimental Validation

We implemented our algorithms in the ComFoRT [29] 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 7.1. 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 [29] 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 7.1 as "Plain"), the non-circular AG rule (referred as **NC**), and the circular AG rule (referred as **C**). When using **C**, Step 2 (i.e., checking if $W(\mathbb{L}(C_1)) \parallel W(\mathbb{L}(C_2)) \subseteq \overline{L_{Dlk}}$) was done via compositional language containment using **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, **C** is in general faster than **NC** but (on a few occasions) consumes negligible extra memory. In several cases, **NC** runs out of time while **C** is able to terminate successfully. Overall, whenever **NC** and **C** differ significantly in any real-life example, **C** is superior.

7.8 Conclusions and Related Work

In this chapter, we presented a learning-based automated assume guarantee paradigm for 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 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.

Overkamp has explored synthesis of supervisory controller for discrete-event systems [110] based on failure semantics [82]. 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. [128] 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. [89]. 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 [83] and for checking refinement of CCS processes, using a more discriminative (than failure trace refinement) notion called stuck-free conformance [61].

Brookes and Roscoe [25] 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 rules.

Chapter 8

Conclusions and Future Directions

Compositional reasoning methods have been of limited use in practice due to lack of automation. In this thesis, we have presented an automated assume-guarantee reasoning (AGR) framework for verifying both hardware and software systems which utilizes machine learning algorithms for finite state models together with model checking algorithms. We have instantiated this framework for both asynchronously executing software systems with rendezvous communcation and synchronous hardware systems with shared memory based communication. Compositional techniques for checking both simulation and deadlock on finite state systems were presented. Finally, we proposed a method to scale the above framework based on SAT-based model checking and lazy learning algorithms.

The automated AGR framework has been put to use in solving the pervasive component substitutability problem for hardware and software systems. Our solution consists of two parts: the containment check involves the simultaneous use of over- and under-approximation of the upgraded components, while the compatibility check uses an incremental AGR technique to check substitutability. The compatibility check focuses on re-validating safety properties with respect to the notion of trace-containment. Since the check is based on the automated AGR framework and the learning algorithm L^*, it can be extended to checking other kinds of properties, i.e., simulation and deadlock, based on the

algorithms developed in this thesis.

The use of machine learning algorithms for computing environment assumptions automatically has made automated compositional reasoning feasible, and to a certain extent, scalable. However, there are a number of interesting open problems to be solved for scaling the technique to industrial size models. Automated techniques for decomposing systems for efficient AGR need to be investigated. An initial approach has been proposed by Nam and Alur [107] based on using hypergraph partitioning algorithms. In the AGR framework for the generalized rule **NC**, we observed that the order of components in the instantiation of the rule has a significant impact on the algorithm run times. Therefore, the problem of ordering of system decompositions in the non-circular rule needs investigation.

Another important research direction is towards making the AGR framework scale to industrial size systems. In particular, we believe that AGR will be effective for systems whose components are *loosely-coupled*, i.e., there is a weak correlation or inter-dependency between the local variables of different components. It may be possible to improve the framework by exploiting the fact that the components are loosely-coupled. We think that a formal characterization of the inter-dependency between system components needs to be developed.

A large number of useful specifications are of the form *something good must happen eventually*. Formally, they are known as liveness specifications and and useful for specifying behaviors of non-terminating systems. Extending the automated AGR framework to checking liveness properties is a non-trivial problem. The main issue is that we need to develop a learning algorithm for the set of ω-regular languages. Recall that the algorithm L^* relies on the characterization of the regular languages in terms of Nerode congruence: the states in the DFA accepting a regular language L are *isomorphic* to the Nerode congruence classes of L. Unfortunately, the congruence-based characterization of ω-regular languages are non-trivial and do not have this isomorphism property with respect to ω-automata, e.g., Müller or Büchi automata. Hence, developing a learning algorithm for the

set of ω-regular languages is a difficult task and needs investigation. Another interesting problem is to learn models with respect to the notion of bisimulation minimization [42]. In contrast to learning regular languages in form of a deterministic finite automata, the bisimulation-based approach may lead to smaller models as a product of learning.

Bibliography

[1] Foci: An interpolating prover. http://www.kenmcmil.com/foci.html. 6.5

[2] The vis home page. http://vlsi.coloradu.edu/ vis/. 6.5

[3] Yices: An SMT Solver. http://yices.csl.sri.com/. 6.2.3, 6.5

[4] *Concurrency verification: introduction to compositional and noncompositional methods*. Cambridge University Press, New York, NY, USA, 2001. ISBN 0-521-80608-9. 3.5

[5] abb-site. ABB Inc. http://www.abb.com. 4.5

[6] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Symp. on Principles Of Programming Languages (POPL)*, 2005. 2.5.2

[7] Rajeev Alur, Luca de Alfaro, Thomas A. Henzinger, and Freddy Y. C. Mang. Automating modular verification. In *International Conference on Concurrency Theory*, pages 82–97, 1999. 3.5.2

[8] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. Mocha: Modularity in model checking. In *CAV*, pages 521–525, 1998. 3.5.2

[9] Nina Amla, Xiaoqun Du, Andreas Kuehlmann, Robert P. Kurshan, and Kenneth L. McMillan. An analysis of sat-based model checking techniques in an industrial envi-

ronment. In *CHARME*, pages 254–268, 2005. 6.2.3, 6.6

[10] Nina Amla, E. Allen Emerson, Kedar S. Namjoshi, and Richard J. Trefler. Abstract patterns of compositional reasoning. In *CONCUR*, pages 423–438, 2003. 3.5.2

[11] G. Ammons, R. Bodik, and J.R. Larus. Mining specifications. *ACM SIGPLAN Notices*, 37(1):4–16, 2002. 2.5.2

[12] Dana Angluin. Learning regular sets from queries and counterexamples. In *Information and Computation*, volume 75(2), pages 87–106, November 1987. 2.2, 2.3, 1, 3.3

[13] Dana Angluin and Carl H. Smith. Inductive inference: Theory and methods. *ACM Comput. Surv.*, 15(3):237–269, 1983. 2.1

[14] Roy Armoni, Limor Fix, Ranan Fraer, Scott Huddleston, Nir Piterman, and Moshe Y. Vardi. Sat-based induction for temporal safety properties. *Electr. Notes Theor. Comput. Sci.*, 119(2):3–16, 2005. 6.2.3

[15] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*, volume 36(5) of *SIGPLAN Notices*, pages 203–213. ACM Press, June 2001. ISBN 1-58113-414-2. 1, 4.3

[16] Thomas Ball and Sriram K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical report MSR-TR-2002-09, Microsoft Research, Redmond, Washington, USA, January 2002. 2.5.1

[17] H. Barringer, D. Giannakopoulou, and C.S Păsăreănu. Proof rules for automated compositional verification. In *2nd Workshop on Specification and Verification of Component-Based Systems, ESEC/FSE 2003*. 3.2, 3.2, 3.4, 3.5.2

[18] Saddek Bensalem, Ahmed Bouajjani, Claire Loiseaux, and Joseph Sifakis. Property

preserving simulations. In *CAV '92: Proceedings of the Fourth International Workshop on Computer Aided Verification*, pages 260–273, London, UK, 1993. Springer-Verlag. ISBN 3-540-56496-9. 3.5.1

[19] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines with parameters. In *FASE*, pages 107–121, 2006. 6.4.4, 6.5, 6.1, 6.6

[20] Marc Bernard and Colin de la Higuera. Gift: Grammatical inference for terms. In *International Conference on Inductive Logic Programming*, 1999. 2.5, 5.5

[21] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Y. Zue. *Bounded Model Checking*, volume 58 of *Advances in computers*. Academic Press, 2003. 6.2.3, 6.2.3, 6.6

[22] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Yhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, March 1999. ISBN 3-540-65703-7. 1

[23] Colin Blundell, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Assume-guarantee testing. In *4th Workshop on Spec. and Ver. of Component-based Systems, FSE*, 2005. 3.5.2

[24] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Ziyad Hanna, Zurab Khasidashvili, Amit Palti, and Roberto Sebastiani. Encoding rtl constructs for mathsat: a preliminary report. *Electr. Notes Theor. Comput. Sci.*, 144 (2):3–14, 2006. 6.2.3

[25] Stephen D. Brookes and A. W. Roscoe. Deadlock analysis in networks of communicating processes. *Distributed Computing*, 4:209–230, 1991. 7.8

[26] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE*

Trans. Computers, 35(8):677–691, 1986. 1

[27] Rafael C. Carrasco, Jos Oncina, and Jorge Calera-Rubio. Stochastic inference of regular tree languages. In *ICGI '98: Proceedings of the 4th International Colloquium on Grammatical Inference*, pages 187–198. Springer-Verlag, 1998. ISBN 3-540-64776-7. 2.5

[28] S. Chaki, E. Clarke, D. Giannakopoulou, and C. S. Păsăreănu. Abstraction and assume-guarantee reasoning for automated software verification. Technical Report 05.02, Research Institute for Advanced Computer Science (RIACS), 2004. 2.5.2

[29] S. Chaki, J. Ivers, N. Sharygina, and K. Wallnau. The ComFoRT reasoning framework. In *Proc. of CAV*, pages 164–169, 2005. 7.1, 7.7

[30] Sagar Chaki, Edmund Clarke, Alex Groce, Joël Ouaknine, Ofer Strichman, and Karen Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design (FMSD)*, 25(2–3):129–166, September – November 2004. 4.3, 5.4

[31] Sagar Chaki, Edmund Clarke, Natasha Sharygina, and Nishant Sinha. Dynamic component substitutability analysis. In *Proc. of Conf. on Formal Methods*, 2005. 3.5.2

[32] Sagar Chaki, Edmund Clarke, Nishant Sinha, and Prasanna Thati. Automated assume-guarantee reasoning for simulation conformance. In *Proc. of 17th Int. Conf. on Computer Aided Verification*, 2005. 3.5.2

[33] Sagar Chaki and Ofer Strichman. Optimized L* for assume-guarantee reasoning. In *TACAS*, 2007. 3.5.2, 6.6

[34] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, Marcin Jurdzinski, and Freddy Y.C. Mang. Interface compatibility checking for software modules. In *CAV*. LNCS 2404, 2002. 4.6

[35] Shing-Chi Cheung and Jeff Kramer. Context constraints for compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 5(4):334–377, 1996. 3.5.1

[36] Shing-Chi Cheung and Jeff Kramer. Checking safety properties using compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 8(1):49–78, 1999. 3.5.1

[37] Edmund Clarke and Allen Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In Dexter Kozen, editor, *Proceedings of Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, May 1981. ISBN 3-540-11212-X. 1

[38] Edmund Clarke, David Long, and Kenneth McMillan. Compositional model checking. In *Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science (LICS '89)*, pages 353–362. IEEE Computer Society Press, June 1989. 3.5.1

[39] Edmund M. Clarke, O. Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and System (TOPLAS)*, 16(5): 1512–1542, September 1994. ISSN 0164-0925. 1, 3.5.1, 4.3, 4.3

[40] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, July 2000. ISBN 3-540-67770-4. 1

[41] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, September 2003. 2.5.1

[42] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, MA, 2000. 1, 3.5.1, 7.3, 8

[43] J. M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning as-

sumptions for compositional verification. In Hubert Garavel and John Hatcliff, editors, *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*, pages 331–346. Springer-Verlag, April 2003. ISBN 3-540-00898-5. 3, 3.2, 4.4.2

[44] J. M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In Hubert Garavel and John Hatcliff, editors, *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*, pages 331–346. Springer-Verlag, April 2003. ISBN 3-540-00898-5. 1.1, 3.4, 3.5.2, 7.1

[45] Jamieson M. Cobleigh, George S. Avrunin, and Lori A. Clarke. Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In *ISSTA*, pages 97–108, 2006. 3.5.2

[46] Ariel Cohen and Kedar Namjoshi. Local proofs for global safety properties. In *CAV*, 2007. 3.5.2

[47] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*, chapter 1. 2002. available at http://www.grappa.univ-lille3.fr/tata/. 1, 5.1

[48] Dennis Dams, Orna Grumberg, and Rob Gerth. Generation of reduced models for checking fragments of ctl. In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 479–490, London, UK, 1993. Springer-Verlag. 3.5.1

[49] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *FSE*, 2001. 4.6

[50] Colin de la Higuera. A bibliographical study of grammatical inference. *Pattern*

Recognition, 38(9):1332–1348, 2005. 2.2, 2.5

[51] Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors. *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 8-12, 1997. Revised Lectures*, volume 1536 of *Lecture Notes in Computer Science*, 1998. Springer. 3.5

[52] Edsger Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. 4.3

[53] Frank Drewes and Johanna Hogberg. Learning a regular tree language. In *LNCS 2710, pp. 279–291, Proc. Developments in Language Theory (DLT) '03*. 5.5

[54] Pierre Dupont. Incremental regular inference. In *ICGI*, pages 222–237, 1996. 2.5

[55] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, pages 81–94, 2006. 6.5

[56] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electr. Notes Theor. Comput. Sci.*, 89(4), 2003. 6.2.3, 6.2.3

[57] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering (ICSE'99)*, pages 213–224, 1999. 2.5.2

[58] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Thread-modular verification for shared-memory programs. In *European Symposium on Programming*, pages 262–277, 2002. 3.5.2

[59] Cormac Flanagan, Stephen N. Freund, Shaz Qadeer, and Sanjit A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1-3):153–183, 2005. 3.5.2

[60] Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In *SPIN*, pages 213–224, 2003. 3.5.2

[61] Cédric Fournet, C. A. R. Hoare, Sriram K. Rajamani, and Jakob Rehof. Stuck-free

conformance. In *CAV*, pages 242–254, 2004. 7.8

[62] P. Garca and J. Oncina. Inference of recognizable tree sets. Technical report dsic ii/47/1993, Departamento de Sistemas Informticos y Computacin, Universidad Politcnica de Valencia, 1993. 2.5, 5.5

[63] Mihaela Gheorghiu, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Refining interface alphabets for compositional verification. In *TACAS*, 2007. 3.5.2, 6.6

[64] Dimitra Giannakopoulou, Corina S. Păsăreanu, and Howard Barringer. Assumption generation for software component verification. In *Proceedings of the ASE*, 2002. ISBN 0-7695-1736-6. 3.5.2

[65] Dimitra Giannakopoulou, Corina S. Păsăreanu, and Howard Barringer. Component verification with automatically generated assumptions. *Autom. Softw. Eng.*, 12(3): 297–320, 2005. 3.5.2

[66] Dimitra Giannakopoulou, Corina S. Păsăreanu, and Jamieson M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *ICSE*, pages 211–220, 2004. 3.5.2

[67] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5): 447–474, May 1967. 2.1, 2.5

[68] E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, June 1978. 2.5

[69] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, June 1997. ISBN 3-540-63166-6. 1, 4.3

[70] Susanne Graf and Bernhard Steffen. Compositional minimization of finite state systems. In *CAV '90: Proceedings of the 2nd International Workshop on Computer*

Aided Verification, pages 186–196, London, UK, 1991. Springer-Verlag. ISBN 3-540-54477-1. 3.5.1

[71] Susanne Graf, Bernhard Steffen, and Gerald Lüttgen. Compositional minimisation of finite state systems using interface specifications. *Formal Asp. Comput.*, 8(5): 607–616, 1996. 3.5.1

[72] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 357–370, 2002. 2.5.2, 1

[73] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, 1994. 1

[74] Anubhav Gupta, Ken McMillan, and Zhaohui Fu. Automated assumption generation for compositional verification. In *CAV*, 2007. 3.5.2

[75] Arie Gurfinkel, Ou Wei, and Marsha Chechik. Yasm: A software model-checker for verification and refutation. In *CAV*, pages 170–174, 2006. 4.3, 4.3, 4.5

[76] P. Habermehl and T. Vojnar. Regular model checking using inference of regular languages. In *Proceddings of INFINITY'04*, 2004. 2.5.2

[77] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. *SIGPLAN Not.*, 39(6):1–13, 2004. ISSN 0362-1340. 3.5.2

[78] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Permissive interfaces. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 31–40, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-014-0. 2.5.2

[79] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In Warren A. Hunt Jr. and Fabio Somenzi, edi-

tors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV '03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 262–274. Springer-Verlag, July 2003. ISBN 3-540-40524-0. 3.5.2

[80] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '02)*, volume 37(1) of *SIGPLAN Notices*, pages 58–70. ACM Press, January 2002. ISBN 1-58113-450-9. 4.3, 6.6

[81] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: Methodology and case studies. In *CAV*, pages 440–451, 1998. 3.5.2

[82] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, London, 1985. 1.1, 3.4, 3.5.2, 7.1, 7.3, 7.8

[83] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003. 7.8

[84] JE Hopcroft and JD Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979. 1, 2.3, 2.4, 2.5.1

[85] James Ivers and Natasha Sharygina. Overview of ComFoRT: A model checking reasoning framework. *CMU/SEI-2004-TN-018*, 2004. 4.1, 4.5, 5.5

[86] Ralph D. Jeffords and Constance L. Heitmeyer. A strategy for efficiently verifying requirements. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference, with FSE*, pages 28–37, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-743-5. 3.5.2

[87] Ranjit Jhala and Kenneth L. McMillan. Microarchitecture verification by compositional model checking. In *CAV*, pages 396–410, 2001. 3.5.2

[88] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983. 1, 3, 3.5.2

[89] V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *Proc. of CAV*, pages 505–518, 2005. 7.8

[90] O. Kupferman and Y. Lustig. Lattice automata. In *Proc. 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 199 – 213. Springer-Verlag, 2007. 7.1

[91] Robert P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994. ISBN 0-691-03436-2. 1, 2.5.1

[92] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994. ISSN 0164-0925. 4.6

[93] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995. 3.5.1

[94] Patrick Maier. A set-theoretic framework for assume-guarantee reasoning. In *ICALP*, pages 821–834, 2001. 3.5.2, 6.2

[95] Patrick Maier. *A Lattice-Theoretic Framework For Circular Assume-Guarantee Reasoning*. PhD thesis, Universität des Saarlandes, Saarbrücken, July 2003. 3.5.2

[96] Stephen McCamant and Michael D. Ernst. Early identification of incompatibilities in multi-component upgrades. In *ECOOP Conference*, Olso, Norway, 2004. 4.6

[97] Kenneth L. McMillan. A compositional rule for hardware design refinement. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 24–35. Springer-Verlag, June 1997. ISBN 3-540-63166-6. 1, 3.5.2

[98] Kenneth L. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In *CAV*, pages 110–121, 1998. 3.5.2

[99] Kenneth L. McMillan. Circular compositional reasoning about liveness. In *CHARME*, pages 342–345, 1999. 3.5.2

[100] Kenneth L. McMillan. Verification of infinite state systems by compositional model checking. In *CHARME*, pages 219–234, 1999. 3.5.2

[101] Kenneth L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.*, 37(1-3):279–309, 2000. 3.5.2

[102] Kenneth L. McMillan. Interpolation and sat-based model checking. In *CAV*, pages 1–13, 2003. 6.2.3, 6.2.3

[103] Kenneth L. McMillan, Shaz Qadeer, and James B. Saxe. Induction in compositional model checking. In *CAV*, pages 312–327, 2000. 3.5.2

[104] Robin Milner. *Communication and Concurrency*. Prentice-Hall International, London, 1989. 1

[105] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981. 1, 3

[106] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th ACM IEEE Design Automation Conference (DAC '01)*, pages 530–535. ACM Press, June 2001. ISBN 1-58113-297-2. 1

[107] Wonhong Nam and Rajeev Alur. Learning-based symbolic assume-guarantee reasoning with automatic decomposition. In *ATVA*, pages 170–185, 2006. 3.2, 3.2, 1, 3.5.2, 6.5, 6.6, 8

[108] Kedar S. Namjoshi and Richard J. Trefler. On the completeness of compositional reasoning. In *Proceedings of the 12th Int. Conference on Computer Aided Verification (CAV2000)*, number 1855, pages 139–153. Springer-Verlag, 2000. 3.5.2, 6.2

[109] P. Oncina, J.; Garca. Identifying regular languages in polynomial time. World Sci-

entific Publishing, 1992. Advances in Structural and Syntactic Pattern Recognition,. 2.5

[110] A. Overkamp. Supervisory control using failure semantics and partial specifications. *Automatic Control, IEEE Transactions on*, 42(4):498–510, April 1997. 7.8

[111] D. Peled, M.Y. Vardi, and M. Yannakakis. Black box checking. In *FORTE/PSTV*, 1999. 2.5.2

[112] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and models of concurrent systems*. Springer-Verlag New York, Inc., 1985. 1, 3, 3.2

[113] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat-based formal verification. *STTT*, 7(2):156–173, 2005. 1, 6.2.3, 6.4.3, 6.6

[114] Corina S. Păsăreanu and Dimitra Giannakopoulou. Towards a compositional spin. In *SPIN*, pages 234–251, 2006. 3.5.2

[115] W. Nam R. Alur, P. Madhusudan. Symbolic compositional verification by learning assumptions. In *Proc. of 17th Int. Conf. on Computer Aided Verification*, 2005. 3.5.2, 6.4, 6.5, 6.6

[116] Kavita Ravi and Fabio Somenzi. Minimal assignments for bounded model checking. In *TACAS*, pages 31–45, 2004. 6.4.3

[117] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. In *Information and Computation*, volume 103(2), pages 299–347, 1993. 2.2, 2.3, 3.3, 3

[118] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Int., 1997. 4.2, 7.2

[119] Yasubumi Sakakibara. Learning context-free grammars from structural data in polynomial time. *Theor. Comput. Sci.*, 76(2-3):223–242, 1990. 2.5, 5.5

[120] Mary Sheeran, Satnam Singh, and Gunnar Stalmarck. Checking safety properties using induction and a sat-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 108–125, London, UK, 2000. Springer-Verlag. 6.2.3

[121] ShengYu Shen, Ying Qin, and Sikun Li. Minimizing counterexample with unit core extraction and incremental sat. In *VMCAI*, pages 298–312, 2005. 6.4.3

[122] Thomas R. Shiple, Massimiliano Chiodo, Alberto L. Sangiovanni-Vincentelli, and Robert K. Brayton. Automatic reduction in ctl compositional model checking. In *CAV '92: Proceedings of the Fourth International Workshop on Computer Aided Verification*, pages 234–247, London, UK, 1993. Springer-Verlag. ISBN 3-540-56496-9. 3.5.1

[123] Sharon Shoham and Orna Grumberg. Monotonic abstraction-refinement for ctl. In *TACAS*, pages 546–560, 2004. 4.3, 4.3, 4.3

[124] Nishant Sinha and Edmund Clarke. Sat-based compositional verification using lazy learning. In *CAV*, 2007. 3.5.2

[125] C. Tinelli and S. Ranise. SMT-LIB: The Satisfiability Modulo Theories Library. http://goedel.cs.uiowa.edu/smtlib/, 2005. 6.2.3

[126] Abhay Vardhan, Koushik Sen, Mahesh Viswanathan, and Gul Agha. Actively learning to verify safety for FIFO automata. In *FSTTCS'04*, LNCS, Chennai, India, December 2004. Springer. 2.5.2

[127] Mahesh Viswanathan and Ramesh Viswanathan. Foundations for circular compositional reasoning. *Lecture Notes in Computer Science*, 2076:835–847, 2001. 3.5.2

[128] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *Proc. of ECOOP*, pages 602–629, 2005. 7.8