Learning Bisimulation

by

Warren Shenkenfelder

B.Sc., University of Victoria, 2005

A Thesis submitted in partial fullfilment if the

Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Learning Bisimulation

By

Warren Shenkenfelder

B.Sc., University of Victoria, 2005

Supervisory Committee

Dr. Bruce Kapron, Supervisor

Department of Computer Science

Dr. Valerie King, Co-supervisor

Department of Computer Science

Dr. Venkatesh Srinivasan, Departmental Member

Department of Computer Science

Supervisory Committee

Dr. Bruce Kapron, Supervisor

Department of Computer Science

Dr. Valerie King, Co-supervisor

Department of Computer Science

Dr. Venkatesh Srinivasan, Departmental Member

Department of Computer Science

Abstract

Computational learning theory is a branch of theoretical computer science that re-imagines the role of an algorithm from an agent of computation to an agent of learning. The operations of computers become those of the human mind; an important step towards illuminating the limitations of artificial intelligence. The central difference between a learning algorithm and a traditional algorithm is that the learner has access to an oracle who, in constant time, can answer queries about that to be learned. Normally an algorithm would have to discover such information on its own accord. This subtle change in how we model problem solving results in changes in the computational complexity of some classic problems; allowing us to re-examine them in a new light. Specifically two known result are examined: one positive, one negative. It is know that one can efficiently learn Deterministic Finite Automatons with queries, not so of Non-Deterministic Finite Automatons. We generalize these Automatons into Labeled Transition Systems and attempt to learn them using a stronger query.

Table of Contents

List of Tables

List of Figures

Common Symbols

Symbol	Meaning
$\prec$	A partial ordering (see Section 1.2.1)
$\sqsubset$	A refinement (see Section 1.2.1)
$[x]$	If x is in a set (especially of strings) this refers to the equivalence class of x (see Section 1.2.1)
$[\![x]\!]$	Denotes x's best-fit equivalence class (see Section 2.5)
$(x\|_i$	The i^{th} prefix of string x (see Section 1.2.3)
$\|x)_i$	The i^{th} suffix of string x (see Section 1.2.3)
$\langle\alpha\rangle\varphi$	There exists an α-transition leading to a state which satisfies φ (see Section 4.2)
$[\alpha]\varphi$	All α-transitions lead to a state which satisfies φ (see Section 4.2)
δ_{pt}	A distinguishing formula for state pt (see Section 4.2.6)
$pt(i)$	A distinguishing formula for i^{th} prefix of trace to state pt (see Section 4.3.2)
$\langle\!\langle \alpha^*_{pt} \rangle\!\rangle$	Total trace to state pt (see Section 5.3.2)

Acknowledgements

I would like to thank my supervisors for putting up with me.
Also, may I extend thanks to anyone who has ever cared for me, even if only for a fleeting instant. I appreciate your thoughts.
On a more technical level, I would like to thank the fine makers of LaTeX for typesetting my thesis. You made my thesis look better then it deserves.
And finally, I would like to thank Nintendo, fine makers of Donkey Kong, Mario, and Zelda, which have all kept me greatly entertained throughout the years.

I dedicate this to anyone who is bothering to read it. I thank-you.

Chapter 1

Introduction to Learning Theory

Computational Learning Theory is a branch of complexity theory that proposes models of learning, then studies the tractability of learning varying classes of objects under these models. The models of learning are often rooted in statistics: Probably Approximately Correct (PAC) learning (Valiant [20]). They vary in the amount of power provided to the learner: assisted versus non-assisted learning. Ultimately these models allow us to design learning algorithms within a consistent and rigorous framework, from which we can study the *predictive* power associated with certain problems. For instance one may ask, given two subsets: one of words in a fixed regular language, and another set of words not in that language; can we construct a minimal Deterministic Finite Automaton (DFA) accepting all the strings in the first set and none of the strings in the second set? It turns out that unless $P \neq NP$, no DFA, whose number of states is a *polynomial* function of the number of states of the minimal such DFA, can be computed in polynomial time.

1.1 Introduction

It is questions such as the learnability of DFAs which provide the impetus for this thesis. However we turn our focus instead to labeled transition systems (LTS), which may be seen as a generalization of DFAs. The underlying theory behind LTSs and DFAs, particularly concerning the notions of equivalence in those theories, suggest strong correlations between the two subjects, further suggesting some value in studying whether these parallels

are more then just superficial. In exploring the contrast we may elucidate the intrinsic differences between these two notions.

We begin our approach of this problem by examining several models of learning and known tractability results relevant to this research. Most of the results are discussed in more depth by Kearns and Vazirani ([12]). Many of the intractability results from the PAC learning model are of particular interest because they are based on Cryptographic assumptions. There is a natural relation between notions of inverting one way functions, and objects which are difficult to learn.

The most pertinent learning theory result, in regards to this thesis, is Angluin's algorithm for the assisted learning of DFAs. We are concerned with it because the theoretical framework for the algorithm displays the most harmony with the theory behind LTS equivalence. The backbone of Angluin's Algorithm is the Myhill-Nerode Theorem [13]. We present in chapter 2 a modest rephrasing of Angluin's algorithm to accentuate the parallels between DFAs and LTSs. We go on to study the theory behind LTSs in the following two chapters: 3 and 4. Chapter 5 deals with constructing a learning algorithm for Labeled Transition Systems. We begin, however, by considering some mathematical preliminaries, following with an introduction to models of learning, which occupies the remainder of this chapter.

1.2 Mathematical Preliminaries

We present the definition of basic mathematical concepts used in the thesis. Our goal is to have a thesis which is logically complete, although we assume fundamental results for brevity. For instance, we assume knowledge of set theory and basic graph theory

terminology such as trees, paths, and directed acyclic graphs. However, we do include discussions of basic complexity theory and basic automata theory due to their primary importance to this thesis.

1.2.1 Relations and Partitions

Since we will use the notion of an equivalence relation extensively in the following chapters, we define it now.

A *partial ordering* of a set A, is a relation $\prec$ which satisfies three conditions:

i) $\forall a \in A,\ a \prec a$

ii) $\forall a, b \in A$, if $a \prec b$ and $b \prec a$ then $a = b$

iii) $\forall a, b, c \in A$, if $a \prec b$ and $b \prec c$ then $a \prec c$

We denote a partial order $\prec$ over the set A as the set of ordered pairs $\{(a, b)|a, b \in A,\ a \prec b\}$. For a partial order $\prec$ we can define a new partial order $\succ = \{(a, b)|(b, a) \in \prec\}$. We call any relation $\equiv$, over a set A, an equivalence relation if $\exists \prec$ over A such that:

$$\equiv = \prec \bigcup \succ$$

For any equivalence relation $\equiv$ over a set A, this relation induces a *partition* of the set into disjoint blocks (equivalence classes), which we will denote by the set $\varphi = \{B_1 \ldots B_n\}$. Each block B_i contains those elements of A related by $\equiv$. That is, if $a, b \in A$ and $a \equiv b$, then $a, b \in B_i$ for some i. Thus we can write $\equiv = \{B_1 \ldots B_n\}$. Where no confusion arises we will both refer to an equivalence relation $\equiv$ as a set φ of equivalence classes $\{B_1 \ldots B_n\}$ and as a set of ordered pairs $\{(a, b)|a, b \in A,\ a \equiv b\}$. In this sense $\equiv = \varphi$. We use the terms equivalence class and partition interchangeably. For each equivalence

class we can select a canonical element to refer to this block. For $x \in A$ we write $[x]$ to refer to the block B_i such that $x \in B_i$.

We say a partition ϕ refines φ (denoted by $\phi \sqsubset \varphi$) if every equivalence class of ϕ is a subset of an equivalence class of φ. The refinement relation $\sqsubset$ is a partial ordering of partitions; for $\phi \sqsubset \varphi$ we say ϕ is finer then φ, and φ coarser then ϕ.

1.2.2 Basic Complexity Theory

Our goal in designing learning algorithms is to separate what can be learned from what cannot be learned. Typical complexity theory sensibilities equate 'not being able to' with 'not efficient', since a problem that takes millions of years to solve is still considered computable. In turn efficiency is equated with polynomial running time (easy problems). If we have an learning algorithm for a concept that runs in time polynomial in the size of the input, we say we can efficiently learn that concept. We want to say a problem can not be learned (*efficiently*) if the fastest algorithm for learning that concept has at least an expected running time that is exponential or worse (hard problems).

The problem with the above statement is we may never be sure we have the fastest algorithm –it is a subjective statement– maybe a faster one is waiting to be discovered. We can, however, compare the relative difficulty of two problems, thus defining a partial order $\prec_r$ over problems. If we can efficiently transform an instance of a problem A into an instance of another problem B, such that we can use an algorithm that solves B to solve A, then clearly A can be no harder then B. Thus $A \prec_r B$. Traditionally we say A *reduces* to B. The key is that the transformation cannot destroy the intricacy of the original problem; instances of A and B related by the transformation must have consistent answers. Furthermore, the transformation must be done in polynomial time —otherwise we could

use an exponential time transformation to pre-solve the problem, making it trivial.

The above is an intuitive definition of reductions. Depending on the type of problem the requirements of what constitute a correct reduction may vary. Consider the context of *decision problems*: given some input instance $\omega \in U = L \bigcup \bar{L}$ for a problem A, an algorithm must decide if $\omega \in L$ (YES-instance) or if $\omega \in \bar{L}$ (NO-instance). Preserving the intricacy of the problems simply means mapping YES-instances to YES-instances, and NO-instances to NO-instances. If the problems are learning problems, the form the reduction takes will be different.

The set P is the set of problems that can be solved efficiently. The set NP is the set of problems whose solutions can be verified efficiently. Clearly $P \subset NP$. It is not known if $NP \subset P$. A problem A is NP-hard if $\forall B \in NP,\ B \prec_r A$. If a problem A is NP-hard and A $\in NP$, then we say the problem A is NP-complete. These definitions can be extended beyond decision problems and it is not uncommon to see search problems and optimization problems described as NP-Hard.

Proposition 1.2.1. *The problem SAT, of determining if a propositional logic formula has a* sat*isfying assignment, is NP-complete.*

Problems which are NP-hard represent the best candidates in NP for being problems without efficient algorithms, though if $P = NP$ many NP-hard problems (the complete ones) would be shown to be easy. Thus, we say we cannot learn a concept if the problem, B, of learning that concept satisfies $A \prec_r B$, where A is NP-hard and $\prec_r$ is a suitable reduction.

1.2.3 Automata

Let $\Sigma = \Sigma^1$ be a set. We call Σ an alphabet, and its elements characters. Define $\epsilon = \Sigma^0$ as the empty string in this context. Define $\forall k > 1$:

$$\Sigma^k = \{a\omega | a \in \Sigma,\ \omega \in \Sigma^{k-1}\}$$

Define Σ^*, the set of words (*strings*) as:

$$\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k$$

A language L (over Σ) is a set of words, $L \subset \Sigma^*$. The complement of a language L, denoted $\bar{L}$ is the unique language satisfying $\Sigma^* = L \bigcup \bar{L}$. Given strings $x, y \in \Sigma^*$ we denoted the concatenation of the two strings as the string $xy \in \Sigma$. We denote the i^{th} prefix of a string $x \in \Sigma^*$, the first i characters, as $(x|_i$. We denote the i^{th} suffix, everything but the first i characters, as $|x)_i$. Thus $x = (x|_i|x)_i$.

We now define Deterministic Finite Automata (DFAs) and Non-Deterministic Automata (NFAs). Later we define Labeled Transition Systems which generalize both. From a theoretical view a DFA is a labeled Markovian finite deterministic dynamical system. By dynamical system we mean it changes state over time, where the possible states are finite. By Markovian we mean the change is dictated only by the current state and not the history of previous states. Additionally the transitions between states are labeled with characters from an alphabet Σ. By deterministic we mean every transition leaving a state is labeled uniquely. DFAs are usually presented as the five-tuple $\{S, \Sigma, T, s_0, F\}$ where we denote the set of all subsets of S as $\mathcal{P}(S)$:

S is a finite set of states

Σ is a finite alphabet

T is a function T:(S,Σ) $\rightarrow \mathcal{P}(S)$, with the restriction that for fixed state $s_i \in S$ and fixed $a \in \Sigma$, T(s_i, a) is a set with one element.

s_0 is an initial state, $s_0 \in \Sigma$

F is a set of final states, $F \subset S$

An NFA is the same five-tuple without the restriction on the transition function T. A string $\omega \in \Sigma^*$ traces a unique path through a DFA, and a set of paths through and NFA. If any of these paths end in a final state in F, we say the automaton accepts that string. The set of strings a DFA or NFA accepts defines a language; we say an automaton *recognizes* that language. For a DFA or NFA A, we denote the language it defines as L(A).

Proposition 1.2.2. *NFA are no more powerful then DFA, in that the set of languages NFAs recognize is the same as the set DFAs recognize.*

This is proved by providing a conversion that can turn any NFA into a DFA. We look at all subsets of the set of states (finite). Every prefix of a string will end up in some set of states of an NFA, a configuration. Since only this configuration will determine the next possible configuration (Markovian), and for each prefix the configuration is unique (deterministic): we can make a DFA out of the transitions of configurations an NFA goes through. Note an accepting configuration contains an accepting state. If the size of the set of states is denoted $|S| = n$, then the size of the set of all subsets of S is denoted $|\mathcal{P}(S)| = 2^n$. Thus a DFA constructed this way could be exponentially larger then the original NFA, where the size is measured in the number of states.

Proposition 1.2.3. *The minimum DFA accepting a Language is unique. The minimum NFA is not unique.*

We must note the minimal NFA is possibly much smaller then the minimal DFA. Since A DFA is an NFA, it is certainly no larger. We can minimize DFAs by combining states that behave the same on all strings. Minimization of DFA is related to the Myhill-Nerode characterization of DFAs seen in Chapter 2. Chapter 3 introduces the notion of bisimilarity; there is a unique minimal NFA bisimilar to the original NFA. In Chapter 3 we present minimization of NFAs in this context; it appears minimization is important as sensible learning algorithms would learn minimal targets. Is there a relationship between the two? We will attempt to answer such questions.

1.3 Models of Learning

To soundly analyze and reason about an algorithm's ability to learn, we need to first provide a rigorous model of what it means to learn. Then any claims made about the difficulty of learning one problem versus another can stand up to scrutiny. This, perhaps is an obvious statement, but one worth making as it highlights our goals, and the challenges in achieving them. Primarily there are several overriding philosophical quandaries encumbering our task. Among these are your typical epistemological questions: 'What does it mean to learn a concept?', 'Can we ever know if we have mastered learning a concept?' and 'Which concepts are difficult to learn?' We have partly answered that last question. If we make the assumption that $P \neq NP$ then we can take as easy those learning algorithms with polynomial running times, and as hard, those which are NP-hard.

The answer to the first question is where people might disagree; however the general consensus is that in machine learning, algorithms are given examples of some larger concept which it must eventually deduce –perhaps too simple a definition of learning for

some. Let us imagine ourselves in the role of a learner: we exist in a universe clearly bounded by the limitations of human perception. The units of our universe are the objects we perceive as being distinct, among which some share common characteristics allowing us to group them in some logical framework. We call these logical groupings concepts, and we can represent them by the subset of the universe they group together. To learn a concept is to then be able to determine if a given element of the universe belongs to that subset. This is not to say learning consists of memorizing this whole subset, nor is it to say learning consists of axiomatizing this subset –it is any method that can accurately decide if a given element belongs to a set not wholly known in advance. This is why we talk of the predictive power of learning. We will use this thought experiment to motivate the standard model used in learning theory. It is worth repeating: we have described a prediction oriented model of learning. Another prevailing notion equates learning with succinctly explaining a set of data, the distinction being that the emphasis is put not on the predictive power of a hypothesis, but on the size of the hypothesis. (Blumer [4]).

We now formalize the above thought experiment.

1.3.1 Model Basics

Regardless of the model, any learning algorithm follows a general cycle of receiving examples and counter-examples, which it in turn uses to update an internal hypothesis (for a more detailed overview see [2]). When the algorithm has enough confidence that the hypothesis is correct or near correct, it terminates. We call the set of instances of objects that can serve as examples or counter examples the instance space, and denote it as the set $\mathcal{U}$, the universe. The goal of the algorithm is to learn some subset of the instance space, which we call the concept, $c \subset \mathcal{U}$.

As an example, the instance space could be the set of all words over some alphabet Σ. A concept could then be any language; however designing an algorithm to learn any arbitrary language is far too difficult a task. Because of this we generally require that the concept is part of some larger collection with well-defined constraints on it; for example, that it be a regular language. We call this collection the concept class $\mathcal{C}$, which is a set of subsets of $\mathcal{U}$. Then the job of the learning algorithm is to be able to develop a hypothesis for any fixed $c \in \mathcal{C}$. Requiring an algorithm to learn any concept in the concept class may be too arduous a task.

We can equivalently think of the concept c as a function from the instance space to the set $\{0, 1\}$, a function which serves to classify elements of the instance space as examples (1) or counter-examples (0), of the target concept. We can then think of the concept class as a family of functions. This is a convenient view of the concept, because it allows the learning algorithm to have oracle access to the function $c : \mathcal{U} \to \{0, 1\}$. Interpreting c as a subset of $\mathcal{U}$ or likewise as a function over $\mathcal{U}$ are interchangeable ideas, and we will refer to concepts c in both contexts, where no confusion arises.

Finally we formalize what we have all but said, and that is a learning algorithm is efficient if it runs in time polynomial in the length of any input.

1.3.2 Representation Schemes

By representation scheme we refer to the choice of how to encode our concepts and our hypothesis. Choosing an encoding is perhaps a minor technicality, but the efficiency of the learning algorithm depends greatly on the choice (see Pitt [16]). It is often easy to overlook this important distinction as the concept classes are often defined in terms of their representation, but not necessarily. Generally a representation scheme is a function

that maps encodings to the elements of the concept class they are meant to encode. For example $\mathcal{F} : \{0,1\}^* \rightarrow \mathcal{C}$, would be a representation scheme that relates binary encodings of concepts to the actual concepts they represent. Poor choice of representation schemes could result in the intractability of an otherwise efficient learning algorithm; it would be unwise to represent regular languages as an explicit list of all the words in the language, as opposed to a table encoding a DFA.

An even more subtle point is that the scheme chosen to encode the concept need not be the same as the one chosen to encode the internal hypothesis. For this reason we define similar to concept classes, a Hypothesis Class $\mathcal{H}$, which also represents a collection of subsets of $\mathcal{U}$. This allows a possible separate representation for the hypothesis; the paper by Pitt [16] gives an explicit example where changing the representation scheme for the hypothesis changes an intractable learning algorithm into an efficient one. Like concepts, hypotheses can also be viewed as functions over $\mathcal{U}$.

For each representation we also want to allow a notion of size. So for example if $\beta \in \{0,1\}^*$ is a representation, we could denote the size of β by its length, $|\beta|$. Furthermore, where applicable we may want to parameterize the concept and hypothesis classes by their sizes as well. So for instance if we let $\mathcal{C}_n = \{0,1\}^n$ we then get:

$$\{0,1\}^* = \mathcal{C} = \bigcup_i \mathcal{C}_i \tag{1.1}$$

Parametrization of the hypothesis class can proceed in an analogous manner. In most cases the representation scheme will be implicit, though for sound and meaningful analysis we must ensure that we are not mixing representation schemes in the same algorithm.

From these basic concepts of modeling learning we present in the next few sections some explicit models for which results exist.

1.3.3 Assisted Learning

In this model the goal of the algorithm is to learn some concept c from a concept class $\mathcal{C}$. Internally the algorithm will be developing some hypothesis h from the hypothesis class $\mathcal{H}$. The algorithm will have oracle access to the function c. An oracle is a black-box function, that is we do not know how it works (black-box) and it takes O(1) time to compute, as if it already knew the answer (hence the term oracle). The algorithm may select any $x \in \mathcal{U}$ and the oracle will return $c(x)$. These are traditionally called *Membership Queries*. This model has an additional query we allow, called the *Equivalence Query*. This notion of learning is due to Angluin, but our presentation of it is due to Kearns and Vazirani [12].

For the Equivalence Query, the algorithm can give this oracle its current internal hypothesis h, and in return the oracle provides an $x \in \mathcal{U}$ such that $h(x) \neq c(x)$, if such x exists. Assisted learning algorithms terminate when the equivalence-oracle can no longer provide such an x, otherwise the algorithm will continue to refine h. We can think of membership and equivalence oracles together as acting like a teacher.

An important question to raise is whether the teacher is malicious; it is not hard to imagine an instance where the equivalence-oracle could provide a counter-example exponentially larger then the concept that is being learned. This is not possible in all problems and not necessarily always a concern. The size of the counter-example is certainly a lower bound on the running time of the algorithm, as at the very least the learning algorithm must read it. With a adversarial oracle we could feed an unnecessarily long counter-example to the learner, making the learning task appear intractable. For this reason we generally assume that the oracles play fair —we can expect a reasonable bound on the size of the

counter-example, obviously dependent on the problem. This is not the same as saying that the teacher is helpful. We make no other assumptions other than correctness and reasonable succinctness.

We now define when a concept class $\mathcal{C}$ is *efficiently learnable using assistance*.

Definition 1.3.1 (Efficient Assisted Learning)**.** *An algorithm A with access to membership and equivalence oracles is an Efficient Assisted Learning algorithm for a concept class $\mathcal{C}$, if for any fixed $c \in \mathcal{C}$ (where the implicit representation of c has size n) A outputs in time polynomial in n, a hypothesis h such that $\forall\ x \in \mathcal{U}, c(x) = h(x)$.*

We say a concept class is efficiently learnable using assistance if such an algorithm exists.

The definition can be modified so that the running time is polynomial in any other reasonable problem dependent input parameters, the length of the counter-examples for instance. This model of learning is often called MAT learning for minimally adequate teacher.

There is a notion of reductions for Assisted Learning problems, denoted $\prec_{MAT}$, due to Angluin and Kharitonov [3]. However, as presented their reduction is for a slight variation on assisted learning.

1.3.4 PAC Learning

This notion of learning is due to Valiant, but our presentation of it is again adapted from [12]. PAC learning has a more bleak world view for modeling learning. The obvious difference is that the algorithm is no longer being assisted; there is no teacher. The first implication is that without an equivalence oracle we can never have absolute certainty

about whether the internal hypothesis is correct. Furthermore, access to the membership oracle is also restricted in that the algorithm can no longer provide an $x \in \mathcal{U}$ and receive a classification $c(x)$ for a fixed $c \in \mathcal{C}$. Rather, the oracle, upon request, selects a random x (according to some unknown but fixed distribution $\mathcal{D}$) and then the algorithm receives the pair $\langle x, c(x) \rangle$. The power this singular oracle provides the algorithm is actually closer to that of the equivalence oracle of Assisted Learning than to the membership oracle (see Section 1.4). Thus we really have a wholly new oracle which we call the Example Oracle. This Example-Oracle creates several problems: for one, our algorithm could be unlucky enough to keep receiving pairs $\langle x, c(x) \rangle$ for the same x, or some small set of x's. Moreover, the distribution from which the oracle selects x is unknown and possibly far from uniform, perhaps giving undue importance to certain facets of the target. This is remedied by having the algorithm maintain two internal parameters ϵ, δ in addition to the internally maintained hypothesis h.

We call ϵ the error parameter and we define it as follows:

$$\epsilon = \Pr_{x \in \mathcal{D}}[c(x) \neq h(x)] \tag{1.2}$$

Note that as h is updated ϵ changes and, hopefully, is getting smaller. The advantage of this is that the error is being measured with respect to the distribution, so if an x is unlikely to be chosen in the underlying fixed distribution, and our algorithm fails to capture that facet of the concept, we do not consider that as contributing much to the error. Now the algorithm can never actually compute ϵ since $\mathcal{D}$ is unknown; the best that it can hope for is an upper-bound. We include as input to the algorithm the constant E representing the tolerable error; we require the algorithm to output an h such that $\epsilon < E$. This allows us to remedy the second problem. However, the primary problem of being so unlucky as to

draw a terrible sample still remains, which is where the parameter δ comes in.

We call δ the confidence parameter, and we define it as follows:

$$\delta = \Pr[\epsilon \leqslant E] \tag{1.3}$$

where the probability is taken over any randomness in the algorithm including the calls to the oracle. Again there is no way to calculate this value; the best we can hope for is an upper bound. We include as input to the algorithm the constant Δ, representing the desired confidence; we require the algorithm to return an h such that $1 - \delta \geqslant 1 - \Delta$.

The formal definition is:

Definition 1.3.2 (PAC learning)**.** *An algorithm A with access to an Example Oracle is a PAC learning algorithm for a concept class $\mathcal{C}$ if for every $c \in \mathcal{C}$ and for any fixed distribution, and for all $0 < E < 1/2$ and $0 < \Delta < 1/2$, the following holds: if A is given inputs E and Δ then with probability at least $1 - \Delta$, A computes a hypothesis h, whose amount of error in modeling the target is no more then E.*

1.4 Known Results

As we have noted, the most relevant result to this thesis is Angluin's result on the Assisted Learning of DFAs [1]. Chapter 2 presents an explanation of her algorithm. This result for an Assisted Learning algorithm should be contrasted with Pitt and Warmuth's negative result on the approximation of DFAs which we mentioned at the start of the chapter. They show in [17] that assuming $P \neq NP$ there is no polynomial time algorithm for computing a DFA whose number of states is polynomial in the number of states of the minimum DFA consistent with some finite collection of labeled examples of a language. This certainly makes PAC learning DFAs seem hopeless. In fact, Kearns and Valiant showed that

under some assumptions about the security of some well known cryptographic protocols, that DFAs are not PAC learnable [11]. Similar cryptographic assumptions were used by Angluin and Kharitonov to show that Assisted Learning of NFAs is not possible in polynomial time [3]. In contrast there is Yokomori's Assisted Learning algorithm for NFA [21] based on a relaxed version of the problem; albeit the relaxation seems to make the problem trivial. We discuss all these results in more depth, using them to inform our work on learning LTS. There is one previous result on learning LTS [8], however the paper only considers a type of LTS that can be built from DFAs, and uses Angluin's algorithm to learn that subtype. The remainder of that paper is on a different topic.

The insights into learning theory that can be gleaned from these papers is significant and provides important circumstantial evidence about the power of learning algorithms –especially with regards to LTSs. In chapter 5 we will present a simple schema for reductions from DFAs to LTSs which will allow us to trivially extend previous results. As Figure 1.1 shows such reductions will transform DFAs into what we will call *pseudo*-DFAs or pDFAs. These reductions are trivial; the existence of accepting states is the only difference between LTSs and DFAs –the reduction simply models *acceptance*-behaviour with the modal logic we will use to describe LTSs when conducting queries (section 5.1).

For now we introduce three problems —all very similar, all related to learning— and examine the implications of their tractability. These problems are meant to summarize the insights from previous work, and have been presented before under varying guises. The first problem is from the Pitt and Warmuth paper [17]. We formalize it now.

Problem 1a: Create_DFA(*POS,NEG,n*):

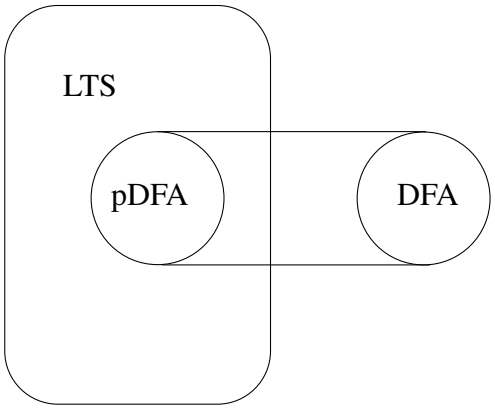


Figure 1.1: Relating DFAs and LTSs

Input: A set POS of strings in some fixed Language L, and a set NEG not in the language L. An integer n.

Output: A DFA with n states that accepts all strings in POS and none in NEG

Problem 1b: Create_DFA(*POS,NEG*):

Input: A set POS of strings in some fixed Language L, and a set NEG not in the language L.

Output: the minimal DFA that accepts all strings in POS and not in NEG

We add the caveat that the cardinality of the set $POS \bigcup NEG$ be polynomial in the size of the DFA we are learning; the algorithm at least must examine each item, providing a lower-bound on the running time. Likewise, we assume that the number of characters in any string in the set $POS \bigcup NEG$ is polynomial in the size of the target DFA.

Gold showed in [9] that Problem 1a was NP-hard, using a broader definition of NP-Hard that includes more than just decision problems. Notice that as n gets larger the

problem gets easier. It is trivial to construct a DFA that accepts exactly the strings in POS if we do not care about the size of n –start by building an NFA. It is even more trivial when NEG $= \emptyset$: return the DFA that accepts all strings. It follows that Problem 1b of finding the minimum such DFA is at least as hard, because knowing the number of states in the minimal DFA would easily allow us to answer the former problem. Gold's proof is interesting as it restates the problem as a transition table filling algorithm: constrained by the determinism of a DFA, and the partial information provided by the sets POS and NEG –we need to compute a filling of this table that satisfies these constraints. This superficial similarity to SAT may lead one to correctly guess that the problem is NP-Hard.

Theorem 1.4.1. *Create_DFA(POS,NEG) cannot be approximated within any polynomial factor.*

Our focus now is to relate this Create_DFA problem with Learning. Notice first that Create_DFA is almost the same problem as PAC-Learn_DFA, where $POS \bigcup NEG$ is the set of labeled counter-examples given to a PAC-Learning algorithm over its execution. If we had an algorithm to PAC learn DFAs, say PAC-Learn_DFA, we could use it to solve Create_DFA by acting as its example oracle. Recall PAC algorithms have access to an oracle, the example oracle, that gives them access to *examples* of the target concept; further recall PAC algorithms take no input.

Algorithm 1.4.2. Create_DFA(*POS,NEG*):

—Begin execution of algorithm PAC-Learn_DFA()

—For each query PAC-Learn_DFA makes to the

Example Oracle pick a *new* element ω from $POS \bigcup NEG$,

if a new one exists, otherwise pick any element

— if $\omega \in POS$, send (ω, YES) as the answer

to the oracles query

— else send (ω,NO) as answer

—else if new elements of $POS \bigcup NEG$ have been exhausted

return to the oracle any element of $POS \bigcup NEG$

—When PAC-Learn terminates return the DFA it returns.

We can also relate it to Assisted-Learning. In this vein we introduce a second problem which we will use to show the relationship between the problem Assist-Learn_DFA() and Create_DFA(*POS,NEG*). Recall that assisted learning algorithms have access to a membership oracle and an equivalence oracle. The problem is Predict_DFA, and it is a minor variation on Create_ DFA. It is also a decision problem and helps relate learning algorithms to traditional algorithms.

Problem 2: Predict_DFA(*POS,NEG,* ω):

> **Input:** A set POS of strings in some fixed Language L, and a set NEG not in the language L. A sting ω.
>
> **Output:** Yes if ω is accepted by the minimal DFA that accepts all strings in POS and none in NEG. Otherwise outputs no.

On one hand Predict_DFA might seem easier then Create_DFA because the problem is asking less, and in fact Predict_DFA $\prec_r$ Create_DFA since:

> Predict_DFA(POS,NEG) := Yes $\leftrightarrow$ ω is accepted by Create_DFA(POS,NEG)

It is not obvious though if Create_ DFA $\equiv$ Predict_ DFA. The point of Predict_ DFA() is that it models the membership queries of Assist-Learn_ DFA(); if it were equivalent to Create_ DFA then the power of the learning algorithm would come from oracle access to solutions to an NP-hard problem. We show how to use Predict_ DFA to solve Create_DFA() by intercepting queries to the oracles:

Algorithm 1.4.3. Create_DFA_2:

—Begin execution of algorithm Assist-Learn_DFA

—For each query Assist-Learn_DFA makes to its

Equivalence-Oracle with hypothesis h do the following:

—For each element ω_i from POS

test if ω_i is accepted by h

— if no send (ω_i,YES) as answer

—For each element ω_i from NEG

test if ω_i is rejected by h

— if no send (ω_i,NO) answer

— If nothing sent to oracle return h as DFA

—For each query ω Assist-Learn_DFA makes

to the Membership-Oracle

—return Predict_DFA(*POS,NEG,* ω) as the answer

This shows that the power of the Equivalence-Query in MAT-Learning is most akin to the oracle of PAC-Learning, whereas the Membership-Query is what provides additional power to assisted-Learning algorithms.

1.4.1 Cryptographic Limitations on Learning

This result by Kearns and Vazirani [12] shows cryptographic limitations on learning. They show that learning DFA under the PAC model would be tantamount to breaking cryptographic functions that have long been assumed to be secure. We already mentioned that there was a natural relation between learning and cryptography, and is easy to explain. The actual reductions are quite complicated though.

Most cryptographic protocols are built around the idea of one-way functions. Let us say a person, BOB, wants to set up a secure channel. Here is one method for achieving that goal. BOB can invent a function f that satisfies two simple properties:

1) f is invertible (a bijection). Thus there exists an inverse function f^{-1} such that $f^{-1}(f(x)) = x = f(f^{-1}(x))$

2) Unless given f^{-1}, said function is hard to compute given only f

This way, to establish a secure channel, BOB publishes f in a directory. Anyone who wants to send him a message x just computes $f(x)$, and sends that. Since BOB has not published f^{-1}, only he can efficiently compute $f^{-1}(f(x))$. Everyone else would have a difficult time. However, by publishing f any other person, say ALICE, can compute pairs $< x, f(x) >$ by themselves. Looking at it another way: $< x, f(x) >$ = $< f^{-1}(y), y >$ for y = $f(x)$. This consist of examples of the inverse function. So, if we had a PAC learning algorithm that could learn the function f^{-1} we could feed the pairs $< x, f(x) >$ to the

algorithm and it would learn f^{-1}. Thus if learning f^{-1} is easy this implies we can invert f easily. In particular Kearns uses the RSA encryption scheme to show that PAC-learning DFA implies computing the inverse of the RSA encryption scheme. For many years it has been assumed that RSA is secure in the sense that it resists computing this inverse. This would imply PAC-learning DFA is hard.

1.4.2 Yokomori's Paper

Yokomori's paper [21] provides an algorithm to learn NFAs in time polynomial in the size of an equivalent DFA. This is a minor result, since DFA are themselves NFA technically. He argues that NFA are often a more pleasing and efficient representation of regular languages — a human would find an NFA more instructive then an equivalent DFA. However this is only true if the NFA are an order of size smaller then the equivalent DFA. Yokomori's NFA are not smaller by any significant order. The paper raises two interesting ideas, first:

Can learning algorithms be used to assist design?

A scenario we can imagine for a learning algorithm is one where we have it assist a human designing a target concept. There are two ways to go about this. In the first, we could have a PAC algorithm, and provide it with examples of our target. The hypothesis returned may assist us in evaluating our target.This is not very helpful to a designer who makes an error in their design specification. That a PAC algorithm's correctness is judged based on the the distribution of the representative sample frustrates this issue more.

The problem is solved by instead using an Assisted-Learning algorithm. In this way, a flawed or incomplete design specification can be corrected or enhanced by forcing the designer to answer the membership queries. It is in this way that learning algorithms could

aid design: by helping to illuminate confusing or underdeveloped designs.

The second interesting idea is this:

Can learning algorithms output hypotheses which are aesthetically pleasing?

Yokomori argues NFA better represent regular languages for humans. This raises the question of how effective various hypotheses are. For instance, our model of learning only cares about the final hypothesis. One metric for how aesthetic a hypothesis is could measure how efficiently it conveys information. This is more of a Human Computer Interaction topic —but an interesting one none-the-less. One 'obvious' requirement is that the hypotheses should resemble any target which has yet to be ruled out by any queries or counter-examples. Though this would be true of any sensible learning algorithm, as it is true of Angluin's algorithm, we can easily alter any algorithm so that it is not true. To complicate matters, Angluin's algorithm, as we will see, outputs hypotheses which do not include much of the information it has learned. This will be evident in the next chapter, but consider all the internal data the algorithm uses to design the hypothesis versus the information contained in the hypothesis. It would be interesting to consider designing hypotheses which demonstrate all the partial information that an algorithm has learned.

1.4.3 Does Minimization Imply Learning?

When we judge a learning algorithm's efficiency, we often do so in the size of the minimal equivalent target. For instance, we can learn a regular language in the size of the minimal DFA that accepts it, but not the minimal NFA. A question to ponder then is, does this suggest a correlation between minimizing something and learning it? In some ways they are opposite processes. Minimizing starts with a target with redundant aspects; our goal is to identify those aspects and remove them. To achieve this we must learn what parts of our

target are fundamental. A learning algorithm starts from nothing, and tries to identify only the fundamental parts of the target. Any time spent learning redundant aspects is wasted, and could lead to the algorithm being inefficient. Consider the following reduction:

Algorithm 1.4.4. Minimize_NFA(N):

—Begin execution of Assist-Learn_NFA

—For each query to the Equivalence-Oracle by

Assist-Learn_NFA with hypothesis H:

—test if N$= H$, if not return counter-example

—For each query x to the Membership-Oracle

Assist-Learn_NFA:

—test if N accepts x, return answer

—return H when execution ends.

Let us go back to the case of DFAs and NFAs, and let us consider the difference between minimal forms of both. In NFAs, non-determinism seems to allow what we shall refer to as *exponential compression*. Pick a fixed regular language L where the size of the minimal DFA accepting that language is exponential in the size of the minimum NFA, m. Thus we can imagine that there exists a string $x \in L$, such that the number of steps it takes the DFA to recognize it is exponentially larger then the number of steps it

takes the NFA. This is because the computation model for NFAs allows us to consider the possible exponential fan-out in parallelism of the computation of whether the NFA accepts x, and reduce its size to the length of single path leading to an accepting state. Thus any learning algorithm would have to learn this exponential compression using tools designed for learning DFAs. Minimization of NFAs is hard, as is learning NFAs under any model, whereas we can efficiently minimize DFAs, and we can learn DFAs under the Assisted-Learning model. This sussing out of non-determinism is a fundamental issue in computer science, and understanding its nature is one of the central questions that motivates this research.

1.5 Motivation

Academic papers often avoid lengthy speculative discussions for good reason —they lack rigour. What purpose they do serve, however, is to provide context to the current research, and to let the reader understand the world as the author sees it.

Let us then take a pause before we start looking at the technical details to consider why we want to study learning theory. The previous section tried to give a flavour of the known results about the tractability of learning some concepts. How are we to interpret these? Beyond the obvious implications to Artificial Intelligence, the algorithms may inform us about our selves, humans. We too are agents of learning —how do we represent concepts? Form hypotheses? What problems are easy for us? The obvious answer is that problems which take polynomial time and space to compute would also be easy for humans to compute, using the same algorithms —tedium notwithstanding. Moreover we study learning theory because we want to understand the intrinsic properties of problems

that make them hard. Why do some problems seem to require more time to solve? More possibilities to check? Again the obvious answer is information. Understanding how information is conveyed should be seen as a cornerstone of computer science.

One of the goals of complexity theory is to be able to examine a problem and uncover some combinatorial property that guarantees the algorithm cannot be solved in polynomial time. We are probably not close to achieving this goal, and there is no reason to believe it is possible. The issue is that all the problems we consider in Computer Science have a multitude of different models and descriptions, and any presuppositions we make concerning problems may have great impact on their tractability. This seems to bar us from creating some combinatorial property that could be used to identify common characteristics of all these problems. Yet, unspoken is a notion that every problem contains some amount of intrinsic complexity. That is, to solve a problem we seem to always have to uncover a certain required amount of information. What is great about the learning model of algorithms is that the idea that information is being uncovered by an algorithm is made explicit. We can see the teacher as an adversary, only giving the learning the minimal amount of information about the target, or more sinister yet: changing the target based on the learner's hypothesis to force more work. In that case the strategy of the learner is to make the minimal change to the hypothesis, so as not to have to undo work later. It seems the changes a query induces upon a hypothesis could define the smallest discrete amount of information about a target, and query complexity could lead to a metric of the complexity of the target.

Chapter 2

Angluin's Learning Algorithm

Angluin's algorithm is an assisted learning algorithm for learning DFAs (Section 1.2.3). Equivalently we can think of it as an algorithm to learn a regular language. It uses two oracle queries:

- The membership query, denoted membership$(\cdot)$, which asks if a string is accepted by the target DFA

- The equivalence query, denoted equivalence$(\cdot)$, which asks if a hypothesis DFA is correct. If not the oracle returns an example of a string the DFA misclassifies.

There is a third way to view Angluin's algorithm. This is the notion of viewing learning as a series of refinements of an equivalence relation, converging to some target relation. In the case of DFAs, the equivalence relations we will be refining are what we call Myhill Nerode Relations. It is this view we take; the benefit being that Myhill Nerode Relations provide simple conditions for when a relation is associated with a language.

This leads naturally to the first obstacle we face, the need to store our hypothesis as an equivalence relation. If we can achieve this, then by assuming the hypothesis relation is a Myhill Nerode Relation (MNR) for a fixed regular language L, we will be able to use membership queries to turn our hypothesis from an equivalence relation into a DFA, which is what we really want to output. The transformation we use must have the property that if our hypothesis relation were a MNR for the fixed language L, then the DFA we build would accept L. Thus, if we give that DFA to the equivalence oracle, in the case

that our hypothesis was not a MNR for L, we will get a counter-example. Moreover, our transformation will be invertible, thus we can use that counter-example to correct our hypothesis by updating the equivalence relation. There are two questions to keep in mind:

- Since our hypotheses are now relations, how should we store relations?

- When we assume the relation to be a MNR, how will we decide which equivalence classes strings belong to?

These questions are important because the number of strings over an alphabet are infinite. We need a finite procedure to classify any given string because we cannot know the whole relation. Here is the strategy we will use. We store two sets of strings. The first set we call the *Access* strings. They represent the known states of the DFA; or, equivalently canonical elements of the known equivalence classes of the associated MNR. The answer to the second question, is the second set: the *distinguishing* strings. For each pair of access strings x_i and x_j, we will keep a distinguishing string $\delta_{i,j}$ such that only one of $x_i\delta_{i,j}$ and $x_j\delta_{i,j}$ is in the language L. We will see we can use these distinguishing strings as one way to put any string x in an equivalence class, that of the canonical string $y \in$ Access, such that membership$(x, \delta_{i,j})$ and membership$(y, \delta_{i,j})$ agree. We call this process picking the best-fit equivalence class. Thus there are three components to remember when considering the hypothesis relation:

1) The access strings y_i forming the known states of the target. Equivalently forming canonical elements of the known equivalence classes $\{[y_1] \dots [y_n]\}$.

2) For each pair y_i, y_j of canonical elements, a distinguishing string δ, such that only one of $y_i\delta$ and $y_j\delta$ is in the language L.

3) A procedure to compare any string $x \in \Sigma^*$ with the known distinguishing formula and pick a *best-fit equivalence class* among the known $[y_i]$'s, to place that string.

We will see how all three ideas intertwine to achieve Angluin's algorithm. The next sections explain these ideas in more detail. Section 2.6 contains a worked example followed by the algorithm in section 2.7. One thing to watch out for is this: to show how to update the hypothesis we need to define finding the best-fit equivalence class, yet to define this we must first have updated the hypothesis once. This may seem like circular reasoning, however to overcome this we define how to do the initial update to create the first and second hypotheses separately from how to do the actual updating. Then all we have to do is show we can update any hypothesis on the assumption we can define a best-fit procedure. Finally we show how given a correct initial update we can define such a procedure. What we have just outlined is the broad idea. To achieve it we must begin by defining Myhill Nerode Relations.

2.1 Myhill Nerode Relations

In this section we lay the groundwork for Angluin's algorithm by recalling this famous theorem.

Definition 2.1.1 (Myhill Nerode Relations)**.** *Fix a language L. Let* $\equiv_L = \{B_1 \dots B_n\}$ *be a partition over an alphabet* Σ*. Then* $\equiv_L$ *is a Myhill-Nerode Relation (MNR) for L if it satisfies the following:*

(i) if $x, y \in B_i$ *then* $\exists j$ *such that* $xa, ya \in B_j, \forall a \in \Sigma$

(ii) n is finite

(iii) $\equiv \sqsubset \{\{x|x \in L\}, \{x|x \notin L\}\}$

These MNR are equivalence relations over languages; and account for the regular languages. We can map every MNR to a unique DFA, and map every minimal DFA to a unique MNR.

Theorem 2.1.2 (Myhill-Nerode)**.** *Let Σ be a finite alphabet. Up to isomorphism $\exists$ a bijection f between DFAs over Σ accepting the language L and MNR for L on Σ^*.*

Proof. To design the bijection f :MNR$\rightarrow$DFA for $\equiv \in$ MNR ($\equiv= \{B_1 \dots B_n\}$) we use the following procedure:

For each B_i we create a state s_i for the DFA. Since n is finite, we will have a finite number of states. If $x \in B_i$ and $xa \in B_j$ then f puts a transition between states s_i and s_j and labels it 'a'. Condition (i) of Definition 2.1.1 ensures that these transitions will be deterministic. Since $\equiv \sqsubset \{\{x \in L\}, \{x \notin L\}\}$ those blocks which are subsets of $\{x \notin L\}$ become non-accepting states. Those that are subsets of $\{x \in L\}$ become accepting states. The block B_i containing ϵ becomes the start state. Note that any string, x, that puts DFA $f(\equiv)$ into a final state will be an element of L; to see this consider all the equivalence classes each prefix of x is in. Clearly f is a bijection, since for each state of a DFA we can create a block and insert into it every string that leads to that state. By the determinism of the DFA a string will only be put in one block, and no block will contain a mix of accepting and non-accepting strings. Because the DFA is finite and deterministic, and accepts only strings in L, a partition built this way will be a MNR. □

Corollary 2.1.3. *The implication of the above theorem is $\exists f, f^{-1}$ such that f^{-1} :DFA$\rightarrow$MNR and f :MNR$\rightarrow$DFA. This means a MNR exists for a language if and only if it is regular.*

Furthermore, for some fixed but unknown MNR there exist the same fixed and unknown function f, that computes the above transformation.

In fact we would like to think of DFAs and their underlying equivalence relations interchangeably. The modern definition of DFA we gave earlier, as a set of states and transitions, while quite practical, does not serve us well for our purposes. We should instead think of DFA as graphical representations of MNRs.

Another important corollary of the Myhill-Nerode Theorem is that states can be identified by their behaviour on prefixes of strings in the language L. To see this, inductively expand condition (i) of Definition 2.1.1 to include strings of length $k > 1$. If for two states, the set of strings that took those states to accepting states were identical, then the two states could be merged without changing the language the DFA accepted. This leads to the following important observation:

Corollary 2.1.4. *In a minimal DFA, for any two distinct states reachable with strings x and y respectively, there exists a string z such that only one of xz and yz is in the language L.*

2.2 Sampling the Partitions

In this section we outline Angluin's algorithm's key idea, sampling a partition as if it were a MNR.

For a fixed but unknown language L, which has DFA A, and an alphabet Σ; the goal of the algorithm is starting with:

$$\equiv_1 := \{\{x | x \in L\}, \{x | x \notin L\}\} \tag{2.1}$$

find a sequence of refinements:

$$\{\Sigma^*\} \sqsupset \equiv_1 \sqsupset \equiv_2 \sqsupset \ldots \sqsupset \equiv_i \sqsupset \equiv_L \tag{2.2}$$

Equation 2.2 says *two* things.

1) Each hypothesis relation is a refinement of the previous

2) The target MNR refines all our hypotheses.

Let us recall the broad outline of the algorithm we gave in the introductory section. First of all, assume we have some way of storing each equivalence class (the hypothesis). Then, for each block we have canonical elements. Furthermore, when given a string x, not a canonical element, we can decide which block it belongs in. Call this block, x's best-fit equivalence class. We will explain how to achieve this in the Section 2.5.

Secondly, since this is an assisted learning algorithm we can assume we have access to two types of queries. At the heart of the algorithm is how to use the counter-example from the equivalence query to make a refinement as in Equation 2.2. The goal is to learn a target DFA A. Since the equivalence query expects a DFA as a hypothesis, but the algorithm is working in terms of equivalence classes, we need a way to turn an equivalence relation into a DFA —recall if the equivalence relation were a MNR then the function f of corollary 2.1.3 serves this purpose.

Here is what we do: at each step we assume the equivalence relation, as we know it, is correct in fully describing the DFA we intend to learn —that is, it is a MNR for L. We can construct a hypothesis DFA from the partition, by sampling canonical elements from each block of the partition. We define a function we call the g-function that does this sampling. The g-function approximates the fixed function f, for a fixed MNR for L.

The g-function assumes the current hypothesis partition is a MNR for L. We use the new DFA derived from g to get a counter-example from the equivalence oracle. It should be noted that even though our hypothesis partition refines the target MNR, the language of the hypothesis DFA will not necessarily be a subset of the language of the target DFA. The reason for the disparity between the language of the target and the language of our hypothesis is caused because we are assuming each hypothesis partition is a MNR when they are not necessarily. However, we will ensure that our initial equivalence class is a refinement of $\{\{x \in L\}, \{x \notin L\}\}$, and that each step of the algorithm maintains this initial property when it computes the next hypothesis equivalence class. Also, it should be clear that conditions (ii) and (iii) of definition 2.1.1 will always be satisfied by our hypothesis' equivalence classes. The only way any of these partitions fails to be a MNR is by failing condition (i):

if $x, y \in B_i$ then $\exists j$ such that $xa, ya \in B_j, \forall a \in \Sigma$

That is why the g-function only samples the canonical element. If it could sample other elements it might send transitions with the same label to different states constructing an NFA. We now define the g-function we use to sample:

Definition 2.2.1 (g-Function)**.** *We define g over the domain of partitions* $\equiv = \{B_1 \dots B_n\} = \{[x_1] \dots [x_n]\}$ *for any n, which refine the partition* $\{\{x|x \in L\}, \{x|x \notin L\}\}$ *and where the* x_i *in the*$[\,]$*'s are the canonical elements.*

— *Create a state for each* $B_i = [x_i]$

— *Set as accepting states those that refine* $\{x|x \in L\}$

— *For each* $a \in \Sigma$ *the transition* $\xrightarrow{a}$ *from* B_i *is determined by figuring out* x_i*'s best fit equivalence class.*

For now we must assume that finding the *best-fit* equivalence class works. Let us again stress the difference between g and f —g constructs the transitions based on just one element of an equivalence class, ensuring the constructed DFA is deterministic. f does the same for an entire equivalence class and condition (i) ensures determinism. Had we used different elements, g may have put in different transitions precisely because the partition is not a MNR. Finally the canonical elements of the known equivalence classes are important because they represent elements of known equivalence classes in the target.

2.3 Computing an Initial Hypothesis

As noted, to force our hypothesis equivalence class to only fail condition (i) definition 2.1.1, we need to make sure of two things:

1) The initial hypothesis refines $\{\{x \in L\}, \{x \notin L\}\}$

2) As we update the hypothesis this condition is maintained

To ensure the second item we will design our procedure that updates hypotheses to work by splitting equivalence classes (Section 2.4). By this we mean if we have a block B_i in a partition that refines $\{\{x \in L\}, \{x \notin L\}\}$, then if we split B_i into two new blocks B_i^1 and B_i^2, this new partition still refines $\{\{x \in L\}, \{x \notin L\}\}$.

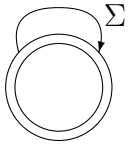
It remains to show how to create the initial partition. We use the following trick: begin by asking a membership query on the empty string ϵ. It will either be in the set $\{x \in L\}$ or the set $\{x \notin L\}$. If we find that $\epsilon \in \{x \in L\}$, we construct a DFA that accepts every string (see Figure 2.1). If $\{x \in L\}$ is not the MNR for L, we will be given a counter-example λ such that $\lambda \in \{x \notin L\}$. The case where we find that $\epsilon \in \{x \notin L\}$ is symmetric. That is

we construct a DFA which accepts no strings, and get a counter-example $\lambda \in \{x \in L\}$. Either way we have constructed an initial partition that satisfies our requirements:

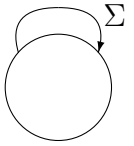
$$\{[\epsilon], [\lambda]\} \sqsubset \{\{x \in L\}, \{x \notin L\}\} \tag{2.3}$$

Initial DFAs:

if membership(ϵ)= YES:



if membership(ϵ)= NO:



Initial counter-example: λ

Initial Partition:

$\{[\epsilon], [\lambda]\} = \{\{x \in L\}, \{x \notin L\}\}$ or $\{\{x \notin L\}, \{x \in L\}\}$

Classes $[\epsilon], [\lambda]$ distinguished by ϵ

Figure 2.1: Initialization of Partition: two possibilities. Where λ is given counter-example and ϵ the empty string.

Further note that, as per Corollary 2.1.4, these two partitions are distinguished by the string ϵ, the empty string:

> Only one of $\epsilon\epsilon$ and $\lambda\epsilon$ is in L.

We need this fact to compute the best-fit equivalence class. Finally note that the target MNR relation refines this hypothesis by definition, and is also finite.

2.4 Updating the Hypothesis

When we get a counter-example to our equivalence query we are discovering a place where the f and g functions disagree. That is because if our hypothesis were correct then $g = f$. Essentially we can find a block $[y]$ of the current hypothesis equivalence relation such that $\exists x \in [y]$ but in the MNR x and y are in different equivalence classes. That is f *knows* x and y are in different equivalence classes; but, the g-function is assuming x is in the same partition as the canonical element y. The next hypothesis will no longer allow this assumption. It adds x as a new canonical element. Moreover, from the counter-example we will compute a string δ that distinguishes x from y. How? We know our current partition fails condition (i) definition 2.1.1. That is we know there exists a block B_i of our hypothesis such that for $x, y \in B_i$ there exists $a \in \Sigma$ such that xa and ya are not in the same block, say B_j, of our hypothesis. Let us say xa is in block B_j and ya in the block B_k. Further assume we have a string δ that distinguishes the canonical elements of blocks B_j and B_k. Such distinguishing strings were mentioned in Corollary 2.1.4. We can then refine the current partition by splitting block $B_i = [y]$ into two blocks: $[x]$ and $[y]_{new} := [y]_{old} \bigcap \overline{[x]}$ where the two new blocks $[x]$ and $[y]$ are distinguished from each other by $a\delta$ and distinguished from the other blocks by whichever strings previously distinguished the old $[y]$ block (this property is needed to correctly compute the best-fit equivalence class). What we still need to show is that this new relation is refined by the underlying fixed MNR for L; however, to prove this we need to know how to compute the Best-Fit equivalence class —it is the Best-Fit function which allows us to realize the access strings and distinguishing strings as an equivalence relation.

2.5 Computing the Best-Fit equivalence Class

Say we have a partition $\{[y_1] \ldots [y_n]\}$, y_i's $\in \Sigma^*, 1 \leq i \leq n$. If we can manage to maintain, for every pair of canonical elements, y_i, y_j, a distinguishing string $\delta_{i,j}$, then we can determine for a new string x, its best-fit equivalence class by the following procedure:

Definition 2.5.1 (Best-Fit equivalence class)**.** *Given a string, x, and a partition* $\{[y_1] \ldots [y_n]\}$ *with pairwise distinguishing strings* $\delta_{i,j}$ *we query the oracle for all i,j:*

membership($x\delta_{i,j}$*)*

The best-fit equivalence class of x, denoted $[\![x]\!]$*, is the block* $[y_l]$ *such that for all i,j:*

membership($x\delta_{i,j}$*) = membership(*$y_i\delta_{i,j}$*)*

The first question should be is this definition sound? What guarantees that a y_l exists such that membership($x\delta_{i,j}$) = membership($y_l\delta_{i,j}$)? The answer is that the above definition is sound only if the hypothesis partition was created in such a way described in Section 2.4. This is because this kind of refinement ensures that the underlying MNR relation still refines each successive hypothesis. Thus x is in some equivalence class of the MNR relations which in turn refines some block $[y_k]$ of the current hypothesis partition. That block is distinguished from all other blocks by some distinguishing string δ_k. Thus for all j:

membership($x\delta_{k,j}$) = membership($y_k\delta_{k,j}$)

And for every other block $[y_l]$, $i \neq k$:

membership($x\delta_{i,k}$) $\neq$ membership($y_l\delta_{i,k}$)

We have seen that correct computation of the best-fit equivalence class relies on the fact that the MNR still refines our hypothesis. However, to compute the necessary changes to a hypothesis we need to be able to compute the best-fit equivalence class in a sound fashion. They seem like cyclical requirements. Luckily, as we have seen, we can compute an initial hypothesis that is refined by the target MNR, where we have a correct set of distinguishing formula. This next theorem proves that after an update such hypotheses are still refined by the target MNR.

Theorem 2.5.2. *Given an equivalence relation* $\equiv_i \sqsubset \{\{x \in L\}, \{x \notin L\}\}$, $\equiv_i = \{[y_1] \dots [y_n]\}$, *refined by the target MNR, and a correct set of distinguishing formula to compute the best-fit equivalence class, then:*

if $\lambda = a_1 \dots a_m$ *is a counter-example to the DFA* $g(\equiv_i)$*, then we can compute a prefix* $(\lambda|_i = a_1 \dots a_i$ *such that* $(\lambda|_{i+1} \in [y_i]$*, for some i, and the target MNR refines the equivalence relation:*

$$\equiv_i = \{[y_1], \dots, [(\lambda|_i], [y_i] \bigcap \overline{[(\lambda|_i]}, \dots, [y_n]\}$$

Proof. The counter-example we are given is a counter-example to the correctness of g. Consider for all i the best-fit equivalence class of $(\lambda|_i$. Since the target MNR refines $\equiv_i$ the canonical elements y_j of $\equiv_i$ describe an equivalence class of the MNR; however, the best-fit equivalence classes of these y_j would contain strings not in their associated equivalence classes of the MNR. We know one of the prefixes of λ is one of these misclassified elements. By using condition (i) of Definition 2.1.1 we know of one way elements misclassified by the g-function manifest themselves; namely, if all the prefixes were being correctly classified, then for all i $[\![(\lambda|_i]\!] = [y_j]$ and $[\![(\lambda|_{i+1}]\!] = [\![y_j a_{i+1}]\!]$. Since $[\![y_j a_{i+1}]\!]$ determines where the state corresponding to $[y_j]$ transitions on a_{i+1}, this further implies that

λ would be accepted by our hypothesis DFA if and only if it were in the language L (recall our current partition refines $\{\{x \in L\}, \{x \notin L\}\}$). Since this would contradict λ being a counter-example, we conclude $\exists i$ such that $[\![(\lambda|_i]\!] = [y_j]$ but $[\![(\lambda|_{i+1}]\!] \neq [\![y_j a_{i+1}]\!]$. Take the first such i. The prefix $(\lambda|_i$ cannot be in the same equivalence class as y_j because of condition (i) of Definition 2.1.1. Thus $(\lambda|_i$ describes a new equivalence class of the target MNR.

We now show that if a block of the target MNR refined an equivalence class of the hypothesis equivalence relation, then this same block refines only one of its two descendant blocks in the next hypothesis. As described before, if δ distinguished $[\![(\lambda|_{i+1}]\!]$ and $[\![y_j a_{i+1}]\!]$ (by assumption) then $a_{i+1}\delta$ distinguishes $[(\lambda|_i]$ and $[y_j]_{new} = [y_j]_{old} \bigcap \overline{[(\lambda|_i]}$. If δ' distinguish the old class $[y_j]$ from all other classes, it continues to distinguish these two new equivalence classes from all others. We need now only show that if B is a block of the MNR, and $B \subset [y_j]_{old}$ then either $B \bigcap [(\lambda|_i] = \emptyset$ or $B \bigcap [y_j] = \emptyset$. This is true because we know all elements of B still produce the same answers from the membership queries for all old distinguishing formula δ. Moreover, because they are all in the same block of the MNR, they are all sent to the same state in the target DFA. Because this DFA is deterministic for any $x \in B$, and for any $z \in \Sigma^*$ including $z = a_{i+1}\delta$, we get the same answer from membership(xz). This means all $x \in B$ end up in the same best-fit equivalence class, which is either $[(\lambda|_i]$ or $[y_j]_{new}$.

□

2.6 An Example

In this section we present a worked example of walking through the logic of Angluin's algorithm on a small target DFA. Hopefully the intuition from the previous sections will be made clear. The target we are learning is shown in Figure 2.2.

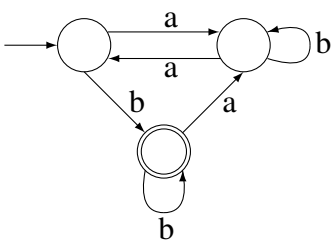


Figure 2.2: Target DFA

Creating the Initial Partition:

Steps:

1) Membership(ϵ) = No

2) Therefore $\epsilon \in \{x \notin L\}$

3) Let initial hypothesis h_1 be the DFA of Figure 2.3

4) Equivalent(h_1) = NO, A counter-example is (bbb,YES)

5) We conclude $bbb \in \{x \in L\}$

6) Let the initial hypothesis relation, $\equiv_1$ be that shown in Figure 2.4

Computing the Next Hypothesis:

Steps:

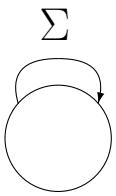


Figure 2.3: Example's Initial Hypothesis

Initial Partition: $\{[\epsilon], [bbb]\}$

Distinguished by: ϵ

Figure 2.4: Example's Initial Partition.

1) Assume $\equiv_1$ is a MNR for L

2) Use the g-function to create a hypothesis DFA h_2 by sampling the canonical elements of $\equiv_1$

   2a) For each y such that $[y] \in \equiv_1$ we create a state in h_2

   2b) For each $a \in \Sigma$, and for each canonical y, we compute $[\![ya]\!] = [y'] \in \equiv_1$

   2c) For the states in h_2 corresponding to y and y' in the previous step, we add an a-transition from the state corresponding to y to the state corresponding to y'

3) The above computations are summarized in Table 2.1

4) The resulting DFA, h_2, is shown in Figure 2.5

Verifying and Modifying the Hypothesis:

Steps:

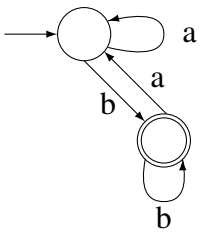


Figure 2.5: The Second Hypothesis DFA

1) Test Equivalent(h_2) = NO. A Counter-example is (aaab, NO)

2) The only condition of Definition 2.1.1 that $\equiv_1$ breaks is condition (i)

3) Thus we can break $aaab$ into three strings p_i, α, s_i such that $aaab = p_i \alpha s_i$. Here 'p_i' is a prefix, 'α' a character, and 's_i' a suffix of $aaab$

4) Furthermore the above string $p_i \alpha s_i$ satisfies $[\![p_i]\!] = [y] \in \equiv_1$ but $[\![p_i \alpha]\!] \neq [\![y\alpha]\!]$

5) We now show the calculations to find p_i,α, and s_i. (skipping $p_0 = \epsilon$)

6) For $p_1 = a$, $\alpha = a$, $s_1 = ab$:

   6a) Compute $[\![a]\!]$:

   6b) Membership($a\epsilon$) = NO

   6c) Therefore $[\![a]\!] = [\epsilon]$

Table 2.1: Computing the Second Hypothesis

membership($\epsilon\, a$)	=	NO	$\rightarrow$	$\epsilon\, a \in [\epsilon]$
membership($\epsilon\, b$)	=	YES	$\rightarrow$	$\epsilon\, b \in [bbb]$
membership($bbb\, b$)	=	YES	$\rightarrow$	$bbb\, b \in [bbb]$
membership($bbb\, a$)	=	NO	$\rightarrow$	$bbb\, a \in [\epsilon]$

6d) Does $[\![a\,a]\!] = [\![\epsilon\,a]\!]$?

6e) From h_2, $[\![\epsilon\,a]\!] = [\epsilon]$

6f) Compute $[\![a\,a]\!]$:

6g) Membership($aa\,\epsilon$) = NO

6h) Therefore $[\![aa]\!] = [\epsilon]$

6i) Does $[\![a\,a]\!] = [\![\epsilon\,a]\!]$? YES

7) For $p_2 = aa$, $\alpha = a$, $s_2 = b$:

7a) We know $[\![aa]\!] = [\epsilon]$

7b) Does $[\![aa\,a]\!] = [\![\epsilon\,a]\!]$?

7c) From h_2, $[\![\epsilon\,a]\!] = [\epsilon]$

7d) Compute $[\![aa\,a]\!]$:

7e) Membership($aaa\,\epsilon$) = NO

7f) Therefore $[\![aaa]\!] = [\epsilon]$

7g) Does $[\![aa\,a]\!] = [\![\epsilon\,a]\!]$? YES

8) For $p_3 = aaa$, $\alpha = b$, $s_3 = \epsilon$:

8a) We know $[\![aaa]\!] = [\epsilon]$

8b) Does $[\![aaa\,b]\!] = [\![\epsilon\,b]\!]$?

8c) From h_2, $[\![\epsilon\,b]\!] = [bbb]$

8d) Compute $[\![aaa\,b]\!]$:

8e) Membership($aaab\,\epsilon$) = NO

8f) Therefore $[\![aaab]\!] = [\epsilon]$

8g) Does $[\![aaa\,b]\!] = [\![\epsilon\,b]\!]$? NO

9) Therefore $[\epsilon]$ can be split into $[\epsilon]$ and $[aaa]$

10) Furthermore $[\epsilon]$ and $[aaa]$ are distinguished by $b\delta$ where δ distinguished $[\![aaa\,b]\!]$ from $[\![\epsilon\,b]\!]$. Thus $\delta = \epsilon$

11) Moreover $[\epsilon]$ and $[aaa]$ are still distinguished from $[bbb]$ by ϵ

12) The new partition $\equiv_2$ is shown in Figure 2.6

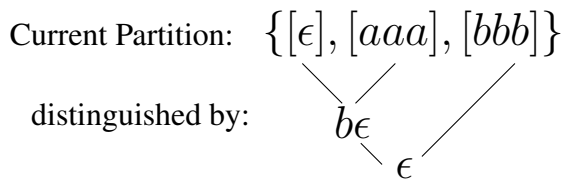


Figure 2.6: Example's Second Partition.

As a final note, we can clearly see from Figure 2.6 that the number of distinguishing formula needed to distinguish all pairs of known equivalence classes is not $\frac{n^2}{2}$ as one might expect. When we split an equivalence class in two, those two subclasses are still distinguished by the string that distinguished their parent. In fact any further subclasses will be distinguished by that same string. Thus we can use the known distinguishing strings to do a binary search to find the best-fit class. Thus computing the best-fit equivalence class takes time $O(\log \frac{n^2}{2}) = O(\log n)$.

2.7 Angluin's Algorithm

These are the important steps of the algorithm, for the assisted learning of target DFA A. At step j of the algorithm we call the equivalence oracle on $\hat{A} = g(\equiv_j)$. If $\hat{A} \neq A$, clearly $\equiv_i$ is not a Myhill-Nerode relation, and the only condition of definition 2.1.1 that it could fail is (i).

Thus we can break the counter-example λ into three parts:

$$\lambda = (\lambda|_i \; a \; |\lambda)_{i+1} \tag{2.4}$$

for $a \in \Sigma$ and $(\lambda|_i, |\lambda)_{i+1} \in \Sigma^*$. Furthermore, it must be the case that $\exists y$ (y is a canonical string) such that for the best-fit equivalence classes we find:

$$[\![(\lambda|_i]\!] = [y] \text{ but } [\![(\lambda|_i a]\!] = [\![(\lambda|_{i+1}]\!] \neq [\![[ya]]\!] \tag{2.5}$$

This is true by condition (i) of definition 2.1.1. The correct conclusion to draw from this is that $(\lambda|_i$ is describing a new equivalence class (that is, distinguished from y by its behaviour on input a. It is demonstrating a point where the g-function erred by just sampling the canonical element of $[y] = [\![(\lambda|_i]\!]$. Having discovered the new equivalence class we can update the partition, by removing the equivalence class $[y]$ and replacing it with two new equivalence classes $[y]_{new}$ and $[(\lambda|_i]$ who are distinguished by string $a\delta$ (where δ was the string that distinguished $[\![\lambda_1 a]\!]$ from $[\![ya]\!]$).

Further note that at each round we only need to compute one best-fit equivalence class. That is first we compute y, such that $[\![(\lambda|_i]\!] = [y]$. Then we compute both $[\![\lambda_i a]\!]$ and $[\![ya]\!]$. However $[\![\lambda_i a]\!] = [\![\lambda_{i+1}]\!]$, so as we increase i we only need to compute one of $[\![\lambda_i]\!]$ and $[\![\lambda_i a]\!]$; in fact we only need to compute the second value, since we can reuse the result for the first value of the next increment of i. Furthermore starting at $i = 0$, we know $[\![\lambda_0]\!] = [\epsilon]$

—and by design $[\epsilon]$ is always a partition of the hypothesis relation. Secondly the value of $[\![ya]\!]$ has also already been computed, when we computed g($\equiv_j$), thus we need only check which state this transition leads to, and pick the corresponding equivalence class of the hypothesis.

The simplified procedure is described in Subroutine 2.7.1 which computes the split of Equation 2.4:

Subroutine 2.7.1. Split:

1 —for each prefix $(\lambda|_i$ of the counter-example λ (where ϵ is best-fit equivalence class of the 0^{th} prefix)

2 —let y_i = $[\![(\lambda|_i]\!]$ (already computed)

3 —compute $y_{i+1} = [\![(\lambda|_{i+1}]\!]$

4 —Find the state, st, of g($\equiv_i$) corresponding to y_i.

5 —compute $\widehat{y_{i+1}}$ as the state that st transition to on a, the $(i+1)^{st}$ character of λ (equivalent to $[\![y_i a]\!]$).

6 —repeat until $\widehat{y_{i+1}} \neq y_{i+1}$

7 —let δ be the distinguishing string for $[\![(\lambda|_i a]\!]$ and $[\![y_i a]\!]$

8 —replace $[y]$ with $[y]$ and $[(\lambda|_i a]$ distinguished by $a\delta$.

9 —$[y]$ and $[(\lambda|_i a]$ still distinguished from all other states by string that distinguished the original partition $[y]$.

2.7.1 Algorithm Description

We present a simple outline of the algorithm.

Algorithm 2.7.2. Angluin's Algorithm

—Initialization
    —membership(ϵ)
        —if yes construct hypothesis DFA that accepts all strings
        —else construct hypothesis DFA that accepts no strings
    —equivalent(hypothesis)
        —if yes terminate
        —else create initial partition as described in figure 2.1
—Repeat following until termination
    —create a hypothesis using g function from equivalence relation
     computed by best-fitting any string into known equivalence classes.
    —equivalence(hypothesis)
        —if yes terminate
        —Else, Run the Split subroutine on counter-example.

Chapter 3

Introduction to Bisimulation

In a typical Computer Science curriculum MNRs are introduced to show how to minimize DFA. We used them to motivate an algorithm that *learned* minimal DFA. In a typical Computer Science curriculum we are taught that there is no unique minimal NFA; however, with the notion of bisimulation instead of MNRs, we can define a unique minimal NFA [13]. When reading this chapter, consider how bisimulation relations a stronger notion than MNRs by allowing us to distinguish non-determinism, but also a weaker notion than MNRs in the sense that bisimulation relations are too exclusive. These observations will have implications when attempting to build a learning algorithm with bisimulation as a foundation.

3.1 Bisimulation

In this section we define Labeled Transition Systems (LTSs) and bisimulation equivalence: the notion of equivalence we intend to use to distinguish LTSs. In the next chapter we introduce Hennesy-Milner logic (HML), which we will use to model LTSs. HML formulæ are more descriptive then strings. Our learning algorithm will interact with the oracles using HML formulæ, thus allowing HML membership queries to strengthen our algorithm. All these concepts are originally due to Milner [14], whose notations and proofs we paraphrase here.

3.1.1 Labeled Transition Systems

Definition 3.1.1 (LTS). *A LTS is a 4-tuple*

$$(S, S_0, \Sigma, \xrightarrow{t}) \tag{3.1}$$

where ...

- S *is a non-empty set of states of the LTS*
- S_0 *is an element of S, called the initial state*
- Σ *is a set of actions the LTS can perform*
- $\xrightarrow{t}$ *is a set of transitions of the LTS,* $\xrightarrow{t} \subset S \times S$. *Note: 't' is a placeholder for the label (*$\in \Sigma$*).*

Similarly, we can define a LTS as a 3-tuple with no initial state (see [14]) or as a 5-tuple with a set of accepting states.

For clarity when we refer to $\xrightarrow{t}$, we refer to the entire transition function, that is

$$\xrightarrow{t} = \bigcup_{\alpha \in \Sigma} \xrightarrow{\alpha} \tag{3.2}$$

We use $\xrightarrow{\alpha}$ to refer to just transitions on a single action, called an α-transition.

For a LTS L = $(S, S_0, \Sigma, \xrightarrow{t})$ with $st, pt \in S$, we write: $st \xrightarrow{\alpha} pt$, to mean: state st transitions to state pt on action α. Where $\alpha^* \in \Sigma^*$ we write: $st \xrightarrow{\alpha^*} pt$ to mean: state st transitions to state pt after the series of actions in the string α^*; that is $\exists\ p_1 \dots p_n \in S$, such that $st \xrightarrow{a_0} p_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} p_n \xrightarrow{a_n} pt$, with $\alpha^* = a_0 a_1 \dots a_n$. In such a case we call pt an α^*-derivative of st.

The original intent of LTSs, as defined here, was to provide meaning to Milner's concurrency language CCS. The states of a LTS modeled states of some process and the

labeled transition exiting a state represented actions available to a process in said state. For our concerns we need only note that LTSs are a generalization of DFAs in that when we ignore final states, a DFA is a LTS if we equate the finite alphabet of the DFA with the actions of the LTS. Note however that LTSs need not be finite, nor deterministic. Learning algorithms for LTS equate to learning a process.

3.1.2 Bisimulation Equivalence

We now desire a notion of equivalence for LTSs. Several exist including:

(i) Language equivalence

(ii) Trace equivalence

(iii) Failure equivalence

(iv) Bisimulation equivalence

We chose Bisimulation equivalence because it can be characterized by HML formulæ (see Chapter 4). The other notions of equivalence are still worthwhile to explore as they may lead to further research. The intuition behind bisimulation equivalence is that two states st and pt are bisimilar if they can perform the same sequences of actions; rather, any α-derivative of st is bisimilar to an α-derivative of pt. More formally, we first define a bisimulation relation:

Definition 3.1.2. *[Bisimulation Relation] For a LTS* $(S, S_0, \Sigma, \xrightarrow{t})$*, let* $\approx \subset S \times S$*. We call* $\approx$ *a bisimulation (over L) if for any* $st, pt \in S$ *and* $\forall\ \alpha \in \Sigma$ *we get:*

(i) if $\exists st'$ *such that* $st \xrightarrow{\alpha} st'$*, then* $\exists pt'$ *such that* $pt \xrightarrow{\alpha} pt'$ *and* $(st', pt') \in \approx$*.*

(ii) if $\exists pt'$ *such that* $pt \xrightarrow{\alpha} pt'$*, then* $\exists st'$ *such that* $st \xrightarrow{\alpha} st'$ *and* $(st', pt') \in \approx$.

We will also want to define the maximal bisimulation, $\simeq$, since it will turn out to be an equivalence relation:

Definition 3.1.3. *Let L =* $(S, S_0, \Sigma, \xrightarrow{t})$ *be a LTS, define* $\simeq$ *as:*

$$\simeq = \bigcup \{\approx : \approx \text{ is a bisimulation over L}\} \tag{3.3}$$

We say $\simeq$ *is over L.*

We now use $\simeq$ to define equivalence:

Definition 3.1.4 (Bisimulation Equivalence)**.** *We define bisimulation equivalence (bisimilarity) both among states of a LTS and among LTSs themselves:*

- *For a LTS L =* $(S, S_0, \Sigma, \xrightarrow{t})$*, we say two states* $st, pt \in S$ *are bisimilar if* $(st, pt) \in \simeq$ *over L.*
- *We say two LTSs,* $L_1 = (S_1, S_{01}, \Sigma, \xrightarrow{t}_1)$ *and* $L_2 = (S_2, S_{02}, \Sigma, \xrightarrow{t}_2)$ *are bisimilar if the LTS L =* $(S, S_0, \Sigma, \xrightarrow{t})$ *with* $S = S_1 \cup S_2$ *and* $\xrightarrow{t} = \xrightarrow{t}_1 \cup \xrightarrow{t}_2$ *and where the initial state is chosen arbitrarily between* S_{01} *and* S_{02}*, satisfies the property:* $(S_{01}, S_{02}) \in \simeq$*, over L.*

This definition is adapted from [18] (pg260). In the next sub-section we prove the claim that $\simeq$ is an equivalence relation, as well as some other properties of bisimulation.

We can easily update the definition of bisimilarity between two LTSs to include a notion of final states: we only need to add the condition that if two states are bisimilar then either they are both final states, or both non-final states.

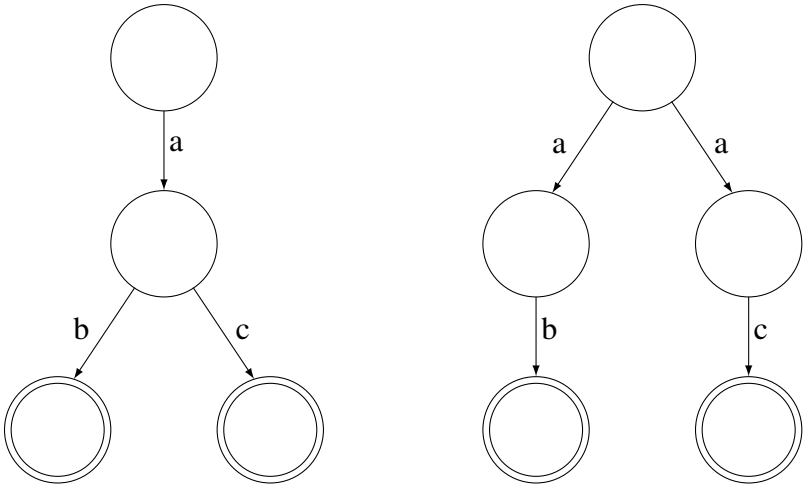


Figure 3.1: Two language equivalent, non-bisimilar LTS

It may not be entirely clear at the moment that bisimulation equivalence is any different then language equivalence. We shall see it is in fact a refinement of language equivalence. Figure 3.1 demonstrates that bisimulation and language equivalence are different.

3.1.3 Maximal Bisimulation

In the following section, and until the end of the chapter, we set up some of the important properties of bisimulation that will allow us to draw analogies between bisimulation theory and MNRs. Not all these parallels will be fruitful, yet they all suggest future work.

Proposition 3.1.5. *For $\simeq$ over a LTS L = $(S, S_0, \Sigma, \xrightarrow{t})$, $\simeq$ is an equivalence relation.*

Proof. We need to show it satisfies the following three properties:

Reflexive: Consider the set $I = \{(st, st) : st \in S\}$. For any st, set I trivially satisfies the conditions of 3.1.2. Therefore $st \approx st$, and I is a bisimulation relation; thus by definition $I \subset \simeq$.

Symmetric: By inspection of 3.1.2, we see that if $\exists$ a bisimulation relation $\approx$ over L, and if $st \approx pt$ (for any $st, pt \in S$), then $\exists$ a bisimulation relation $\approx^{-1}$ such that $pt \approx^{-1} st$. Therefore $\approx^{-1} \subset \simeq$.

Transitive: Consider two bisimulations $\approx_1$ and $\approx_2$ over L. For any $st, vt \in S$ consider those such that $st \approx_1 \circ \approx_2 vt$. Then $\exists pt \in S$ such that $st \approx_1 pt$ and $pt \approx_2 vt$. By Definition 3.1.2 if $st \xrightarrow{\alpha} st'$ then $\exists pt'$ such that $pt \xrightarrow{\alpha} pt'$ and $st' \approx_1 pt'$. Likewise we find $\exists vt'$ such that $pt' \approx_2 vt'$. Therefore $st' \approx_1 \circ \approx_2 vt'$ A symmetric argument shows that if $vt \xrightarrow{\alpha} vt'$ we again get $st' \approx_1 \circ \approx_2 vt'$. Hence $\approx_1 \circ \approx_2$ is a bisimulation and thus $\approx_1 \circ \approx_2 \subset \simeq$.

□

We might also hope $\simeq$ is a bisimulation relation.

Proposition 3.1.6. *For $\simeq$ over a LTS L = $(S, S_0, \Sigma, \xrightarrow{t})$, $\simeq$ is a bisimulation relation.*

Proof. We need only prove for a collection $\mathcal{C}$ of bisimulation relations over L, $\bigcup \mathcal{C}$ is a bisimulation relation. Let $\mathcal{R} = \bigcup \mathcal{C}$. For any $st, pt \in S$, if $st \; \mathcal{R} \; pt$ then $\exists \approx_i$ (a bisimulation relation over L) such that $st \approx_i pt$. By Definition 3.1.2, for any $\alpha \in \Sigma$ such that $\exists st'$ such that $st \xrightarrow{\alpha} st'$ then $\exists pt'$ such that $pt \xrightarrow{\alpha} pt'$ and $st' \approx_i pt'$. Therefore for any such α, st' and pt' it is also true that $st' \; \mathcal{R} \; pt'$ by definition. □

3.1.4 Degrees of Bisimilarity

One useful concept is that of degrees of bisimilarity. We will need it for an important proof later. By degrees, we mean to assign a number, k, to quantify exactly how close to being bisimilar two LTSs are, as a sort of *almost*-bisimilarity. To do this we need only make one subtle modification to the definition of bisimilarity.

Definition 3.1.7 (k-bisimilarity)**.** *For two LTSs* $L_1 = (S_1, S_{01}, \Sigma, \xrightarrow{t}_1)$ *and* $L_2 = (S_2, S_{02}, \Sigma, \xrightarrow{t}_2)$*, with start states* S_{01} *and* S_{02}*, we say* L_1 *and* L_2 *are k-bisimilar if the start states are k-bisimilar, written:* $S_{01} \simeq_k S_{02}$

where we define $\simeq_k$ *inductively as follows:*

- $\simeq_0 = (S_1 \cup S_2) \times (S_1 \cup S_2)$
- *For* $st, pt \in (S_1 \cup S_2)$*,* $st \simeq_{k+1} pt$ *if and only if* $\forall \alpha \in \Sigma^*$*:*
  - *(i) if* $\exists st'$ *such that* $st \xrightarrow{\alpha} st'$*, then* $\exists pt'$ *such that* $pt \xrightarrow{\alpha} pt'$ *and* $(st', pt') \in \simeq_k$*.*
  - *(ii) if* $\exists pt'$ *such that* $pt \xrightarrow{\alpha} pt'$*, then* $\exists st'$ *such that* $st \xrightarrow{\alpha} st'$ *and* $(st', pt') \in \simeq_k$*.*

The difference in the definition is that each successive state is only required to be of one degree of bisimilarity less. It is not hard to see that our old definition of bisimilarity is equivalent to $\simeq_\infty$, which leads us to the following corollary:

Corollary 3.1.8.

$$\simeq = \bigcap_k^\infty \simeq_k \tag{3.4}$$

3.2 Minimizing LTSs

Among all LTSs bisimilar to another LTS, some subset of them must represent the smallest such bisimilar LTS. It would be conceptually desirable for the cardinality of this set to be one: allowing us to speak of *the* minimum LTS; as is the case for minimal language equivalent DFAs, we desire uniqueness. In essence we have already shown this, but we want to show it can be done efficiently. Otherwise, how could we hope to learn it?

We achieve our goal of proving this by ultimately demonstrating an algorithm that computes the minimum bisimilar LTS (described in various forms by Kanellakis [10],Tarjan [15],

and Fernandez [7]). To develop the minimization algorithm: first we observe that $\simeq$ is the maximal bisimulation over a LTS; then, we develop a notion of compatibility between partitions and LTSs. This notion will allow us to rephrase the problem in terms of partitions. What does this afford us? We can compute the partition $\simeq$ and turn its equivalence classes into states of a LTS; compatibility will ensure the partition is a bisimulation; and finding the *coarsest* such partition will ensure the associated LTS is minimal. Our focus now is on demonstrating how to compute $\simeq$.

3.2.1 Deciding Bisimilarity of LTSs

We begin with an obvious observation:

Corollary 3.2.1. *For a LTS L, the bisimulation relation $\simeq$ over L is the largest bisimulation relation, ordered by inclusion. Thus, its partitions are coarsest.*

Proof. This follows directly from the definition of $\simeq$ and 3.1.6. □

A high level description of the algorithm is that it builds the states of the minimum LTS out of the equivalence classes of $\simeq$. Since $\simeq$ equates as *many* states (of the original LTS) with comparable behaviors as possible— a new LTS derived this way will have as *few* states as possible, and it will be bisimilar to the old LTS.

To state this more formally we now define compatibility (definition from Fernandez [7]), which momentarily we will prove to be a restatement of bisimilarity in terms of equivalence classes. We use $\xrightarrow{\alpha}{}^{-1}$ to denote the pre-image of the function $\xrightarrow{\alpha}$. Thus $\xrightarrow{\alpha}{}^{-1}(A)$ is the set of states that have α-transitions to states in the set A.

Definition 3.2.2 (Compatibility)**.** *For a LTS L =* $(S, S_0, \Sigma, \xrightarrow{t})$*, and a partition of S,* $\varphi =$

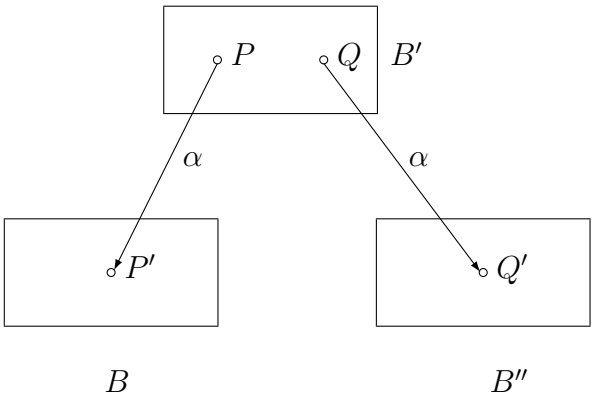


Figure 3.2: The idea behind proof of 3.2.3

$\{B_1 \dots B_n\}$, we say φ is compatible with $\xrightarrow{t}$ if and only if $\forall \alpha \in \Sigma$ and $\forall B, B' \in \varphi$

$$B \cap \xrightarrow{\alpha}^{-1}(B') \neq \emptyset \Longrightarrow B \subset \xrightarrow{\alpha}^{-1}(B') \tag{3.5}$$

Notice the similarity between Equation 3.5 and the conditions of MNRs. One apparent difference is that MNR are over strings which could uniquely identify states of a DFA. The notion of compatibility is over states of a LTS, which we have yet to explain a way to uniquely identify.

Lemma 3.2.3. *For a partition $\varphi = \{B_1 \dots B_n\}$, φ is compatible with $\xrightarrow{t}$ if and only if φ is a bisimulation.*

Proof. First we note that $\xrightarrow{\alpha}^{-1}(B) = \{st : st \xrightarrow{\alpha} pt \wedge pt \in B\}$, the pre-image of the function $\xrightarrow{\alpha}$.

$\Leftarrow$ Since φ is a bisimulation, for any two elements st, pt of any equivalence class B we have $st \; \varphi \; pt$. Thus by Definition3.1.2 we know any α-derivative of st (say st') must

be matched by an α-derivative of pt (say pt'), and $st'\ \varphi\ pt'$. Thus if the pre-image of some equivalence class (say B') under the relation $\xrightarrow{\alpha}$ contains some element st of B , every element of B has an α-derivative in B', giving us Equation 3.5.

$\Rightarrow$ Figure 3.2 shows the contrapositive as $B \bigcap \xrightarrow{\alpha}^{-1} (B') \neq \emptyset$ but $B \not\subset \xrightarrow{\alpha}^{-1} (B')$. If the partition is not a bisimulation, then the implication of Equation 3.5 clearly need not hold, and in fact must not hold for some B, B'.

□

This lemma provides a simple condition for whether a partition is a bisimulation relation or not. This trivially parlays into an algorithm which just checks to see if this condition is met, and if not, rectifies it:

Algorithm 3.2.4. Bisimilarity Checker:

—For a partition $\varphi = \{B_1 \ldots B_n\}$, a transition function $\xrightarrow{t}$ and a LTS, with actions Σ, and states S

—Start with the partition: $\{S\}$

—Find B, B', α that do not satisfy equation 3.5.

—Repeat until no more such B, B', α can be found

—Replace B by the equivalence classes:

- $B \cap \xrightarrow{\alpha}^{-1}(B')$
- $B \setminus \xrightarrow{\alpha}^{-1}(B')$

Certainly this algorithm will compute a bisimulation relation eventually. We want to show that it will terminate in polynomial time computing the coarsest compatible partition. We now prove this claim (adapted from [7]):

Proposition 3.2.5. *Algorithm 3.2.4 computes in polynomial time the coarsest compatible partition*

Proof. Since the number of blocks of φ cannot exceed the cardinality of S, and the algorithm increases the number of blocks each step, the algorithm will eventually terminate. The partition will be compatible when it terminates, as the main loop just splits the problematic equivalence classes; incidentally, the finest possible partition $\{\{st\}|st \in S\}$ is compatible with any $\xrightarrow{\alpha}$. To show that the final partition is the coarsest, we further need to demonstrate that at any step of the algorithm, any stable partition is a refinement of the current computed partition (including obviously the final partition at the last step). By stable, we mean that we cannot find B, B' and α that do not satisfy Equation 3.5. We argue for this statement inductively.

Our inductive hypothesis will be that any stable partition is a refinement of the current partition. This is certainly true of $\{S\}$. Suppose at step k of the algorithm we compute partition ϕ^k, and furthermore that we found B, B', α not satisfying equation 3.5 for ϕ^k, and that our inductive hypothesis holds at this step. Now, by the induction hypothesis, let φ be any stable refinement $\varphi \sqsubset \phi^k$.

We see that φ may not contain the block $B \in \phi^k$ as it is a refinement of ϕ^k; however it must contain some collection $\hat{B} = \{\hat{B}_1 \dots \hat{B}_n\}$ where each $\hat{B}_i \in \varphi$, and the disjoint union is $\bigcup \hat{B} = B$. Likewise B' may or may not be in φ; however some collection of blocks $\check{B}$ has elements $\check{B}_i \in \varphi$ which satisfy $\bigcup \check{B} = B'$. We want to show that if we perform

the replacement step of the algorithm using such B, B', α to obtain ϕ^{k+1}, the inductive hypothesis still holds.

Since the algorithm only changed block $B \in \phi^k$ we need only focus our attention there; the only possible worry is that the blocks of $\hat{B}$ no longer refine the blocks that will replace B. Note that we can express ϕ^{k+1} as:

$$\phi^{k+1} = (\phi^k \setminus \{B\}) \bigcup \{(B \bigcap \xrightarrow{\alpha}{}^{-1}(B'))\} \bigcup \{(B \setminus \xrightarrow{\alpha}{}^{-1}(B'))\} \tag{3.6}$$

It is sufficient to show that each $\hat{B}_i$ in the collection $\hat{B}$ is contained entirely in either of the two new blocks $B \bigcap \xrightarrow{\alpha}{}^{-1}(B')$ and $B \setminus \xrightarrow{\alpha}{}^{-1}(B')$. This is logically equivalent to the statement:

$$\nexists \hat{B}_i \text{ such that } \hat{B}_i \cap \xrightarrow{\alpha}{}^{-1}(B') \neq \emptyset \Longrightarrow \hat{B}_i \subset \xrightarrow{\alpha}{}^{-1}(B') \tag{3.7}$$

Why? Because if $\hat{B}$ shares any elements with $B \bigcap \xrightarrow{\alpha}{}^{-1}(B')$ then it shares elements with $\xrightarrow{\alpha}{}^{-1}(B')$. Equation 3.7 says if that is the case then $\hat{B}_i$ is contained in $\xrightarrow{\alpha}{}^{-1}(B')$, and thus is totally disjoint from $B \setminus \xrightarrow{\alpha}{}^{-1}(B')$.

If however $\hat{B}_i$ does not share elements with $B \bigcap \xrightarrow{\alpha}{}^{-1}(B')$ then since $\hat{B}_i \subset B$, the only conclusion is that $\hat{B}_i$ is disjoint from $\xrightarrow{\alpha}{}^{-1}(B')$. Therefore from the above points we reach the conclusion that $\hat{B}_i \subset B \setminus \xrightarrow{\alpha}{}^{-1}(B')$.

All that remains is to prove Equation 3.7 holds.

We know since φ is compatible with $\xrightarrow{\alpha}$ and the elements of $\check{B}$ are blocks of φ then :

$$\forall \check{B}_j, \hat{B}_i \cap \xrightarrow{\alpha}{}^{-1}(\check{B}_j) \neq \emptyset \Longrightarrow \hat{B}_i \subset \xrightarrow{\alpha}{}^{-1}(\check{B}_j) \tag{3.8}$$

If $\hat{B}_i \cap \xrightarrow{\alpha}{}^{-1}(B') \neq \emptyset$ then $\exists \check{B}_j$ such that $\hat{B}_i \cap \xrightarrow{\alpha}{}^{-1}(\check{B}_j) \neq \emptyset$ which implies $\hat{B}_i \subset \xrightarrow{\alpha}{}^{-1}(\check{B}_j) \subset \xrightarrow{\alpha}{}^{-1}(\check{B}')$.

Since any stable partition is a refinement of the one we compute, our refinement is the unique coarsest refinement. The running time of this algorithm is clearly dependent on

the time it takes to find B, B', α satisfying Equation 3.5. This would take no longer then checking every state and transition of the LTS. That constitutes one round. The number of rounds is bounded by the number of states, since no partition is finer then putting a single state in each equivalence class. □

From this partition we construct the new LTS by creating a state for each equivalence class. Conceptually we can arbitrarily select any process name from the equivalence class to give the state. By the compatibility with $\xrightarrow{t}$ adding the transitions to the LTS is also a trivial task. And as the above proof demonstrates, since the partition was the coarsest, the LTS will have the fewest states of any LTS constructed in this manner. Likewise any smaller LTS would have an underlying partition that would be smaller, contradicting our previous claims.

Definition 3.2.6 (Minimal LTS)**.** *A minimal LTS is one where any bisimilar LTS has no fewer states.*

We end with two very simple corollaries.

Definition 3.2.7. *A* dead state *is a state with no transitions.*

Corollary 3.2.8. *A minimal LTS has at most a single dead state.*

Corollary 3.2.9. *An acyclic LTS has a single dead state, since finite DAGs must have a source and a sink.*

Corollary 3.2.10. *Any LTS that can do every action indefinitely from every state can be minimized to a single LTS that has a self-loop on every action in* Σ

Proof. A state with a single self loop can also do every action indefinitely. □

3.2.2 Minimization of NFA's

Since Non-deterministic Finite Automata (NFAs) differ from LTS only in that they include a concept of a final or accepting state, the question of minimizing NFAs becomes an interesting case study in comparing and contrasting the expressive power of language equivalence versus bisimulation equivalence. While it has been shown that finding a minimal NFA in polynomial time is hard [10], the above algorithm can be trivially adapted to compute minimal Bisimulation Equivalent NFA. We start by computing the maximal bisimulation over the NFA N. We need only ensure that for any partition, the member states are either all accepting states, or all non-accepting states. Then the partitions of states of N become states themselves for the minimal NFA. This algorithm is easier then LTS minimization because of the existence of final states.

Very briefly, the above property suggests a simple table filling algorithm where we identify states which are not bisimilar. We can start by marking all accepting states as being non-bisimilar to non-accepting states. From this we work backwards, if two states on the same labeled-transition enter two states already marked as being non-bisimilar, we know the original states must not be bisimilar and can mark them as such. This process continues until no further changes can be made (See Kozen [13]):

- Create a half table of all pairs of states (s,p)
- mark the table entry (s,p) if only one of the states is an accepting state
- until no changes occur mark a cell (s,p)
    - if $\exists p'$ such that $p \xrightarrow{a} p'$ and $\forall s'$ such that $s \xrightarrow{a} s'$, (p', s') is marked.
    - if $\exists s'$ such that $s \xrightarrow{a} s'$ and $\forall p'$ such that $p \xrightarrow{a} p'$, (p', s') is marked.

We will not prove that this procedure works, other then to comment that the existence of final states simplifies the computation of a bisimulation relation, as we can immediately identify non-bisimilar states.

3.3 Conclusion

We have seen how to use bisimulation to minimize LTSs. Thus we can identify redundant states. This seems to suggests the plausibility of assisted learning LTSs using bisimulation somehow, as efficient identification of redundancies would seem to be a necessity. In the next chapter we begin to consider what to use as membership queries for LTSs.

Chapter 4

Hennesy-Milner Logic

In this chapter we will introduce Hennesy-Milner Logic (HML) which will serve as the format for membership queries. HML is a modal logic used primarily for model checking. By modal we mean the logic deals with different *modes* of truth beyond truth and falsity, namely possibility and necessity. Since it is used for model checking, unlike other logical systems, HML is devoid of rules of inference; though Stirling [19] expands upon HML and defines a complete tableau proof system for the resulting temporal logic. Again Milner [14] is the best source for an introduction to the topic, followed by [19]. We borrow much of our presentation from these two sources.

4.1 Hennesy-Milner Logic

Since the original formulation of HML, several improvements and additions designed to simplify the syntax of the formulæhave been adopted. We first introduce the most standard presentation of HML, discussing later those syntactic improvements pertinent to our problem —including μ-calculus. Eventually we will adopt a convention where counter-example HML formulæ will be presented in a consistent manner but the HML queries we ask are allowed to be in any valid form. We now introduce HML and its formulæ :

4.1.1 Valid Formulæ

All HML formulæ are built from *the two* atomic formulæ : (1) the constant true and (2) the constant false, using the following rule:

$$\Phi ::= \text{ true} \mid \text{false} \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid [\alpha]\Phi \mid \langle\alpha\rangle\Phi \tag{4.1}$$

where α is drawn from some alphabet Σ. Ultimately, Σ will be the alphabet, or action set, of some associated LTS. Then, the intent of the formulæ is to express what is and is not possible of the LTS. The $\wedge$ and $\vee$ expressions are the 'and' and 'or' connectives of propositional logic. $[\alpha]$ and $\langle\alpha\rangle$ are parameterized versions of the traditional modal operators $\Box$ and $\Diamond$, representing necessity and possibility, respectively. The intended interpretation of these symbols when relating them to LTSs is that $[\alpha]$ means for all α-derrivatives, and $\langle\alpha\rangle$ means there exists some α-derivative. These meanings are made clear by the rules for satisfaction of a formula, given next. As a final note: if in the course of computing an algorithm we construct a formula $\Phi \wedge \Psi$ where $\Psi = \emptyset$ (if, for instance, Φ had yet to be defined) we consider $\Phi \wedge \emptyset$ logically equivalent to Φ. Likewise we will consider $\Phi \vee \emptyset$ logically equivalent to Φ. Thus undefined variables will not affect the rules of satisfaction we define in the next section.

4.1.2 Satisfaction

As stated, the HML formulæ are used only to express statements about LTSs. Thus we would like to define a notion of logical satisfaction which coincides with this intent. To the point, we would like an interpretation of the formulæ where a state of an LTS satisfies a formula, if the formula expresses some possible ability of that state. We note now that we will refer both to states of an LTS satisfying a HML formula and to a LTS itself satisfying

a HML formula —which should be interpreted as the starting state of the LTS satisfying the formula.

Given a state st in some LTS L=$(S, S_0, \Sigma, \xrightarrow{t})$, the rules for satisfaction are:

$$st \models true \tag{4.2a}$$

$$st \not\models false \tag{4.2b}$$

$$st \models \Phi \wedge \Psi \Leftrightarrow st \models \phi \text{ and } st \models \psi \tag{4.2c}$$

$$st \models \Phi \vee \Psi \Leftrightarrow st \models \phi \text{ or } st \models \psi \tag{4.2d}$$

$$st \models [\alpha]\Phi \Leftrightarrow \forall pt \text{ such that } st \xrightarrow{\alpha} pt, pt \models \Phi \tag{4.2e}$$

$$st \models \langle\alpha\rangle\Phi \Leftrightarrow \exists pt \text{ such that } st \xrightarrow{\alpha} pt, pt \models \Phi \tag{4.2f}$$

where Φ and Ψ are themselves HML formulæ and $\alpha \in \Sigma$. Note that the modal operators encode the transitions of the system. Since the modal operators are often thought of as actions or behaviours of some systems, when a state satisfies some formula Φ we say the state *admits* the actions/behaviours encoded in the modal operators of Φ.

For either of the modal operators it is useful shorthand to express a sequence such as $[a_1][a_2]\ldots[a_n]$ more compactly as $[a_1 a_2 \ldots a_n]$; that is, we allow $\alpha \in \Sigma^*$ in the above definition of satisfaction. We also shorten *true* to *tt* and *false* to *ff*. It is also not hard to see that the expression $\langle\alpha\rangle tt$ expresses the possibility a state can do an action α. Whereas it should be made clear that $[\alpha]ff$ expresses the fact that a state cannot do the action α, as opposed to $\langle\alpha\rangle ff$. Certainly the only way a state could satisfy $[\alpha]ff$ is if it had no α actions, because *no* state satisfies *ff*. Any α action would inevitably lead to a state that did not satisfy *ff*, thus by 4.2e $[\alpha]ff$ could not be satisfied by any state with an α action. We summarize these facts in Proposition 4.1.2 following some introductory examples.

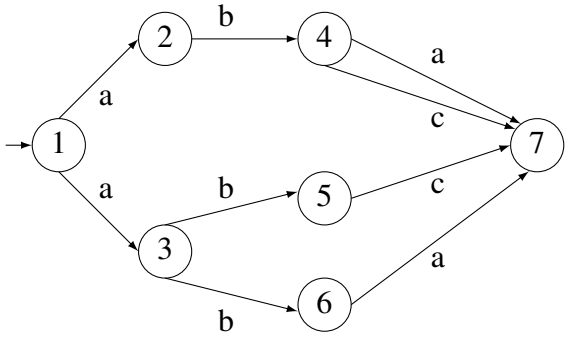


Figure 4.1: An example of a more complex LTS

4.1.3 An Example

Figure 4.1 shows a more involved LTS. As it turns out we can find, for each state, a HML formula which is satisfied uniquely by that state.

$$\langle aba \rangle tt \tag{4.3a}$$

$$\langle b \rangle (\langle a \rangle tt \ \wedge \ \langle c \rangle tt) \tag{4.3b}$$

$$\langle b \rangle ([a] ff \ \vee \ [c] ff) \tag{4.3c}$$

$$\langle a \rangle tt \ \vee \ \langle c \rangle tt \tag{4.3d}$$

$$\langle c \rangle tt \ \wedge \ [a] ff \tag{4.3e}$$

$$[c] ff \ \wedge \ \langle a \rangle tt \tag{4.3f}$$

$$[a] ff \ \wedge \ [b] ff \wedge \ [c] ff \tag{4.3g}$$

Equation 4.3 lists formulæ satisfied by a unique state in the order the states are numbered in Figure 4.1. For instance, state③ satisfies formula 4.3c, because there exist states ⑤ and⑥, such that③ $\xrightarrow{b}$⑤ and⑤ satisfies $[a]ff$, since it cannot do any a actions; likewise ③ $\xrightarrow{b}$ ⑥ and⑥ satisfies $[c]ff$. Compare that with state② and formula 4.3b, which can admit a b action followed by either an a *or* c action. From this observation, in some sense it would be desirable if $(\langle a\rangle tt \ \wedge \ \langle c\rangle tt)$ and $([a]ff \ \vee \ [c]ff)$ were the negations of each other —explaining the differences in states② and③ of Figure 4.1. We define negation as such, in light of this observations in Section 4.1.5.

4.1.4 Minimal LTS and HML

For the example LTS in Figure 4.1 we can deduce from the fact that each state has a HML formula that distinguishes it from every other state, in conjunction with the Hennessy-Milner Theorem 4.4.2, that this LTS is minimal. Until we prove the Hennessy-Milner theorem, we will assume the following proposition to be true.

Proposition 4.1.1. *For a minimal LTS, for any pair of states pt and st, $\exists\Phi$ such that $st \models \Phi$, but $pt \not\models \Phi$.*

4.1.5 Properties of Negation in HML

Consider introducing a logical not operator, $\neg$. We would expect classic theorems such as:

$$\neg(\langle a\rangle tt \ \wedge \ \langle c\rangle tt) \equiv \neg(\langle a\rangle tt) \ \vee \ \neg(\langle c\rangle tt) \tag{4.4}$$

to hold, as a consequence of DeMorgan's Laws. Thus we might expect

$$\neg(\langle a\rangle tt) \equiv [a]ff \tag{4.5}$$

to be a valid theorem –and from the definition of satisfaction this appears to be true. This is a result of the same duality for existential quantifiers:

$$\neg\forall x\ \Phi(x)\ \equiv\ \exists x\ \neg\Phi(x) \tag{4.6}$$

We summarize these properties in the following proposition.

Proposition 4.1.2.

(i) $\langle a\rangle tt$ *expresses that a state can do action* α

(ii) $[a]ff$ *expresses that a state cannot do action* α

(iii) $\langle a\rangle ff\ \equiv ff$

(iv) $[a]tt\ \equiv\ tt$

(v) $\neg(\langle a\rangle tt) \equiv [a]ff$

Proof. These do not need much explaining: (i) is obvious since any state satisfies true, a state need only have some α action to satisfy this formula. We already explained (ii) in Section 4.1.2. For (iii) no state satisfies *ff*, so by 4.2f a state satisfies this formula exactly when it satisfies *ff*; therefore they are logically equivalent. For (iv), if the state has no α transitions, then 4.2e is vacuously true; if it does have an α transition, the formula is equally as vacuous as any state satisfies *tt*. Again (v) is explained by 4.6. □

Introducing the negation operator actually allows us to reduce the variety of HML formulæ. This next section is about trying to define a minimal set of HML formulæ sufficient for describing distinguishing features of LTS.

4.2 Reducing Hennesy-Milner Logic

Later when we design a Learning Algorithm we will need to interpret HML counter-examples. We can make this analysis easier if we let the learner and the teacher agree on a minimalist variant of HML for conducting the equivalent-queries. Explicitly, our algorithm will be allowed to make membership queries using any formula that could be considered a HML formula (those of equation 4.1 and subsection 4.2.1); however, the equivalence queries will only return formulæ of the form described in subsection 4.2.6.

4.2.1 A reduced HML

Having formally introduced the negation operator we can use duality laws to reduce the variety of HML formulæ. This reduced version of HML (*reduced HML*) will only have three connectives, and the atomic formula *tt*. They are:

- $\Phi \wedge \Psi$ logical-and
- $\neg\Phi$ logical-not
- $\langle\alpha\rangle\Phi$ a possible action
- tt the atomic value

where Φ, Ψ are valid HML formulæ. Clearly we have not lost any expressive power. We can define *ff* as $\neg tt$, $[\alpha]\Phi$ as $\neg\langle\alpha\rangle\neg\Phi$, and $\Phi \vee \Psi$ as $\neg(\neg\Phi \ \wedge \ \neg\Psi)$. We can generalize $\Phi \wedge \Psi$ to $\bigwedge_i \Phi_i$. For simplicity we consider $\bigwedge_0^0 \Phi_i = tt$.

4.2.2 Incorrect Actions

Let us first make an obvious observation about formula that contain the constant *ff*. The main concern with such formulæ is understanding their meaning intuitively. If we look at the formula $[\alpha][\beta][\gamma]ff$ what should we infer? Probably that the LTS cannot perform a γ action after it has done an α and a β action. The main point is that the formula $[\alpha][\beta][\gamma]ff$ says nothing of whether there are any α or β actions in sequence, but only that *if* their were, they would not be followed by a γ action.

Going back to the definition of when a state satisfies a HML formula (Equations 4.2) we see a state satisfies $[\alpha][\beta][\gamma]ff$ if all α transitions lead to a state which satisfies $[\beta][\gamma]ff$; likewise such a state could only satisfy that formula if all β transitions lead to a state which satisfies $[\gamma]ff$, which is any state which does not have a γ transaction. Compare that with the formulæ $\langle\alpha\rangle\langle\beta\rangle\langle\gamma\rangle ff$ and $\langle\alpha\rangle\langle\beta\rangle[\gamma]ff$. The first is equivalent to *ff*, whereas the second means that there exists some α and β transitions where we end up in a state that cannot do a γ transition, as opposed to the statement that after *all* series of $\alpha\beta$ moves we end up in a state which cannot do a γ transition.

The semantics of *ff* are fortunate for us in that the only difference between a formula ending in tt and one ending in *ff* is the meaning of the modal operator which precedes the *ff*, which we know must be [ ] given Proposition 4.1.2 (iii).

4.2.3 Generalized Actions

We also want to generalize the modal operator $\langle\rangle$ to accept as a parameter a set of actions. In particular $\langle\Sigma\rangle tt$, where Σ is the alphabet of actions, says *something* is possible. Likewise $[\Sigma]ff$ expresses that nothing is possible. This second generalization is quite important as it can be used to identify *dead states*, ones which can do no actions. We also denote

$[\Sigma]ff$ as $[-]ff$. Dead states uniquely satisfy $[-]ff$.

4.2.4 Eliminating Negation

We do not actually care about reduced HML except for the negation operator. But even then we only want to use negation when it is the outermost connective. We want to be able to convert such HML formula into normal HML formula, defined by equation 4.1. We can use an obvious recursive procedure to achieve this:

1) $\neg(\phi \wedge \psi) \Rightarrow (\neg\phi) \vee (\neg\psi)$

2) $\neg(\phi \vee \psi) \Rightarrow (\neg\phi) \wedge (\neg\psi)$

3) $\neg(\langle\alpha\rangle\phi) \Rightarrow [\alpha](\neg\phi)$

4) $\neg([\alpha]\phi) \Rightarrow \langle\alpha\rangle(\neg\phi)$

5) $\neg ff \Rightarrow tt$

6) $\neg tt \Rightarrow ff$

Since all the formulæ move the "¬" symbol into a deeper nesting, and since any HML formula has either a *tt* or a *ff* at their deepest level of nesting, this procedure absorbs any "¬" symbol.

4.2.5 Eliminating 'NO'-instances

When we start designing our algorithm in Chapter 5, by virtue of using equivalence queries to get counterexamples, we expect to get two types of replies:

1) (YES,δ): Our LTS does not satisfy δ, but should

2) (NO, δ): Our LTS satisfies δ, but should not

We can use negation elimination to limit the oracle to type 1) replies. It is conceptually easier to consider only this one possibility. Why exactly this is the case will be made clear momentarily; essentially, we need to reconcile the notion of negation used in HML with our natural language use of negation.

Certainly we cannot eliminate the NO-instances if it fundamentally changes our problem. So first consider Table 4.1 of possible counter-example types:

Table 4.1: Possible counter-examples

Instance type:	*tt*	*ff*
YES	($\langle\alpha\rangle tt$, YES)	($[\alpha]ff$, YES)
NO	($\langle\alpha\rangle tt$, NO)	($[\alpha]ff$, NO)

Their intended semantics are:

(YES,*tt*) –The LTS should do an 'a' action

(YES,*ff*) –The LTS does not do an 'a' action but should

(NO, *tt*) –The LTS does an 'a' action but should not

(NO, *ff*) –The LTS should not *not* do an 'a' action

Clearly (YES,*tt*) and (NO, *ff*) have the same semantics, as do (YES,*ff*) and (NO, *tt*). This is foreshadowed by the identity 4.1.2 $\neg(\langle a\rangle tt) \equiv [a]ff$. Thus, we expect we can change (δ, NO) instances into ($\neg\delta$, YES) instances.

The earlier concern —that the notion of negation "$\neg$" in HML differs in some fundamental but non-intuitive way from our *natural language* notion of negation present in (δ, NO) instances— can easily be laid to rest. We only need recall the definition of satis-

faction. If we altered the LTS so that:

$$\text{LTS} \models \neg\delta$$

then certainly,

$$\text{LTS} \not\models \delta$$

rendering our initial concern unfounded. To express the counter-example in the syntax of equation 4.2.6 we use the procedure of section 4.2.4.

As an example, assume we have the LTS of Figure 4.2:

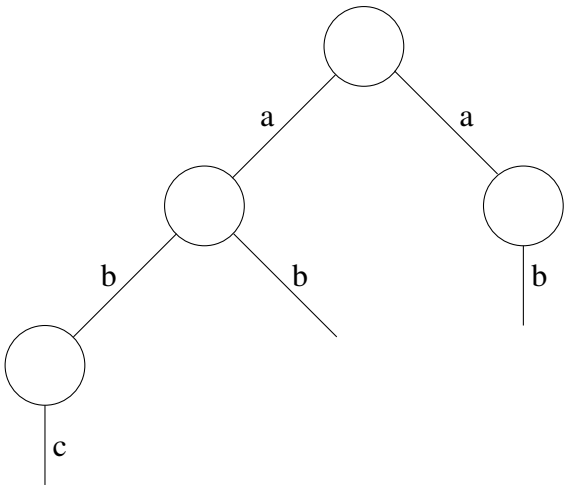


Figure 4.2: Interpreting counter-examples

Further assume that we do an equivalence query and receive as a counter-example the following equation:

$$(\langle a\rangle\langle b\rangle[c]\mathit{ff}, \text{NO})$$

which would convert to:

$$\neg(\langle a\rangle\langle b\rangle[c]\mathit{ff}) \Rightarrow [a][b]\langle c\rangle tt$$

Which would be interpreted as: whatever states we can reach from the start after doing the actions ‘a’ and ‘b’ in order, we must be able to do a ‘c’ action. Such a modification would give us the LTS of figure 4.3

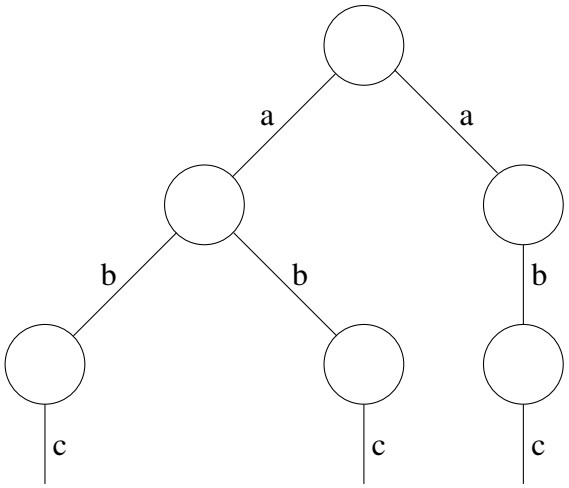


Figure 4.3: Interpreting Counter-Examples results

which no longer satisfies $\langle a \rangle \langle b \rangle [c] \mathit{ff}$, as desired. Hopefully this example makes apparent the advantage of (YES, –) instances: comparing the LTS to the formula is straightforward.

4.2.6 Distinguishing Formula

Our goal is to design a grammar to describe the simplest set of HML formulæ that can elucidate disagreements. The requirement is this: any HML formula Ψ of section 4.2.1 that distinguishes two LTSs can be related to a formula ψ in our new grammar. By combining the two previous subsections (4.2.1 and 4.2.4) we derive this most simplistic version of HML which we shall call *the distinguishing formula*. To derive it we need only make two simple observations:

i) For two non-bisimilar LTSs, L_1 and L_2, let Ψ be a *reduced* HML formula. If $\Psi = \bigwedge \Phi_i$ then $\exists \Phi_k$ such that if $L_1 \models \Psi$ but, $L_2 \not\models \Psi$ then $L_1 \models \Phi_k$ and, $L_2 \not\models \Phi_k$ by equation 4.2c.

ii) Many of the formulæ that can be made using the grammar of subsection 4.2.1 can be trivially simplified. For example: $\forall \Phi$ and $\forall \alpha \in \Sigma$ note that: $\Phi \wedge \mathit{ff} = \mathit{ff}$, $\Phi \vee tt = tt$, $\langle \alpha \rangle \mathit{ff} = \mathit{ff}$, and $[\alpha] = tt$.

Given any normal HML formula and using the second observation recursively we can remove any instance of *tt* or *ff* appearing as their atomic form in equations of the forms $\bigvee_i \Phi_i$ or $\bigwedge_i \Phi_i$ or any instances appearing along side redundant modal operators, by replacing them as show in table 4.2:

Table 4.2: Evaluating Subformulæ

Formula:	$\Phi_k =$*tt*	$\Phi_k =$*ff*
$\bigwedge_i \Phi_i$	$\bigwedge_{i \neq k} \Phi_i$	*ff*
$\bigvee_i \Phi_i$	*tt*	$\bigvee_{i \neq k} \Phi_i$
$\langle \alpha \rangle \Phi_k$	$\langle \alpha \rangle tt$	*ff*
$[\alpha] \Phi_k$	*tt*	$[\alpha]$*ff*

By repeating these replacements, any HML formula will converge to one where all occurrences of the atomic symbols *tt* and *ff* are paired with the modal operators $\langle\rangle$ and [], respectively. Thus we can take our atomic primitives to be $\langle \alpha \rangle tt$ and $[\alpha] \mathit{ff}$, for any $\alpha \in \Sigma$.

By repeatedly using the first observation we can eliminate any distinguishing formula whose primary connective is $\wedge$. Thus by combining these two observations with the grammar of 4.2.1 we get the following grammar:

- $\neg \Phi$
- $\langle \alpha \rangle (\bigwedge_i \Phi_i)$

- $\langle\alpha\rangle tt$

where the Φ's are valid formulæ of the above form.

Finally using the negation elimination of section 4.2.4 repeatedly we are left with a grammar of the form:

- $\langle\alpha\rangle(\bigwedge_i \Phi_i)$
- $[\alpha](\bigvee_i \Phi_i)$
- $\langle\alpha\rangle tt$
- $[\alpha]ff$

Again the Φ_i's are valid formulæ of the above form.

4.3 Issues with Interpreting HML Counter-Example

In this section we anticipate some problems with trying to understand what potential HML counter-examples could be saying about a target LTS.

4.3.1 Traces

There is a simple subset of HML formulæ that behave much like strings in DFAs. We call these traces after trace equivalence:

Definition 4.3.1 (trace)**.** *A trace is the subset of HML formulæ built only using $\langle\alpha\rangle$ for some $\alpha \in \Sigma$ an alphabet.*

Notice that the definition would seem to presuppose that a trace is a valid HML formula. For instance:

$$\langle a\rangle\langle b\rangle\langle a\rangle tt$$

is a valid trace whereas,

$$\langle a\rangle\langle b\rangle\langle a\rangle$$

would normally not be considered a trace. However for our purposes we will not make this distinction and consider formulæ that would be traces, were they not missing the atomic tt formula, as valid trances because they can be used to build other valid HML formulæ. For instance say we had a state in an LTS which satisfied a formula δ. Furthermore, suppose a trace to that state was the string aba. We would expect then the LTS to satisfy the corresponding trace $\langle a\rangle\langle b\rangle\langle a\rangle tt$. But we would also expect it to satisfy $\langle a\rangle\langle b\rangle\langle a\rangle\delta$.

It is in this sense that traces behave like strings. They allow us to access and to make statements about states in a LTS by prepending them to formulæ. If p is a state in an LTS we denote its trace as $\langle\alpha^*_p\rangle$, where the string α^* is the actual trace.

Like strings we can take prefixes and suffixes of traces:

Definition 4.3.2 (Prefix/Suffix)**.** *The i^{th} prefix of a trace $\langle\alpha^*_p\rangle = \langle\alpha_1 \ldots \alpha_n\rangle$ is the formula:*

$$\langle\alpha_1 \ldots \alpha_i\rangle tt$$

We denote it $(\langle\alpha^*_p\rangle|_i$ where $(\langle\alpha^*_p\rangle|_0$ is the empty trace. The corresponding i^th suffix is the formula:

$$\langle\alpha_{i+1} \ldots \alpha_n\rangle tt$$

which we denote $|\langle\alpha^*_p\rangle)_i$ where $|\langle\alpha^*_p\rangle)_n$ is the empty suffix.

A convention we shall use later (see the **Split** subroutine in Chapter 5), is that if δ_{pt} is a distinguishing formula for a state pt, then $pt(i)$ is a distinguishing formula for the state that represents the i^{th} prefix of the trace to pt.

4.3.2 The lack of a Distributive Law

HML formula, and all modal logics, do not have analogues of the distribution laws for the modal operators. For instance consider the following formulæ:

$$\begin{aligned} [a][a](\neg(\langle a\rangle)tt \vee \langle b\rangle tt) \vee\ [a][a](\neg(\langle b\rangle)tt \vee \langle a\rangle tt) \\ [a][a](\neg(\langle a\rangle)tt \vee \langle b\rangle tt \vee\ \neg(\langle b\rangle)tt \vee \langle a\rangle tt) \end{aligned} \tag{4.7}$$

It is easy to see that the second equation is equivalent to tt, and thus all states satisfy it. The first equation, which can be derived from the second by distributing the $[a][a]$ over the central $\vee$ operator, is not equivalent to tt. The difference is obvious: the first equation does not require both $\neg(\langle a\rangle)tt \vee \langle b\rangle tt$ and $\neg(\langle b\rangle)tt \vee \langle a\rangle tt$ to hold simultaneously on all aa paths – a far weaker statement.

However there is one important case where we can use a *kind* of distributive law which will be useful latter for our algorithm. If there is only one choice for an a-transition at a state then the following two formulæ express the same thing:

$$\begin{aligned} \langle a\rangle(\langle b\rangle tt \wedge\ \langle c\rangle tt) \\ \langle a\rangle\langle b\rangle tt \wedge\ \langle a\rangle\langle c\rangle tt \end{aligned} \tag{4.8}$$

Thus if a state has only one a-transition and we have a counter example of the form $\langle a\rangle(\bigwedge \varphi_i)$ we can consider each of the $\langle a\rangle(\varphi_i)$ formulæ separately, simplifying the algorithm.

4.3.3 Dealing with OR

We can use similar reasoning for $\bigvee$; however, it becomes slightly more complicated. Consider the meaning of the word *or*. If one makes the statement: "either *A* or *B* or *C* is true," the statement is weak in the sense that any subset of {A,B,C} could be true. In terms of thinking about how a learning algorithm could interpret such a statement, this means that when the equivalence-oracle returns saying the target satisfies $\phi \vee \psi \vee \varrho$, we need to determine which arguments ϕ, ψ, ϱ the LTS actually satisfies, and which are just decoys. For $[\alpha] \bigvee_i^n \varphi_i$, along each individual α-transition only some subset $T \subset \{1 \dots n\}$ of indices of the φ_i's represent φ_i's actually modeled by the target. This suggests that there exists a modified formula $\bigwedge_T \varphi_i$ satisfied by the target. Membership queries can help determine the set T to an extent. However this subset T could be different for each α-transition, meaning the set T is dependent on which α-transition we are considering. No single set T is guaranteed to work along all α-transitions. For a fixed α-derivative, once we know which of the φ_i's are true —that is, behaviours that state should exhibit— we can check each one individually to see if they themselves are counter-examples of the hypothesis, or if they are already modeled.

The procedure is as follows: faced with a counter-example $[\alpha] \bigvee_i^n \varphi_i$ at a state with several α-transitions, let us say we can identify each transition by using the distinguishing formula δ_j of the state it transitions to. Then for each φ_i we can test whether $\langle\alpha\rangle (\delta_j \wedge \varphi_i)$ is a true behaviour of the state in question. For each individual α-transition this leaves us with a subset of the φ_i's actually represented by the LTS. Let this subset be T. It needs to be said emphatically: this does *not* necessarily mean that in the target LTS there exists a

state that satisfies:

$$\langle\alpha\rangle\left(\delta_j \wedge \bigwedge_{i\in T} \varphi_i\right)$$

because what is a single state in the hypothesis may be accounting for multiple states in the target (this is important for the **Or-Elimination** subroutine defined in Chapter 5). What it does mean is that in the target there exists a set of k states transitioned to on α that all satisfy δ_j (the hypothesis thinks they are only one state, as δ_j represents the only currently known behaviours) and for k sets of indices $T_1 \cup \ldots \cup T_k = T$ there is a specific $l, 1 \leq l \leq k$ for each of these k states such that, that state also satisfies the formula $\bigwedge_{i\in T_l} \varphi_i$. Thus when confronted with a counter-example of the form $\bigvee_i^n \varphi_i$ we can turn it into a formula $\bigvee_T \varphi_i$. Then we arbitrarily select one of the $\varphi_m, m \in T$ not satisfied at the state in question to use as a correction (for $\bigvee \varphi_i$ to be a counter-example then none of them should be satisfied). Since the state in the hypothesis was representing k separate states, picking this φ_m to make the correction has the effect of fixing the corrected state to one of the k possible states that satisfies $\bigwedge_{i\in T_l} \varphi_i$ where $m \in l$. The fixing occurs because in the future φ_m will be used in the membership queries to identify this state.

4.3.4 Divining Non-Determinism

In this section we consider whether we can use HML formula to determine what actions must represent non-deterministic choices. To start, notice that given two formulæ δ and $\neg\delta$, no single state can satisfy both formula. What then do we make of the formula: $\langle\alpha\rangle\delta \wedge \langle\alpha\rangle\neg\delta$ if the target is telling us a state in our hypothesis should satisfy both?

Consider the following formulæ. Regarding the leftmost LTS in Figure 4.4, assume that we are told that we want the hypothesis LTS to now satisfy the formula $\langle ab\rangle(\langle b\rangle tt \wedge [a]ff)$ —but without removing any current states. The only way is to introduce a non-

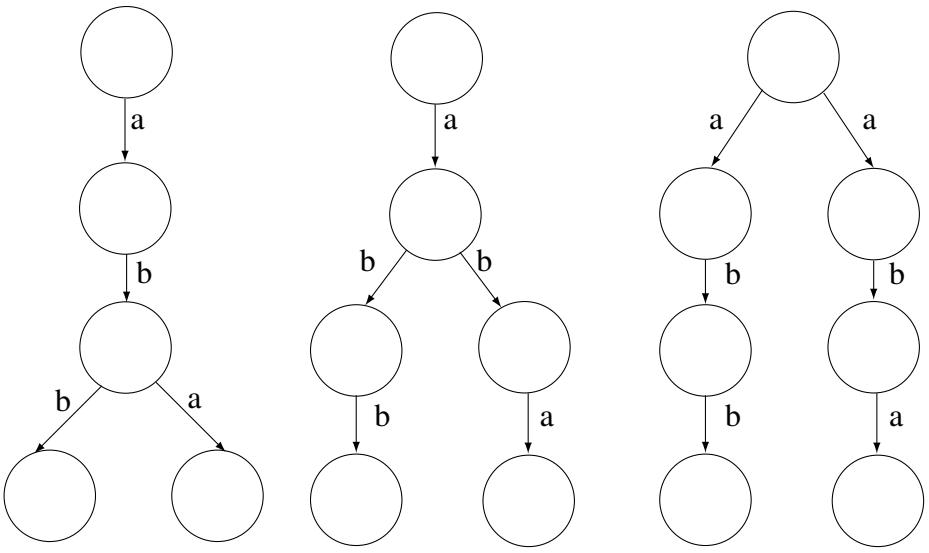


Figure 4.4: Possible Non-deterministic branching

deterministic action. These are the kinds of problems that will face us when designing a learning algorithm. For instance, if that formula represents behaviour existing in some target LTS but not our current hypothesis, we need a way to find where the non-determinism starts in the target. Consider the formula of Table 4.3 and what they might tell us about Figure 4.4:

Table 4.3: Divining Non-Determinism in Figure 4.4

Formula:	Satisfied by:	Formula:	Satisfied by:
$\langle ab\rangle(\langle b\rangle tt \wedge\ [a]ff\ )$	none	$\langle a\rangle\langle b\rangle[a]ff$	center/right
$\langle a\rangle(\langle bb\rangle tt \wedge \langle b\rangle[a]ff)$	center/right	$\langle a\rangle[b][a]ff$	right
$(\langle ab\rangle\langle b\rangle tt \wedge\ \langle ab\rangle[a]ff)$	center/right	$[a]\langle b\rangle[a]ff$	none

It might seem that the righthand formulæ of Table 4.3 are the best candidates for divining where to split the states. Unfortunately these formula can be misleading; undiscovered sections of the target can affect their values. The left-hand formulæ while less informative do tell us that regardless of undiscovered actions of the target, we can at least do some se-

quence of actions to reach a state that can do *both* actions of two different sequences (the two arguments of the $\wedge$-operator). For the above example, since $(\langle ab\rangle\langle b\rangle tt \wedge \langle ab\rangle[a]ff)$ is a counter-example we know it must be modeled by the target LTS; we already knew we should do $\langle ab\rangle\langle b\rangle tt$ and the counter-example told us we could do $\langle ab\rangle[a]ff$. The formula $\langle a\rangle(\langle bb\rangle tt \wedge \langle b\rangle[a]ff)$ tells us something interesting. After an a-transition we can reach a state that can do both: $\langle bb\rangle tt$ and $\langle b\rangle[a]ff$ action sequences. This is a new piece of information, and could help us place this split.

4.3.5 The problem of infinite behaviour

The previous subsection began to consider how we can interpret HML given as a counter-example to some hypothesis in a learning algorithm. Now we consider a well known short coming of HML. As we have said, HML was developed to model processes, but there are certain types of processes that HML is weak at modeling. For instance, there is no way to say succinctly that an action will eventually happen, and no way to say that an action can happen infinitely often.

In that second case, Figure 4.5 shows this insidious behaviour. Here the rightmost LTS is the target. Given the counter-example $\langle cc\rangle$ a sensible algorithm however would construct the leftmost LTS, because we know there is a state that transition on c to a state which has a c transition; the smallest such LTS is a state with a c-labeled self loop. A counter-example to this would be $\langle c\rangle[c]ff$. The middle LTS seems like the next safest choice. The problem is that it now contains an incorrect transition; one we would have to remove to reach the target. This seems to be a weakness in such an algorithm as an adversarial teacher could force extraneous computations.

It is not a simple task to distinguish between a LTS that can do an action infinitely

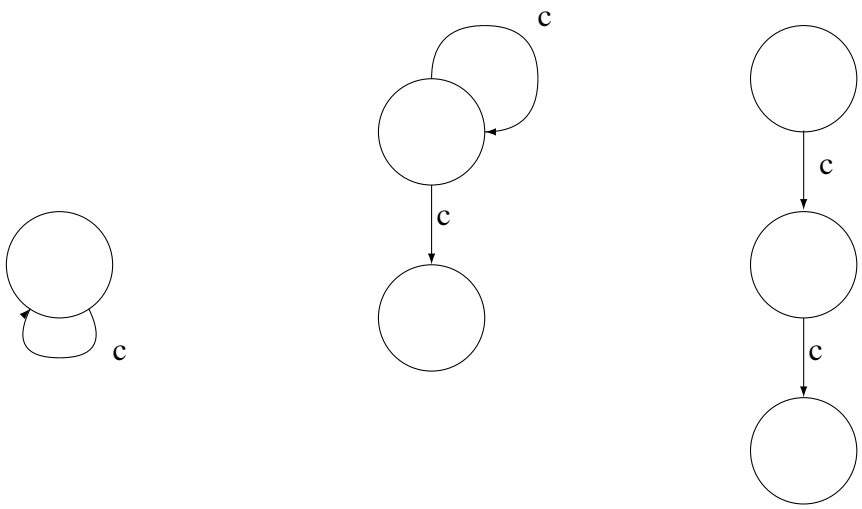


Figure 4.5: An incorrect hypothesis

often and one that can do one n times before it deadlocks. This is because there is no HML formula to distinguish infinite behaviour; however for each n there will be a formula of size $n+1$ ($\langle a_1 \ldots a_{n+1} \rangle$) that distinguishes between an infinitely recurring action and one that can only happen n times in series.

Certainly, strings are not sufficient to elucidate when a DFA enters a cycle, and yet Angluin's algorithm overcomes this problem. However, that was a different model, as a DFA had accepting states, which could force counter-examples to exhibit long-term behaviour: behaviour that passed through cycles. Knowing a priori that these accepting states exist allow Angluin's Algorithm to work forwards and backwards from the start state and an accepting state to help isolate cycles. When HML formula counter-examples elucidate behaviours ending inside a cycle without some future state to anchor future behaviours to, isolating a cycle is difficult. There is a much more powerful version of HML called μ-calculus that could overcome this problem.

While a full definition of modal μ-calculus is beyond the scope of this thesis, let us

briefly examine its two new operators. We adapt our definitions from [5]. We introduce the μ operator, also called the least fixed point operator, and ν, the greatest fixed-point operator. A fixed point of a function f, is a member, x, of the domain, such that $f(x) = x$. We can think of μ and ν as operators that search for these fixed points. So if we are to write an equation

$$\mu Z.f(Z) \tag{4.9}$$

This refers to finding the smallest fixed point Z of the equation $Z = F(Z)$. This notation allows us to write recursive equations —without recursion. Here is how we can use these new operators to define the notions of *Eventually* and *Always*:

$$\mu Z.P \vee ([-]Z \wedge \langle - \rangle tt) \tag{4.10}$$

$$\nu Z.P \wedge [-]Z \tag{4.11}$$

We can interpret the meaning of these equations as follows: a state of a LTS $st \models \mu Z.P \vee ([-]Z \wedge \langle - \rangle tt)$ if $st \models Z$ where $Z = P \vee ([-]Z \wedge \langle - \rangle tt)$. For this to be satisfied either:

1) $st \models P$, or,

2) $st \models [-]Z$ and $st \models \langle - \rangle tt$.

In the second case the formula $st \models \langle - \rangle tt$ ensures the current state has a transition; that means a state cannot vacuously satisfy $[-]Z$ by having no transitions. What this means is that st can only satisfy Z, if the actions P happen *eventually*. Likewise a state $st \models \nu Z.P \wedge [-]Z$ if $st \models Z$ where $Z = P \wedge [-]Z$. It is easy to see the only way this is satisfied is if P always holds down every reachable transition. We do not need to worry

about st vacuously satisfying $[-]Z$, as that would only happen if P itself could be satisfied by a state with no actions. The fact that the second equation uses a greatest fixed point has to do with the fact that we want P to be true forever, along paths infinite in size.

However, μ-calculus may be too powerful a query to allow. It still is possible that LTS with cycles could be learned with a weaker query. Certainly, if we fix some canonical form for the LTS, we may also be able to overcome the problem of infinite behaviour (see Section 6.3).

4.4 Proof of Hennesy-Milner Theorem

In this section we prove the main result that let us use HML formulæ to talk about bisimilarity.

4.4.1 Depth of Formulæ

To simplify the proof of the Hennesy Milner theorem of the next subsection, we want to be able to reason about the depth of the formulæ. We define the depth of a formula over reduced HML (due to Milner [14]).

Definition 4.4.1 (Depth)**.** *Let ϕ, ψ be a HML formula, let $\alpha \in \Sigma^*$. The depth of the formula ϕ is defined as:*

(i) $depth(tt) = 0$

(ii) $depth(\langle\alpha\rangle\phi) = depth(\phi) + 1$

(iii) $depth(\neg\phi) = depth(\phi)$

(iv) $depth(\phi \wedge \psi) = max\{\ depth(\phi), depth(\psi)\ \}$

4.4.2 Hennesy-Milner Theorem

This theorem justifies the importance of HML, as it demonstrates that HML formulæ distinguish non-bisimilar LTSs. The proof is adapted from Milner [14].

Theorem 4.4.2. *Two LTSs L_1, L_2 are bisimilar if and only if they satisfy exactly the same HML formulæ.*

Proof. The proof proceeds by induction on the depth of the formulæ. Recall a LTS satisfies a HML formula if the LTS's start state satisfies the formula. We consider the '*if*' and '*only if*' directions in turn:

($\Rightarrow$)

Assume $L_1 \simeq L_2$.

Base Case:

The only depth zero formulæ are *tt* and $\neg$*tt*. Both $L_1 \models tt$ and $L_2 \models tt$ vacuously. Likewise, neither satisfy $\neg tt$.

Inductive Hypothesis

Assume that for all formulæ of depth less then k (integer k) the theorem holds. Let ϕ be a formula of depth k. Without loss of generality assume $L_1 \models \phi$. We consider the three possible outermost operators of ϕ:

(i) if $\phi = \neg\phi'$ we can in turn consider ϕ' as it has depth k as well.

(ii) if $\phi = \varphi \wedge \psi$ then either φ or ψ has depth k, consider this formula instead.

(iii) if $\phi = \langle\alpha\rangle\phi'$ for $\alpha \in \Sigma^*$, then let S_{01}, S_{02} be the start states of L_1 and L_2 respectively. Then $\exists\ S'_{01}$ such that $S_{01} \xrightarrow{\alpha} S'_{01}$ ($S'_{01} \models \phi'$), and by bisimilarity we also must have a

S'_{02} such that $S_{02} \xrightarrow{\alpha} S'_{02}$ and $S'_{01} \simeq S'_{02}$. By the induction hypothesis since $S'_{01} \models \phi'$ and depth(ϕ) $< k$, then $S'_{02} \models \phi'$. Clearly then $S_{02} \models \phi$.

($\Leftarrow$)

Assume $L_1 \not\simeq L_2$.

We want to construct a formula ϕ such that $L_1 \models \phi$ but L_2 does not.

Here we need to use the idea of k-bisimilarity to match the degree to which two LTSs are bisimilar with the depth of the formulas. In particular, if k is the minimum such k such that two LTSs are not bisimilar, then we can find a depth k formula, or less, to distinguish them.

Base Case:

Since every state is 0-bisimilar, no HML formula can distinguish them. We start with 1-bisimilarity. For two states to be 1-bisimilar they must be able to match each others α-derivatives leading to 0-bisimilar states. This simplifies to just being able to match α-derivatives, which is the notion of trace equivalence. If two LTSs are not trace equivalent, and $\alpha \in \Sigma^*$ is a trace on which they differ, then $\langle\alpha\rangle tt$ is a HML formula that distinguishes them, and is of depth 1.

Inductive Hypothesis

We start by assuming that for all non k-bisimilar LTSs we can find a formula of no more then depth k. Assume $L_1 \not\simeq_{k+1} L_2$. By the assumption we know that for start states S_{01}, S_{02} $\exists \alpha \in \Sigma^*$ such that $\exists S'_{01}$ such that $S_{01} \xrightarrow{\alpha} S'_{01}$ but for all S'_{02} such that $S_{02} \xrightarrow{\alpha} S'_{02}$ we find $S'_{01} \not\simeq_k S'_{02}$, by the definition of bisimilarity. Call the set of such $S'_{02}, D = \{D_1 \ldots D_n\}$.

Using the inductive hypothesis we can find formulæ ϕ_i for each D_i, such that $S'_{01} \models \phi_i$ and $D_i \not\models \phi_i$, where the depth of each ϕ_i is no more than k. Then the formula:

$$\Phi = \langle\alpha\rangle \bigwedge_i D_i \tag{4.12}$$

is a HML formula of no more then depth $k+1$ that distinguishes L_1 and L_2.

Note: we implicitly assume that each state has a finite number of exiting transitions, as we are assuming of our LTSs.

□

4.5 A Partition of HML formulæ

A problem that prevents us from extending Angluin's algorithm to LTS using HML formulæ is that the states of a minimal LTS do not partition HML formulæ, whereas the states of a DFA partitions strings. In this section we consider what it takes to partition HML formula. We start by defining set-bisimilarity:

Definition 4.5.1 (Set-Bisimilarity)**.** *Two sets of states, U and V, of an LTS are bisimilar, denoted $U \simeq V$, if and only if*

- *for every state $u \in U$ there exist $v \in V$ such that u and v are bisimilar states.*
- *for every state $v \in V$ there exist $u \in U$ such that u and v are bisimilar states.*

For every HML logic formula ϕ, define $||\phi|| = \{s | s \models \phi\}$ where s is a state of the underlying LTS. We can then define the equivalence relation $\equiv$ as follows:

$$\equiv = \{(\phi, \psi) |\ ||\phi|| \simeq ||\psi||\} \tag{4.13}$$

Clearly $\equiv$ is an equivalence relation since $\simeq$ is an equivalence relation. Moreover $\equiv$ is a partition of HML formulæ. The question is, can we build an LTS where the states represent the partitions of $\equiv$? The problem is that the size of the partition could be exponential in the number of states of the underlying LTS, let alone deciding on a rule to place transitions. Explicitly, if the number of states is n, then the number of possible partitions is the number of subsets with cardinality n, or 2^n. Consider for instance a minimal LTS, where each state satisfies a unique HML formula. In particular consider a three state LTS. Let us call the states s_1, s_2, s_3. Furthermore, the only two transitions are an a-transition from s_1 to s_2, and a b-transition from s_2 to s_3. Let us call the unique set of distinguishing formula for s_1, s_2, s_3: $\delta_1, \delta_2, \delta_3$ respectively. Then for instance $s_1 \in ||\delta_1||,\ s_2 \in ||\delta_2||,\ s_3 \in ||\delta_3||$. The equivalence classes of set bisimilarity (denoted $[\cdot]_{\equiv}$) would be the states of this supposed LTS. For instance $[\ ||\delta_2||\ ]_{\equiv}$ would be a state. Consider now the set of all possible subsets of $\{s_1, s_2, s_3\}$, namely $\{\{\}, \{s_1\}, \{s_2\}, \{s_3\}, \{s_1, s_2\}, \{s_1, s_3\}, \{s_2, s_3\}, \{s_1, s_2, s_3\}\}$. The question is: is it possible that for any of the elements, S, of this set, there does not exist a HML formula φ, such that $||\varphi|| = S$? Under this view point, the sets of states are related to the set of formula satisfied only at those combination of states. Thus for any of the sets of states to be bisimilar, their respective elements would have to be bisimilar states. Since the underlying LTS is minimal this is only possible if for some subset of states there was no HML formula satisfied at that collection of states. Then so long as we can show that each set of states can be associated with some formula it uniquely satisfies, we can convince ourselves that no two sets of states are bisimilar by the Hennesy-Milner Theorem. Thus any LTS whose states partitioned the HML formula could have states exponential in size of the original LTS. This is the case for the example LTS. Here is one list of formulæ that are satisfied at each subset of states:

- $\{\}$: ff is uniquely satisfied at that combination of states
- $\{s_1\}$: $\langle ab\rangle tt$ is uniquely satisfied at that combination of states
- $\{s_2\}$: $\langle b\rangle tt$ is uniquely satisfied at that combination of states
- $\{s_3\}$: $[-]\mathit{ff}$ is uniquely satisfied at that combination of states
- $\{s_1, s_2\}$: $\langle a\rangle tt \vee \langle b\rangle tt$ is uniquely satisfied at that combination of states
- $\{s_1, s_3\}$: $[b]\mathit{ff}$ is uniquely satisfied at that combination of states
- $\{s_2, s_3\}$: $[a]\mathit{ff}$ is uniquely satisfied at that combination of states
- $\{s_1, s_2, s_3\}$: tt is uniquely satisfied at that combination of states

The only states of the LTS we care about are the ones associated with the singleton sets $\{s_i\}$, as these are the states of the target LTS. Thus, on the surface, the problem does not seem substantially different from before.

Chapter 5

Learning LTS

We begin this chapter with some obvious results following directly from previous work. We then address some of the anticipated problems with learning LTSs. We use this as a spring board to design the algorithm from the ground up. We consider three subsets of LTSs: trees, DAGs, and full LTSs. We focus this chapter on tree LTSs. We conclude by proving first that the algorithm works if the target is deterministic. This acts as a sanity check, as learning deterministic trees should be fairly easy. We then expand the proof to include non-deterministic trees, and finish with an example of the algorithm.

A notice on notation used in this chapter: we use $i's$ and j's to refer to arbitrary states. We use pt and st to refer to known and connected states. We use p and p_i, and s and s_i to refer to a parent state p (respectively s) and its arbitrary children p_i (respectively s_i).

5.1 Preliminary Results

As promised in section 1.4 we can use a simple reduction to extend some results about DFA learning to LTSs. This is a result of being able to model DFAs with LTSs. We will transform an instance of learning DFAs using strings to an instance of learning LTSs with HML. This reduction will work for any algorithm that presents us with examples $\omega \in POS \bigcup NEG \subset \Sigma^*$, where POS is a set of examples of a regular language and NEG a set of counter-examples. We can turn these strings into traces of the LTS. It will not work in the case of membership queries because there is no obvious way to turn any

HML formulæ into strings to feed to a membership query oracle for DFAs.

The reduction is very simple. First given a DFA over the alphabet Σ we consider a LTS over the alphabet:

$$\Sigma^{+} = \Sigma \bigcup \{accept, reject\} \tag{5.1}$$

Given disjoint sets *POS* and *NEG* we can create two new sets:

$$\begin{aligned} POS' = &\{\varphi = \langle\alpha_1\rangle \ldots \langle\alpha_k\rangle\langle accept\rangle tt \mid \omega = \alpha_1 \ldots \alpha_k \ \in POS\} \\ &\bigcup \{\varphi = \langle\alpha_1\rangle \ldots \langle\alpha_k\rangle\langle reject\rangle tt \mid \omega = \alpha_1 \ldots \alpha_k \ \in NEG\} \end{aligned} \tag{5.2}$$

$$\begin{aligned} NEG' = &\{\varphi = \langle\alpha_1\rangle \ldots \langle\alpha_k\rangle[reject]ff \mid \omega = \alpha_1 \ldots \alpha_k \ \in POS\} \\ &\bigcup \{\varphi = \langle\alpha_1\rangle \ldots \langle\alpha_k\rangle[accept]ff \mid \omega = \alpha_1 \ldots \alpha_k \ \in NEG\} \end{aligned} \tag{5.3}$$

The set POS' puts the restriction on the target LTS that any accepting trace for a DFA leads to an *accept* action in the associated LTS. Likewise, traces to non-accepting states lead to *reject* actions. Similarly NEG' ensures that accepting traces never lead to a *reject* action in the LTS. Likewise, rejected traces do not lead to *accept* actions. This ensures that rejecting states and accepting states are never bisimilar.

We now want to prove that if we could Predict_LTSs then we could Predict_DFAs (recall Problem 2 from Chapter 1). Under the assumption $P \neq NP$, if we cannot do the latter, we can conclude that we cannot do the former. To complete the proof consider the following question: If the algorithm Predict_LTS existed and we fed it the sets POS' and NEG' could we transform the resultant LTS back into a DFA?

First we note that the LTS we learned is more than likely non-deterministic. However if we cannot predict the DFA efficiently in time polynomial in the size of the minimum DFA, then we certainly cannot predict the NFA in time polynomial in the size of the

minimum NFA, which is no bigger then the size of the minimum DFA. Thus we conclude that negative learning results for DFA carry over to LTS as long as no membership queries are involved.

We now turn our focus to Assisted Learning; the fact that we know Angluin's Algorithm learns DFAs using strings says nothing of the situation for the more general case of LTSs. However they are not so dissimilar that we cannot try and construct a new algorithm by analogy.

5.2 Motivation of Algorithm

We will learn a LTS on a state by state basis, using counter-examples to discover new states. Like Angluin's algorithm, we will keep track of states using a set of Access formulæ. As stated, we will use HML formulæ for counter-examples. Thus what we want is to develop a set of HML formulæ, and a relation over them that forms an equivalence class of states —each equivalence class is a state in the hypothesis LTS, and all transitions of the hypothesis LTS are consistent with known transitions.

Unfortunately, the chief difference between DFAs and LTSs is that DFAs induce a natural equivalence relation over *all* strings, MNR. For the states of LTSs, the natural equivalence relation they induce is only over some set of yet to be known distinguishing formulæ intrinsic to the target LTS. For instance a formula such as $[a]tt$ is likely satisfied by many states; therefore, it would almost never be a distinguishing formula. Another strategy is to try and rig a canonical form for LTSs. For instance, Angluin's algorithm works on the implicit assumption that every state has a transition defined for every character in the

associated alphabet. In terms of transition functions:

$$\forall\, s \in S, \forall a \in \Sigma, \exists s_i \text{ and } s \xrightarrow{a} s_i$$

Moreover DFA have two distinct types of states, accepting and non-accepting; and, all string behaviour can be anchored to one of these two outcomes.

We can imagine other ways of overcoming these obstacles, such as making requirements on the expressiveness of counter-examples. Yet agreeing on these canonical LTS and restricted queries are not in the spirit of the original question: Can we develop an assisted learning algorithm for LTSs? And while we cannot completely avoid making concessions, it seems then we must at least rely on developing this set of unknown distinguishing formulæ for each state from the counterexamples we receive. For a state pt, we will denote its distinguishing formula as: δ_{pt}.

These distinguishing formulæ cannot be known a priori. Furthermore, during the iterative process of developing a hypothesis LTS, HML formula currently known to distinguish between states in the hypothesis may not remain valid distinguishing formulæ. As more information about the long term behaviour of the LTS is discovered we cannot be sure such formula will continue to distinguish current states from future states. Thus if we are to maintain a set of distinguishing formulæ, it will be necessary to retroactively update them. Our goal is to maintain the following property:

Definition 5.2.1 (Correctly-Distinguishing Property)**.** *Given a state pt with multiple α-transitions to a set of states P, we will say these α-transitions have the Correctly-Distinguishing Property if for any two states $p_i, p_j \in P$, with respect to the hypothesis LTS:*

1) $p_i \models \delta_{p_i} \ \wedge \ p_j \not\models \delta_{p_i}$

2) $p_j \models \delta_{p_j} \ \wedge \ p_i \not\models \delta_{p_j}$

3) no more then one of the states is a dead state.

Recall for a state pt we denote a trace $\alpha_1 \ldots \alpha_n$ to that state as $\langle \alpha^*_{pt} \rangle = \langle \alpha_1 \rangle \ldots \langle \alpha_n \rangle$. Necessitated by the non-determinism of LTS, the access formula for a state pt is actually the ordered pair : $(\langle \alpha^*_{pt} \rangle, \ \delta_{pt})$ —the first element is a trace, the second a HML formula. If all the states have the Correctly-Distinguishing Property for all characters in Σ we can use the following membership query to test if the state pt can admit the behaviours φ, described in terms of an HML formulæ:

$$\text{membership}(\langle \alpha^*_{pt} \rangle (\delta_{pt} \wedge \varphi)) \tag{5.4}$$

We will expand on this notion later after we develop a stronger definition of trace.

Definition 5.2.2 (coincide/consistent)**.** *For a fixed state* pt *with distinguishing formula* δ_{pt}*, we say* pt *coincides or is consistent with* φ *if the query:*

$$\text{membership}(\langle \alpha^*_{pt} \rangle (\delta_{pt} \wedge \varphi)) \tag{5.5}$$

returns true.

Note this definition assumes that the distinguishing formula is correct. After the stronger definition of total traces is introduced (Definition 5.3.2) the notion of what *consistent* means will be strengthened.

Like Angluin's algorithm we will coax the counter-examples into these distinguishing-formulæ. This leads us to our first concession we must make: the counter-examples need to be polynomial in the size of the target LTS. The time it takes to interpret a counter-example is a lower bound on the running time of the algorithm —this concession is impossible to

avoid. Secondly, we will require non-redundant counter-examples —the distinguishing-formula motivated in section 4.2.6 fulfil this requirement. Again, since formulæ of this type could easily be computed from any counter-example, we may as well require this form to simplify the analysis.

We will not yet require the target LTSs to be in a canonical form or put additional restrictions on the counter-examples. We will briefly entertain these ideas in Chapter 6.

5.2.1 The problem of Overlapping Distinguishing Formulæ

In this subsection we consider an insidious complication to the Correctly Distinguishing Property: that the distinguishing behaviours may not be mutually exclusive. This property is illustrated in Figure 5.1. All this means is that two distinguishing formulæ can share a common sub-formula; or rather, that substantial amount of behaviour can be shared between non-bisimilar states.

However, if we are judicious in maintaining the Correctly-Distinguishing Property for each hypothesis we develop, we can avoid this complication. First, why do we care about the Correctly-Distinguishing Property? Obviously to distinguish between paths that mirror each other in the following sense:

Definition 5.2.3 (Non-Deterministically Mirrored States/Paths)**.** *We say that two states non-deterministically mirror one another if they can both be reached by a single trace. A path of transitions:*$\alpha_1 \rightarrow \cdots \rightarrow \alpha_n$ *is non-deterministically mirrored by another path:* $\beta_1 \rightarrow \cdots \rightarrow \beta_n$*, if the intermediate states non-deterministically mirror each other.*

Consider Figure 5.1. We show several examples of mirroring traces leading to multiple paths. Each path is labeled with the distinguishing formula of its first state. Assuming that

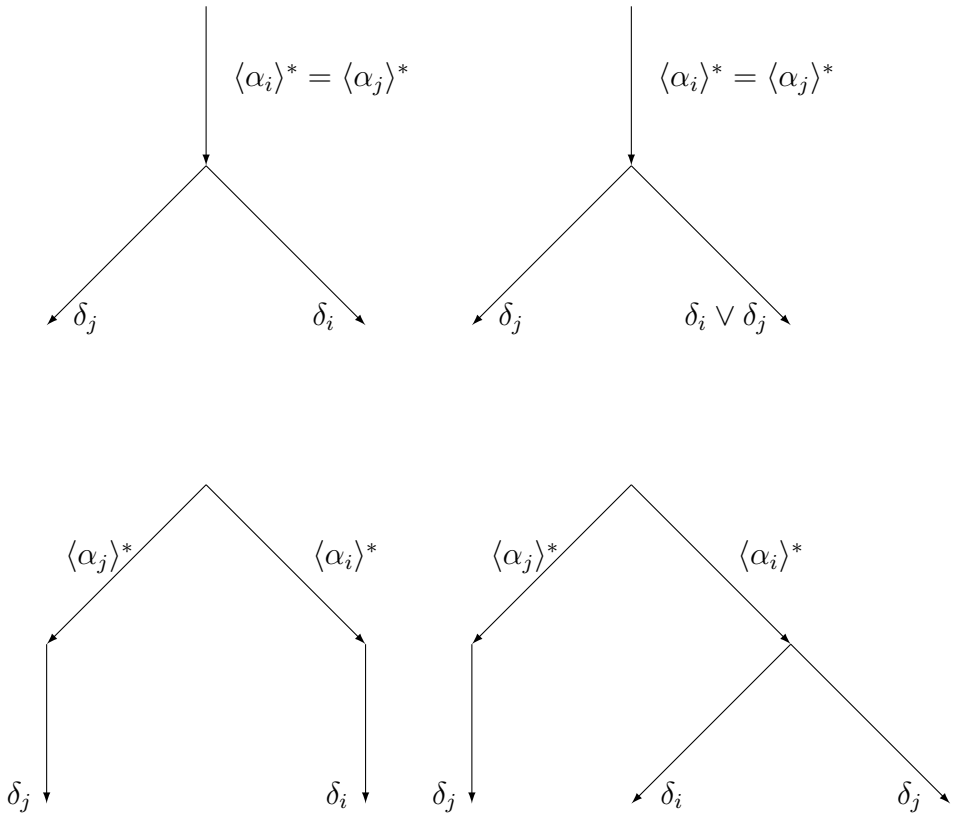


Figure 5.1: Actions which are not mutually exclusive

the δ_i and δ_j branches of Figure 5.1 are Non-Deterministically Mirroring one another, then at some point in the past we would to have had split them. Thus the distinguishing formulæ δ_i and δ_j would have to be mutually exclusive. Equivalently they have the Correctly-Distinguishing Property.

5.3 Examples

In this section we try to anticipate some problems we are likely to encounter when designing the learning algorithm. It expands on Section 4.3.

5.3.1 Discovering New States

It turns out there are two ways we can make discoveries about a hypothesis. We could find that a state should satisfy the formula $\langle\alpha\rangle tt$. From this we can assume it has an α-transition. But how can we make the best choice about where to place that transition? Does it go to an existing state or does it go to a new state?

Consider a more complex counter-example: $\langle\alpha\rangle\langle ab\rangle[c]\langle b\rangle tt$. Here we have plenty of information about the state that the α-transition goes to. But we are faced with the same problems: should we send it to an existing state? Or to a new state? Here we consider the case in which it should be sent to a new state. If that is the case we will send the transition to the dead state. This important to point out because in our algorithm the dead states serve a dual purpose. On one hand it actually represents one of the few states we know in most cases exists beforehand, and we can fashion a priori its distinguishing formula. On the other hand, by having no actions it serves as a placeholder for states with unknown actions. That is, if we are certain a transition leads to a new state, but unsure of what that state's abilities are, by sending the transition to the dead state we force the counter-examples to provide information on the actions of the new state. Thus we must treat transitions entering the dead state as a special case.

This creates a specific problem. For our algorithm we will want to not give the dead state any specific distinguishing-formula. Thus $\delta_{dead} = \emptyset$. However many of those states

we tentatively send to the dead state will have known distinguishing formulæ. For instance we may have discovered a new state distinguished from a previous state by the formula $[\alpha]ff$. Since the dead state satisfies $[\alpha]ff$, we cannot rule out the possibility that this new state is the dead state. However, it could very well be a totally new separate state; thus we do not want to forget this distinguishing formula. In the future we may need it to maintain the correctly distinguishing property. To achieve this we define the notion of a Tentative Distinguishing Formula:

Definition 5.3.1 (Tentative distinguishing formulæ)**.** *Tentative distinguishing formulæ are distinguishing formulæ applied to transitions going to the dead state. When a transition going to the dead state is asked for the distinguishing formula δ of the next state, if that transition has a tentative distinguishing formula, we return that formula as the distinguishing formula for the next state, otherwise we would return $\emptyset$. For a transition $\xrightarrow{\alpha}$ we denote its tentative distinguishing formula δ as the transition $\xrightarrow{\alpha\{\delta\}}$*

If and when these tentative transitions are found to be actual states, we add the tentative distinguishing formula to the new state's distinguishing formula at the time of its creation. Note: we ignore Tentative Distinguishing Formulæ otherwise, and we remove them when we send such transitions to actual states.

5.3.2 Making Non-Deterministic Choices

This subsection contains two examples of the pit-falls of interpreting a counter-example for a given hypothesis. Consider the LTS of figure 5.2. Let us examine two scenarios for this LTS:

Example 1:

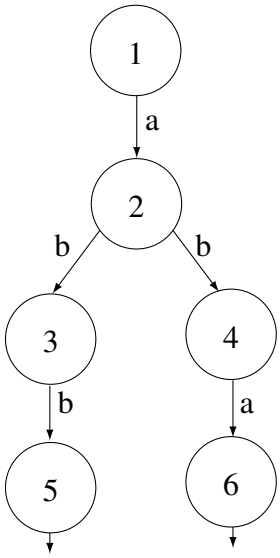


Figure 5.2: Example hypothesis

If the counter-example we receive is $\langle a\rangle((\langle bb\rangle tt) \wedge (\langle bac\rangle tt))$ it seems logical to think that the counter-example and the hypothesis disagree at state ⑥ —because it lacks a 'c' action. But it could just as well mean state ③ is lacking an 'a' action. Had we received the more explicit counter-example $\langle a\rangle((\langle b\rangle(\langle b\rangle tt \wedge [a]ff)) \wedge (\langle b\rangle([b]ff \wedge \langle ac\rangle)))$ we could be certain the disagreement was at state ⑥.

Example 2:

Consider now the counter-example $\langle a\rangle[b]\langle b\rangle tt$. Again it seems clear that state ④ is in disagreement because it fails to admit a 'b' actions. Had we previously been told that state ③ should not admit a 'b' action then it would be either state ① or ② in error. That is, *Error* in the sense that we are currently failing to describe some non-deterministic behaviour which mirrors states ① and ②.

To distinguish non-determinism, we will need to clarify counter-examples, by modifying them to include the distinguishing strings (see Definition 5.3.2). The second example suggests we should not only keep track of the actions we can do, but also that, those actions which we know we cannot do are themselves necessary parts of distinguishing formulæ.

Example 1 suggests that if given the counter-example $\langle a\rangle\,((\langle bb\rangle tt) \wedge (\langle bac\rangle tt))$, after the initial 'a'-action we must decide between two 'b'-actions and two sub-formula of the counter-example $(\langle bb\rangle tt) \wedge (\langle bac\rangle tt)$ both with 'b'-actions next in their sub-formulæ of the '$\wedge$' sub-formula. Since we had previously distinguished states 3 and 4, we must have distinguishing formulæ δ_3, δ_4 for them. Thus, we could have performed the following membership queries:

i) $\langle ab\rangle(\delta_3 \wedge \langle b\rangle tt)$

ii) $\langle ab\rangle(\delta_4 \wedge \langle b\rangle tt)$

iii) $\langle ab\rangle(\delta_3 \wedge \langle ac\rangle tt)$

iv) $\langle ab\rangle(\delta_4 \wedge \langle ac\rangle tt)$

We would find that items i) and iv) are true, correctly modifying the sub-formula to:

$$(\langle b\rangle(\delta_3 \wedge \langle b\rangle tt)) \wedge (\langle b\rangle(\delta_4 \wedge \langle ac\rangle tt)) \tag{5.6}$$

The prescription for this situation is to develop a subroutine **Which**, whose purpose is to decide which non-deterministic choice best coincides with what the counter-example is illustrating. **Which** uses the idea from Section 4.3.2 to split apart $(\bigwedge \varphi_i)$ formulæ and examine each φ_i separately. Using the membership queries of Equation 5.4 we can further split apart $(\bigwedge \varphi_i)$ by associating non-deterministic transitions among the φ_i's with their

known distinguishing-formulæ. We also need to develop a more rigorous type of trace that encodes these non-deterministic decisions. We call this new type of trace a total trace:

Definition 5.3.2 (Total Trace)**.** *The total trace to a state* st *with trace* $\langle \alpha^*_{st} \rangle$ *is denoted* $\langle\!\langle \alpha^*_{st} \rangle\!\rangle$*. If the trace is:*

$$\alpha^* = \alpha_1 \alpha_2 \dots \alpha_n$$

there exists some sequence of indices $T \subset \{1 \dots n\}$*,* $T = \{t_1 \dots t_m\}$ *that includes all the transitions of* $\langle \alpha^*_{st} \rangle$ *that are non-deterministic choices:*

$$\alpha_{t_1}, \dots, \alpha_{t_m}$$

which transition out of states $st_{t_1} \dots st_{t_m}$*. Then the total trace is a function* $\langle\!\langle \alpha^*_{st} \rangle\!\rangle(\varphi)$*:*

$$\langle\!\langle \alpha^*_{st} \rangle\!\rangle(\varphi) = \langle \alpha_1 \rangle \dots \langle \alpha_{t_t - 1} \rangle (\delta_{st_{t_1}} \wedge (\langle \alpha_{t_1 + 1} \rangle \dots (\varphi)) \dots) \tag{5.7}$$

where for each state with a non-deterministic choice, we encode the necessary choice to get to the next state.

We can also define a notion of prefixes/suffixes for total traces. The i^{th} prefix of a total trace to state pt, is the total trace to the i^{th} prefix of the trace to pt. Likewise the i^{th} suffix of the total trace to a state pt is the total trace of the i^{th} suffix of the trace to pt. Note further that in the case that the LTS is a tree, the total trace is unique; otherwise it may or may not be unique. Regardless of uniqueness, we can easily compute the total trace to a fixed state by following the path to that state, and for each non-deterministic choice we make, add the distinguishing formula of the subsequent state to the trace.

As claimed before using this strengthened notion of traces, we can think of a stronger idea of consistency:

Definition 5.3.3 (coincide/consistent)**.** *For a fixed state* pt *with distinguishing formula* δ_{pt}*, we say* pt *coincides or is consistent with* φ *if the query:*

$$\text{membership}(\langle\langle \alpha^*_{pt} \rangle\rangle (\delta_{pt} \wedge \varphi)) \tag{5.8}$$

returns true.

5.4 Variants of LTS

Our examples are suggesting an algorithm decidedly un-Angluin. First and foremost Angluin's algorithm relied on the existence of accept/reject behaviour, giving her counter-example strings which showed long term behaviour, thereby allowing us to learn *distant* states. By learning distant states we mean knowing a state exists, and knowing of a trace to it, though we have yet to discover the shortest trace to it. Our counter-examples can be as short as $\langle a \rangle tt$, allowing us only to learn *near* states. Again, we would need an artificial scheme to force longer counter-examples. The disadvantage of learning near states is that it makes it difficult for the algorithm to recognize when a *new* state is in fact a new transition forming a circuit. Because of this problem we will limit designing the learning algorithm to a subset of LTSs.

We are going to develop the algorithm incrementally for three increasingly more complex variants of LTS. Each one is a restriction of regular LTSs:

(i) The LTS is a labeled directed tree

(ii) The LTS is a labeled directed acyclic graph

(iii) The LTS is a labeled directed graph

These classes warrant more explanation as no minimal LTS would ever form a tree, for a trivial reason: all leaves are deadlocked states and thus bisimilar. By a directed tree we mean that the LTS's transitions only converge at the dead state. That is, the LTS would be a tree if we split the dead state into leaves for each path leading to it. A directed acyclic graph has no directed cycles. If we ignore directions it may have cycles or it may be a tree. Finally, all finite LTSs are directed graphs, so the third category includes every LTS we could hope to learn.

The main idea of the algorithm is simple: we start with a modified model-checker which compares the distinguishing formula of the counter-example against the current hypothesis. As we follow the transitions in the LTS, we peel off the outer connectives of the counter-example to derive new sub-formulæ that must be satisfied by each successive state. We refer to all these sub-counter-examples as the *current-counter-examples* or *cce*. Thus at each stage of interpreting the counter-example we are comparing the cce to a state in our hypothesis. Let us now consider the individual concerns of the three main categories of LTS:

5.4.1 Labeled Directed Trees

Here we sketch an algorithm on how to deal with directed trees. LTS that are directed trees are the easiest to learn because all new behaviour elucidated by counter-examples must necessarily be those of new states or behaviours eventually leading to a single deadlocked state. Thus, we do not need to worry about distinguishing new states from every previously learned state. Tree LTS look like tree graphs where all the leaves have been merged into a single state.

The point of tree LTSs is that they let us focus on discovering non-deterministic be-

haviours; however, they are contrived in that we know all states yet to be learned are not bi-similar to any learned states. As an example, if we split a state we know that the successor states will stay split —ensuring that we will eventually discover more distinguishing behaviour, even if the two split paths start to look the same again. This simplifies maintaining the set of distinguishing formulæ. The task is simplified even further by the fact that the structure of the HML formlulæ themselves are tree-like. Main connectives of the formula act as root nodes for a subtree, with each argument of $\bigvee$ and $\bigwedge$ suggesting a branch. Never mind that some of these arguments may be suggesting the same branch.

However, this creates an annoying problem not present in less contrived LTSs. Figure 5.3 demonstrates the problem: forcing the hypothesis itself to be a tree means that for some intermediary hypothesis we may be forced to keep separate states we know are not bisimilar. In the case of Figure 5.3, assuming the lefthand LTS is some hypothesis, were we given the counter-example:

$$\langle b \rangle (\langle ab \rangle tt \wedge \langle aa \rangle tt) \tag{5.9}$$

this would be enough to discover some of the middle states on the righthand LTS. Notice however we have yet to learn those x, y-transitions which distinguish the middle states from the lower states. However, we know prior to learning of the x, y-transitions that these middle and lower states are not bisimilar; this is because they are on different branches and the target is a tree. They only appear bisimilar in a hypothesis, because we have not learned enough of the two branches to distinguish them. The problem is of our own making.

The total trace serves as a remedy to this problem, since we will encode all the non-deterministic choices we will make following the path along either branch. Total traces

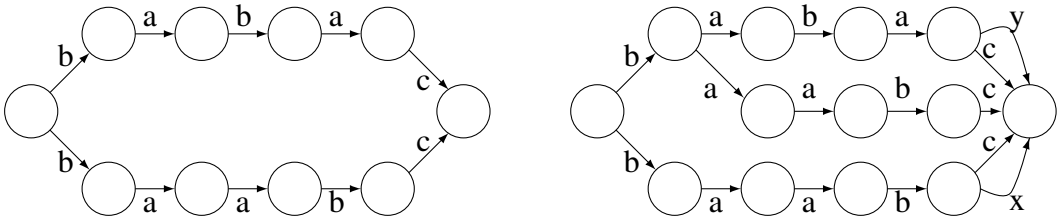


Figure 5.3: The Problem with Trees

uniquely identify all states along both branches because the first states after a split will be guaranteed to be distinguished.

A final note, despite their contrived nature, tree LTSs are relevant. Consider any *evolutionary process*. When we describe a system as evolving we generally mean two things:

- The system is not regressive

- The system is not convergent

There is no specific goal in an evolutionary system, just continued change. By 'not regressive' we mean that a system does not go back on itself. This suggests an acyclic process. By 'not convergent' we mean that we do not expect divergent systems to necessarily evolve into the same system (analogues not homologues). This suggests evolution produces a tree like process. Tree LTS can model these evolutionary processes.

5.4.2 Directed Acyclic Graphs

With these LTSs we need to be careful that new states we discover are not in fact previously learned states. In particular, split paths can again merge later on in the LTS. There is still a single deadlocked state so this can help us anchor paths between it and the start state. Acyclic processes are not at all contrived, and do not suffer from the problem of Figure 5.3. This is because any states that appear bisimilar would remain as a single state. There is

none of the artificial separation caused by descendant states of known non-bisimilar states being kept apart because we know the target is a tree. Thus partial distinguishing-formulæ and not total traces, are sufficient in describing any fixed state. Since states now have multiple paths to them, total traces will still serve a purpose.

5.4.3 General LTS

We do not solve the problem of learning general LTSs by the end of this thesis; however, we can speculate here about how to deal systems which potentially never halt. The only conceivable way of dealing with this is to assume that any behaviour that appears to be repeated is in a cycle; in the case where there is a choice, go with the smallest possible cycle. If it is not a cycle then there is a HML formula to refute that claim. If however it were a cycle but our hypothesis never assumed cycles, our learning algorithm would run forever. Because we are not using μ-calculus queries or a canonical form for LTSs, learning LTS with cycles presents a problem.

5.4.4 A Thought Experiment

We will use the following idea to inspire our design for learning the target concepts. We are going to view the algorithm as the reverse of the following process: taking sub-LTSs. In each case, we are going to develop a class of sub-LTSs; for example, we will define sub-Tree-LTSs. These sub-LTSs will have several important properties: they will be connected; all states will be reachable through transitions from the start state; they will be the same type (Tree, DAG, General) as the target; they will combine states that appear bisimilar; and finally, they will represent precursors to the target. We call sub-LTSs arising from this kind of procedure ‘effective’ sub-Tree-LTSs. The idea is that the process of making

a subtree is effectively the same as removing a portion of the target which the hypothesis has yet to learn about. This is a kind of un-learning, perhaps a *forgetting algorithm*. As an example, we begin with a definition of effective sub-Tree-LTSs for deterministic tree-LTSs:

Definition 5.4.1 (Effective Sub-LTS for Deterministic Tree-LTSs)**.** *An effective sub-Tree for a deterministic Tree-LTS is any Tree-LTS formed by the following process:*

- *We select an arbitrary set of states, not including the start state*
- *For each state we remove every descendant state and transition starting from this state if it has not already been removed.*
- *Any states that now appear bisimilar are combined, including all new leaf states with the caveat that combining states cannot destroy the tree property.*

The last step is not necessarily deterministic, therefore the procedure will not produce unique LTSs. However, any LTS that results from this procedure is an effective sub-LTS. In Section 5.6.4 we define Effective Subtrees for all Tree-LTS.

5.5 Assisted Tree-LTS Learning

We present an outline of the algorithm, denote the target LTS S, and the current hypothesis LTS $\hat{S}$. We denot the current state *pt*. Let cce denoted the current counter-example. This is the sub-formula of the counter-example λ that the current state *pt* must satisfy for the hypothesis to be correct. Here is how the algorithm is organized:
Figure 5.4 shows three main components:

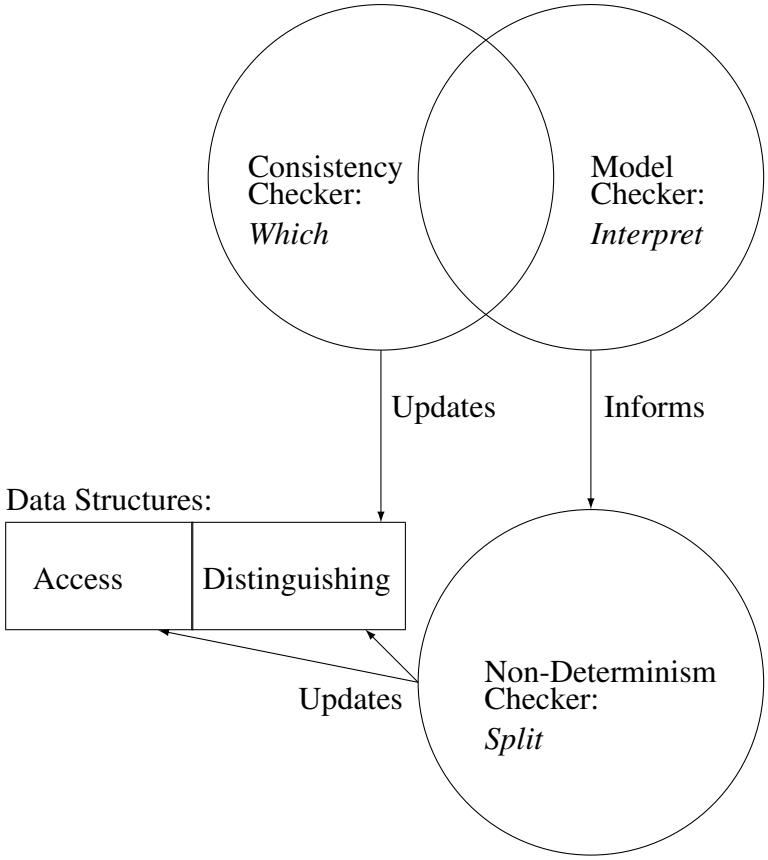


Figure 5.4: Operation of Algorithm

Model Checker:

The model checking component is realized in the Subroutine **Interpret**. Here we compare the counter-example to the hypothesis, checking if the hypothesis satisfies the counter-example. From the definition of 4.2e and 4.2f it follows that model checking is a recursive procedure where we follow transitions to new states, paring off the outermost [ ] or $\langle\ \rangle$ operator in the counter-example, then model-check the derived sub-formula at that new state.

Upon completion **Interpret** returns one of three things:

1) *NO-ERROR*

2) *SIMPLE-ERROR(*$\langle \alpha^*_{pt} \rangle$*, pt,* α*,* β*)*

3) *COMPLEX-ERROR(*$\langle \alpha^*_{pt} \rangle$*, pt, EP)*

SIMPLE-ERRORS are errors where the actions the counter-example prescribes for the current state are consistent with the current actions of the state. *COMPLEX-ERRORS* are those errors where the current state disagrees with the counter-example.

In the case of *SIMPLE-ERRORS*, α represents a known transition to the dead state in the hypothesis, whereas β represents an action that follows α in the target. This means we are learning a new state. If $\beta = \emptyset$ then we are learning a new α-transition. For *COMPLEX-ERRORS* the end-piece or EP is the disagreement that distinguishes mirrored paths of states to be split. Note that if we pass **Interpret** a counter-example given by the equivalence-oracle, **Interpret** should never return *NO-ERROR*, by definition of a counter-example.

Consistency Checker:

The consistency checking component works concurrently with the model checker to ensure that the behaviour we are being told to add to the hypothesis is being added at a place in the hypothesis whose path leading there makes similar non-deterministic choices consistent with those currently known of the target. This consists of the subroutine **Which**. **Which** decides *which* path through the LTS to take when confronted with a non-deterministic choice. It maps states of the hypothesis to sets of states of the target consistent with known distinguishing behaviour.

Non-determinism Checker:

This function is handled solely by the **Split** subroutine. Given a trace to a complex error found by **Interpret**, **Split** decides where along the path the non-deterministic mirroring began. It returns nothing, but does update the access traces and the distinguishing formula.

Finally Section 5.7 contains a worked example of the algorithm; one may want to glance over it now before continuing on reading.

5.5.1 Main Routine of Algorithm

The main section of the algorithm initializes an empty LTS. We then ask the Equivalence-oracle for a counter-example. If one exists we enter a loop which will run until no counter-example is found. Once the loop terminates we return our hypothesis as the answer. Inside the loop we run **Interpret** on the counter-example to find the disagreement; all disagreements will manifest themselves as a single disagreeing transition —that an action should or should not be happening. However, these single disagreements can compound; in actuality we catch some errors much earlier, particularly if the **Which**-subroutine cannot find a suitable path to take. These are single disagreements in the sense that $[\alpha]$ is a single modal operator. From here we try to classify this single misplaced transition as a simple or complex error; this is handled by **Interpret**. Simple errors require only local changes; complex errors require non-local changes. Recall each simple error leads to a new state or the dead state —there are no cycles. Since each iteration of the loop adds one transition, and possibly a new state, and since, as we shall see, the behaviours modeled in the hypothesis are a subset of those in the target, the loop cannot run forever, and the algorithm will eventually terminate.

Algorithm 5.5.1. Learn_LTS:

1 —initialize hyp as a single state

2 —Compute Equivalent(hyp)

3 —If equivalent: terminate; else counter example must be of the form $\langle\alpha\rangle\phi$

4 —let hyp be the LTS that has a start state $st \xrightarrow{\alpha} d$, a dead state.

5 —while(not Equivalent(hyp))

6 —let cce be the counter-example to equivalence

7 —let result = Interpret($\langle\rangle$, st, cce)

8 —Update(result)

9 —return hyp

5.5.2 Update

This subroutine updates the hypothesis once the type of error has been ascertained. If we are adding transitions at a state with pre-existing transitions, we add the transition to the dead state as a placeholder for the new state. If we find that the dead state should do an action, we place that action between the transition that originally led us to the dead state, and the dead state itself. We do this because many transitions from other states could converge at the dead state, and it might be that case that none of these other transitions admit this new action.

Subroutine 5.5.2. Update(*result*)

1 —IF result = *NO-ERROR*

2 —this is not possible

3 —IF result = *SIMPLE-ERROR(*$\langle\alpha^*_{pt}\rangle$*, pt,* α*,* β*)*

4 —If $\beta = \emptyset$ then: \\ *sending a transition to the dead state*

5 —In hyp add an α-transition from *pt* to the dead state.

6 —Let $(\delta_{pt})^{new} = (\delta_{pt})^{old} \wedge \langle\alpha\rangle tt$

7 —Otherwise: \\ *finding a transition at the dead state*

8 —Create a new state *st* with distinguishing formula $\delta_{st} = \langle\beta\rangle tt$

9 —In hyp redirect the α-transition starting at *pt* and ending at the dead state to a transition ending at *st*. The trace of *st* is now $\langle\alpha^*_{st}\rangle = \langle\alpha^*_{pt}\rangle\langle\alpha\rangle$

10 —If the redirected transition has a tentative distinguishing formula δ, remove it and let $\delta^{new}_{st} = \delta^{old}_{st} \wedge \delta$.

11 —add to *st* a β-transition to the dead state

12 —Find the $\Delta = \langle\alpha\rangle tt$ sub-formula or the $\Delta = \langle\alpha\rangle(EP)$ sub-formula of δ_{pt}, representing the α-transition that leads to the dead state.

13 —replace $(\delta_{pt})^{old} = \delta_j \wedge \Delta$ with $(\delta_{pt})^{new} = \delta_j \wedge \langle\alpha\rangle(\delta_{st})$ or with $\delta_j \wedge \langle\alpha\rangle(EP \wedge \delta_{st})$ respectively

14 —IF result = *COMPLEX-ERROR(*$\langle\alpha^*_{pt}\rangle$*, pt, EP)*

15 —use details to call Split($\langle \alpha^*_{pt} \rangle$,$pt$, EP)

5.5.3 Interpret

This next subroutine compares the counter-example to the hypothesis. Once it finds a disagreement it determines its nature: simple or complex. Recall we want to make the *smallest possible adjustment* to the hypothesis that results in it conforming to the counter-example. To test for a *SIMPLE-ERROR* we do a membership query to see if there exists a state in the target that has the same characteristics (same trace, same distinguishing formula) as the state we are presently examining *plus* the characteristics of the missing action. If such a state exists, we can safely add the transition to that state. If no such state exists, or the nature of the disagreement is such that we are being told we cannot do an action at a state which admits that action, correcting the error requires non-local change. Essentially this is because the distinguishing formula of the current state and the newly learned action cannot be reconciled –possibly because they directly conflict with each other. This suggests the existence of a non-deterministically mirrored state distinguished from the present state by the elucidated action.

These complex errors we describe can be more complicated still. Consider a case where the [ ] modal operator is the main connective of the associated sub-formula, $[\alpha]\delta$ for example, of the cce for some state. It could be the case that some set of this state's descendant states do not exhibit their cce in a simple way; yet, other mirrored states do not exhibit their cce in a complex way. We cannot consider each of these errors in isolation

because the [ ] modal operator enforces that δ must hold in *all* descendent states, including the two displaying the errors. Thus we must further distinguish the case we previously had consider, where the complex error at a descendant state is suggesting that this descendant state is being non-deterministically mirrored from a yet to be found mirroring descendant state, from the new case where the common ancestor and many of the descendant states are all being non-deterministically mirrored from an unfound branch at an even earlier ancestor state.

Finally, because of the tree like nature of HML, some of the sub-formulæ of the counter-example could be in complete agreement with the hypothesis. That is, some of the sub-formula will fail to elucidate any error. In these cases **Interpret** returns *NO-ERROR*.

Subroutine 5.5.3. Interpret($\langle\alpha^*_{pt}\rangle$, *pt*, cce):

1 □ if cce = $\langle\alpha\rangle tt$

2 —CASE I: No α-transition

3 —no other transitions?

4 —where st was the parent state, and β the label of the transition leading to this current dead state:

5 —return *SIMPLE-ERROR*($\langle\alpha^*_{st}\rangle$, *st*, β, α)

6 —other non α-transitions?

7 —test: membership($\langle\langle\alpha^*_{pt}\rangle\rangle(\langle\alpha\rangle tt \wedge \delta_{pt})$)

8 —If 'YES'

9 —return *SIMPLE-ERROR*($\langle\alpha^*_{pt}\rangle$, *pt*, α, $\emptyset$)

—Otherwise:

—Let $(\delta_{pt})^{new} = (\delta_{pt})^{old} \wedge [\alpha]ff$

—return *COMPLEX-ERROR(*$\langle\alpha^*_{pt}\rangle$*, pt,* $\langle\alpha\rangle$*)* \\ **TYPE I**

—CASE II: One or more α-transitions

—return *NO-ERROR*

□ if cce = $\langle\alpha\rangle(\bigwedge_{i=1}^{m}\varphi_i)$

—CASE I:no α-transition

—no other transitions

—where st was the parent state, and β the label of the transition leading to this current dead state:

—return *SIMPLE-ERROR(*$\langle\alpha^*_{st}\rangle$*, st,* β*,* α *)*

—other non α-transitions

—test: membership($\langle\langle\alpha^*_{pt}\rangle\rangle(\langle\alpha\rangle tt \wedge \delta_{pt})$)

—If 'YES'

—return *SIMPLE-ERROR(*$\langle\alpha^*_{pt}\rangle$*, pt,* α*,* $\emptyset$ *)*

—Otherwise:

—Let $(\delta_{pt})^{new} = (\delta_{pt})^{old} \wedge [\alpha]ff$

—return *COMPLEX-ERROR(*$\langle\alpha^*_{pt}\rangle$*, pt,* $\langle\alpha\rangle$*)*\\ **TYPE I**

—CASE II:one α-transition

—to the dead state?

—Pick a $\langle\beta\rangle$ operator from among the φ_i's that start with '$\langle\ \rangle$'-operators

—If no such $\langle\beta\rangle$ exists then return *NO-ERROR*

—return *SIMPLE-ERROR(*$\langle\alpha^*_{pt}\rangle$*, pt,* α*,* β*)*

—to a regular state?

—Denoted this next state *st*

—For each φ_i compute following:

—result = Interpret($\langle\alpha^*_{pt}\rangle\langle\alpha\rangle$, *st*, φ_i)

—return first result not equal to *NO-ERROR*

—Otherwise return *NO-ERROR*

—CASE III:several α-transitions

—call **Which(**α**,** ***pt*****,** $\langle\langle\alpha^*_{pt}$**,** cce**)** $\backslash\backslash$ contains **TYPE II** error

□ if cce = $[\alpha]$*ff*

—CASE I:no α-transition

—Not a disagreement (vacuously satisfied)

—CASE II:any α-transitions

—return *COMPLEX-ERROR(*$\langle\alpha^*_{pt}\rangle$*, pt,* $[a]$*ff)*$\backslash\backslash$ **TYPE III**

□ if cce = $[\alpha](\bigvee_{i=1}^{m}\varphi_i)$

—let *pt* be current state

—CASE I:no α-transition

—Not a disagreement (vacuously satisfied)

—CASE II:several α-transitions to states st_j

—For each α-transition to a state st_j

—Use **OR-elimination**($\langle\langle\alpha^*_{st_j}\rangle\rangle$, st_j, $\bigvee_{i=1}^{m}\varphi_i$) to get $\bigvee_{i\in T\subset\{1\ldots m\}}\varphi_i$

—If $T=\emptyset$

53 —let $(\delta_{pt})^{new} = (\delta_{pt})^{old} \wedge \langle\alpha\rangle(\bigwedge_{i=1}^{m} \neg\varphi_i)$

54 —return *COMPLEX-ERROR(*$\langle\alpha^*_{pt}\rangle$*, pt,* $[\alpha](\neg\delta_{pt})$*)* **TYPEIV**

55 —For each $\varphi_i, i \in T$ compute the following:

56 —result = Interpret($\langle\langle\alpha^*_{st_j}\rangle\rangle$, st_j, φ_i)

57 —If any result returns *NO-ERROR*

58 —go to the next state st_j

59 —If all results return *COMPLEX-ERROR(*$\langle\alpha^*_{st}\rangle$*, st,* EP_i*)*

60 —let $(\delta_{pt})^{new} = (\delta_{pt})^{old} \wedge \langle\alpha\rangle(\bigwedge_{i=1}^{m} \neg\varphi_i)$

61 —return *COMPLEX-ERROR(*$\langle\alpha^*_{pt}\rangle$*, pt,* $[\alpha](\bigvee_{i=1}^{m} \varphi_i)$*)* **TYPEIV**

62 —If all states had *NO-ERROR* as a result at least once

63 —return *NO-ERROR*

64 —Else return *any* SIMPLE-ERROR or COMPLEX-ERROR found for any state.

Note: for the case cce = $[\alpha](\bigvee_{i=1}^{m} \varphi_i)$, we need to be careful about verifying whether the descendant states of the current state satisfy the φ_i's. Our concern is that we could find a *COMPLEX-ERROR* down one of the α transitions, but **Split** suggests we place the start of the new branch before the current state. This kind of correction would not fix the fact that on all α-transitions, one of the φ_i needs to be satisfied. It has placed the new satisfying state elsewhere. This is not a problem arising from the creation of a wrong hypothesis. The problem is that the distinguishing formula would not be maintained correctly. However, for *COMPLEX-ERROR*s found downstream of the $[\alpha]$ operator, we do not need to worry

about the split being placed before the current state, except in the case $T = \emptyset$. There is no worry because **Or-Elimination** reduced the φ_i's only to those we know each state st_j could accept. Thus any *COMPLEX-ERROR* found downstream can be fixed in place, at the state at which it was found. If $T = \emptyset$, we need to mirror the entire current state; but the distinguishing formula is noted properly. Note that the **Which**-subroutine would normally catch this problem before we ever found $T = \emptyset$, the only exception being if the current hypothesis is still deterministic, we would not place a call to **Which**.

5.5.4 Split subroutine

The **Split**-subroutine is the key to the algorithm. Given a disagreement and a trace to that disagreement, **Split** determines where along the path from the start state to the disagreement the non-deterministic mirroring begins.

Subroutine 5.5.4. Split($\langle\alpha^*\rangle$,pt, EP):

1 —Let $\delta_{pt(i)}$ denote the distinguishing formula of the i^{th} prefix of $\langle\alpha^*_{pt}\rangle = \alpha_1 \ldots \alpha_n$, the trace to state pt.

2 —For each prefix and suffix of $\langle\alpha^*_{pt}\rangle$ from $i = 1 \ldots n-1$, test:

3 —membership$\big(\langle\!\langle\alpha^*_{pt}\rangle\!\rangle|_i \left(\delta_{pt(i)} \wedge \left(|\langle\!\langle\alpha^*_{pt}\rangle\!\rangle)_i \delta_{pt(i)} \wedge |\langle\!\langle\alpha^*_{pt}\rangle\!\rangle)_i EP\right)\right)$

4 —until first i such that above test fails

5 —Create $n - i$ new duplicate states $(\langle\alpha^*_{pt}\rangle|'_i \ldots (\langle\alpha^*_{pt}\rangle|'_{n-1}$ with distinguishing formulæ $|\langle\alpha^*_{pt}\rangle)_i$(EP) $\ldots$ $|\langle\alpha^*_{pt}\rangle)_{n-1}$(EP), respectively, in a manner similar to updating a simple error.

6 —Add an α_i-transition from $(\langle\alpha^*_{pt}\rangle|_{i-1}$ to the new state $(\langle\alpha^*_{pt}\rangle|'_i$

7 —Change distinguishing formula δ of $(\langle\alpha^*_{pt}\rangle|_{i-1}$ to $\delta \wedge \langle\alpha_i\rangle|\langle\alpha^*_{pt}\rangle)_i$(EP).

8 —Connect $(\langle\alpha^*_{pt}\rangle|'_i \ldots (\langle\alpha^*_{pt}\rangle|'_{n-1}$ with the same transitions as states $(\langle\alpha^*_{pt}\rangle|_i \ldots (\langle\alpha^*_{pt}\rangle|_{n-1}$

9 —If EP begins with modal operator $\langle\beta\rangle$

10 —create a new state $(\langle\alpha^*_{pt}\rangle|_n$ with distinguishing formula EP

11 —Send a β-transition from this new state to the dead state.

12 —Otherwise send a $\langle\alpha_n\rangle$-transition from state $(\langle\alpha^*_{pt}\rangle|_{n-1}$ to the dead state. Assign this transition a tentative distinguishing-formula of EP.

When we create the path of mirroring states for **Split** (line 5), it is easiest to think of the operation as a copy and paste of the mirrored path. Later, when we prove correctness of the algorithm we will assume the distinguishing formulæ of both mirroring branches conform to a certain template where the only difference in these templates between mirrored paths is the addition of the EP distinguishing formula; think of the mirroring path as being created in the exact same manner as the original path —keeping the remainder part of the templates is exactly the same.

5.5.5 Which subroutine

When confronted with a non-deterministic choice, the **Which**-subroutine decides *which* path to follow —determined by whether actions known to occur down that path coincide with actions the counter-example illustrates.

Subroutine 5.5.5. Which(α, pt, cce, $\langle\alpha^*\rangle$):

1 —Let φ be the sub-formula of cce $= \langle\alpha\rangle\varphi = \langle\alpha\rangle(\bigwedge_i \phi_i)$

2 —For each α-transition at current state pt to next state st with distinguishing formula δ_{st} test:

3 —membership$\big(\langle\langle\alpha^*_{pt}\rangle\rangle\,(\delta_{pt} \wedge \langle\alpha\rangle\,(\delta_{st} \wedge \varphi))\big)$

4 —If no consistent α-transitions:

5 —Must test if we can add the α-transition here:

6 —membership$\big(\langle\langle\alpha^*_{pt}\rangle\rangle\,(\delta_{pt} \wedge \langle\alpha\rangle\varphi)\big)$

7 —If Yes

8 —return *SIMPLE-ERROR(*$\langle\alpha^*_{pt}\rangle$, pt, $\alpha\{\varphi\}$, $\emptyset$ *)*

9 —Otherwise:

10 —Let $(\delta_{pt})^{new} = (\delta_{pt})^{old} \wedge [\alpha]\neg(\varphi)$

11 —return *COMPLEX-ERROR(*$\langle\alpha^*_{pt}\rangle$, *pt,* $\langle\alpha\rangle\varphi$*)* \\ **TYPE II**

12 —If one consistent α-transition

13 —then treat this as if this one consistent transition were the only α-transition

14 —If many consistent α-transitions

15 —Pick any α-transition to proceed along

An important note to make about **Which** concerns the last case "*many consistent* α*-transitions*": because we work to maintain the correctly-distinguishing property (see sec-

tion 5.2), we do not need to worry about the possibility that the distinguishing formulæ for the α-transitions are not mutually exclusive (see figure 5.1) except that this new behaviour might be correct down both transitions. However this does not mean we want to recurse down both paths. For example, look at Figure 5.3. If we consider the lefthand LTS to be the hypothesis and the righthand the target: we see that if the counter-example is $\langle b\rangle(\langle ab\rangle tt \wedge \langle aa\rangle tt)$ then the actions $\langle ab\rangle tt \wedge \langle aa\rangle tt$ coincide with both b-transitions in the target. Furthermore, recursing down either will lead to a correct choice in updating the hypothesis. Note, however, if we were to make changes down both paths, our hypothesis would contain states not in the target LTS.

Another important point to note is that in the case of '*no consistent transitions*'. The point where we add the new α transition (line 8) is actually a special case of *COMPLEX-ERROR*; it would be the same as if the **Split**-subroutine tested all prefixes up to n instead of $n-1$. The only reason we treat it as a special case is to simplify the proof. We want it so that **Split**'s membership query for the n^{th} prefix returns false. It is the same reason for the test on line 21 of **Interpret**. It has to do with the meanings of $\langle\ \rangle$-operators and [ ]-operators —we can 'fix' a state with a $\langle\ \rangle$-operator by adding a new transition, yet this would not work for a [ ]-operator.

5.5.6 Dealing with 'OR'

This next subroutine decides which sub-formulæ of $\bigvee \varphi_i$ coincide with the given state.

Subroutine 5.5.6. OR-Elimination($\langle\alpha^*_{pt}\rangle$, pt, ($\bigvee_{i=1}^{m} \varphi_i$)):

1 —Let δ_{pt} be the distinguishing formula for state pt

2 —For each φ_i test:

3 —membership($\langle\langle \alpha_{pt}^{*} \rangle\rangle(\delta_{pt} \wedge \varphi_i)$)

4 —Let $T \subset \{1 \ldots m\}$, such that $j \in T \leftrightarrow$ for φ_j the above test returned true

5 —return $\bigvee_{i \in T} \varphi_i$

5.5.7 Summary of Split Types

We consider Types I- IV:

TYPE:

I - Current state has no α-transition

- Current counter-example (cce) says that there is a state with this trace that can do an α-transition
- no state can do an α-transition after said total trace and the currently known distinguishing formula

Remedy:
- add to the existing state's distinguishing formula the fact that it cannot do an α-transition
- create a new state that can do an α transition
- Use **Split** to locate the branching point
- **NOTE:**The two mirroring states are now distinguished by $[\alpha]ff$ and $\langle\alpha\rangle tt$

II - None of the α-transitions leaving this state led to states with distinguishing formula that agreed with the behaviour elucidated in the counter-example

- Adding a new α-transition at the current state, eventually leading to a state that agreeing with the counter-example is not possible

Remedy:
- We now know the current state satisfies $[\alpha](\neg\varphi)$ where φ is the behaviour in the current counter-examples
- we know there is a new state that satisfies $\langle\alpha\rangle\varphi$
- use **Split** to determine branch point for new state
- **NOTE:**The two mirroring states are now distinguished by $[\alpha](\neg\varphi)$ and $\langle\alpha\rangle\varphi$

III
- The current state has an α-transition
- the counter-example says that there is a state with the same trace that does not permit an α transition

Remedy:
- create a new mirroring state with $\neg(\langle\alpha\rangle tt)$ as its distinguishing formula
- use **Split** to find branching point
- **NOTE:** The two mirroring states are now distinguished by $\neg(\langle\alpha\rangle tt)$ and $\langle\alpha\rangle tt$

IV
- On all α-transitions from this state any of the φ_i's must hold
- None of the states that *any* of the α-transitions lead to can satisfy *any* of the φ_i actions
- Or, additionally, all the φ_i's would induce a *COMPLEX-ERROR* if taken as corrections

Remedy:
- This current state has at least one α-transition that does not model any of the φ_i actions
- There must be a new state that can do at least one φ_i action on all α-transitions
- **NOTE:**The two mirroring states are now distinguished by $\langle\alpha\rangle(\bigwedge \neg\varphi)$ and $[\alpha](\bigvee \varphi)$

Proposition 5.5.7. *None of the known split types can be resolved without creating a non-deterministically mirroring branch.*

Proof. We simply note that all the known split types represent adding behaviour to a state that would cause the state to satisfy ff. □

Lemma 5.5.8. *Irrespective of whether branches are placed correctly by **Split**, after a single split, the Correctly-Distinguishing Property is maintained.*

Proof. By inspection of the new states' distinguishing formulæ. □

5.6 Proof of Correctness

We are going to start by assuming that the target LTS is deterministic. This will simplify the algorithm's analysis, we will need only to show that those parts dealing with simple errors are correct. We can ignore any *COMPLEX-ERROR*, the **Which**-subroutine, and **Split**. We can also delay looking at **OR-Elimination**; even though we may invoke it on a deterministic target, we will see it is not necessary and thus we delay proving its correctness.

5.6.1 Deterministic Target

Our first goal is to prove that if the target LTS is deterministic then **Interpret** would only return *NO-ERROR* or *SIMPLE-ERROR*. To this end, we prove that the current hypothesis will always be deterministic if the target is deterministic. This will allow us to eliminate counter-examples containing the modal operator [ ]. By considering the [ ]-operator to be a $\langle\ \rangle$-operator, we are not losing any meaning (that is, if the target satisfies $\langle\alpha\rangle\varphi$ then the target also satisfies $[\alpha]\varphi$ because of determinism). However, we could be adding new meaning where it was not intended; this is because a formula of the form $[\alpha]\varphi$ can be vacuously satisfied by states without α-transitions. Clearly changing $[\alpha]\varphi$ to $\langle\alpha\rangle\varphi$ in that case has a different meaning. What we are saying is our algorithm catches those cases, and in all others effectively views the [ ]-operator as a $\langle\ \rangle$-operator. In addition, assuming **Or-Elimination** works correctly, this includes cases where formula $[\alpha](\bigvee \varphi_i)$ become $\langle\alpha\rangle(\bigwedge_T \varphi_i)$.

Let us look at some simple things we can say about using our algorithm on deterministic targets.

Lemma 5.6.1. *If the current hypothesis is deterministic our algorithm will never call the '**Which**'-subroutine.*

Proof. By definition of deterministic, none of the states will have multiple transitions with the same label. □

Corollary 5.6.2. *If the current hypothesis is deterministic then:*

membership($\langle\alpha^*\rangle\delta$), and

membership($\langle\langle\alpha^*\rangle\rangle\delta$)

are always equal.

Lemma 5.6.3. *If the current hypothesis is deterministic then we have yet to encounter a complex error.*

Proof. *COMPLEX-ERROR*s call **Split**, **Split** introduces non-determinism; the lemma states the contrapositive. □

We want to extend this previous result to make the stronger statement that when we use our algorithm to learn a deterministic targets we will never encounter complex errors. This allows us to conclude that complex errors are the result of non-determinism and not an error in the algorithm's ability to learn determinism. We are envisioning the algorithm as having two halves: one half learns determinism and returns simple-errors, the other half learns non-determinism and returns complex-errors.

Lemma 5.6.4. *If the current hypothesis is deterministic then, if* $\langle \alpha^*_{pt} \rangle$ *is a trace to a state* pt *in the hypothesis, and* δ_{pt} *its distinguishing formula, for any state* pt*:*

$$\begin{aligned} &\text{IF the hypothesis} \models \langle \alpha^*_{pt} \rangle (\delta_{pt}) \\ &\text{THEN the target} \models \langle \alpha^*_{pt} \rangle (\delta_{pt}) \end{aligned} \tag{5.10}$$

Proof. By lemma 5.6.3 and by examining the procedure for correcting *SIMPLE-ERROR*s in the subroutine Update 5.5.2 we surmise that the lesser statement:

$$\begin{aligned} &\text{if the hypothesis} \models \langle \alpha^*_{pt} \rangle tt \\ &\text{then the target} \models \langle \alpha^*_{pt} \rangle tt \end{aligned} \tag{5.11}$$

is true. If all the errors so far have been simple, then the formation of the distinguishing formulæ δ conform to this simple grammar:

$$\delta := \langle \alpha \rangle tt \mid \langle \beta \rangle (\bigwedge \delta_i) \tag{5.12}$$

where $\alpha, \beta \in \Sigma$ and where $(\bigwedge \delta_i)$ is the conjunction of finitely many δ_i's including $\bigwedge_{i=0}^{0} \delta_i = \delta_0$.

Therefore:

$$\begin{aligned} \text{If hypothesis } &\models \langle \alpha^*_{pt} \rangle (\bigwedge \delta_i), \\ \text{then } \forall i, \text{ the hypothesis } &\models \langle \alpha^*_{pt} \rangle \delta_i \end{aligned} \tag{5.13}$$

Using the above argument inductively we see, each δ_i is of the form:

$$\langle \alpha^*_i \rangle \left(\bigwedge \delta'_i \right)$$

for some equation δ'_i.

Therefore:

$$\text{If hypothesis } \models \langle \alpha^*_{pt} \rangle \left(\bigwedge \delta_i \right) \tag{5.14}$$

by Equation 5.11 we can conclude:

$$\forall i \text{ target } \models \langle \alpha^*_{pt} \rangle \langle \alpha^*_i \rangle tt \tag{5.15}$$

Now, because the target is deterministic we can conclude that:

$$\text{target } \models \langle \alpha^*_{pt} \rangle \left(\bigwedge \langle \alpha^*_i \rangle tt \right) \tag{5.16}$$

Since '$\bigwedge$' is never the inner most connective, continuing the above argument will allow us to reach the conclusion that:

$$\begin{aligned} \text{if the hypothesis} &\models \langle \alpha^*_{pt} \rangle \delta_{pt} \\ \text{then the target} &\models \langle \alpha^*_{pt} \rangle \delta_{pt} \end{aligned} \tag{5.17}$$

□

Lemma 5.6.5. *Algorithm 5.5.1 will never encounter* COMPLEX-ERROR*s if given a deterministic target.*

Proof. Since the target is deterministic each state has a unique trace, thus if a counter-example says of the target that:

$$\begin{gathered} \text{target} \models \langle \alpha^*_{pt} \rangle \varphi \quad \text{and} \quad \text{target} \models \langle \alpha^*_{pt} \rangle \psi \\ \text{then we conclude that :} \quad \text{target} \models \langle \alpha^*_{pt} \rangle \left(\varphi \bigwedge \psi \right) \end{gathered} \tag{5.18}$$

If you look at the four instances where **Split** is called (section 5.5.7), we see that the sub-formula of the counter-example, which we will call φ_{ce}, which eventually forms the variable EP, say φ_{ce}, and a sub-formula of the distinguishing formula of the current state, say ψ_{df}, are irreconcilable, in that $\varphi_{ce} = \neg\psi_{df}$.

If you look at the simple errors, a hypothesis built from only simple errors can only be deterministic, regardless of the target. Furthermore, such a hypothesis satisfies the property of Lemma 5.6.4. Assume that we have a hypothesis that has been built only using simple errors. This is possible since the first error is always simple. Now counter-examples express only actions represented in the target, thus the target satisfies $\langle \alpha^*_{pt} \rangle (\varphi_{ce})$. Because of Lemma 5.6.4, the target also satisfies $\langle \alpha^*_{pt} \rangle (\psi_{df})$ because the hypothesis satisfies it. Therefore, because the target is deterministic we can conclude that:

$$\begin{aligned} \text{target} &\models \langle \alpha^*_{st} \rangle (\varphi_{ce} \wedge \psi_{df}) \\ &\models \langle \alpha^*_{st} \rangle (\neg\psi_{df} \wedge \psi_{df}) \\ &\models \langle \alpha^*_{st} \rangle (\mathit{ff}) \\ &\models \mathit{ff} \end{aligned} \tag{5.19}$$

Thus we have shown for deterministic targets:

> Encountering a complex-error implies there exists a
> state *st* such that, $st \models \mathit{ff}$

Since no state satisfies *ff*, we conclude that for all states we only encounter non complex-errors. □

Corollary 5.6.6. *If given a deterministic target all of our intermediary hypothesis will be deterministic.*

Note that we have achieved our goal, proving that for deterministic targets we will not encounter complex errors. This ultimately came down to verifying that all complex errors (Section 5.5.7) arise from irreconcilable differences. The rest followed directly. Alternatively we have not shown that if the target has non-determinism we will encounter complex-errors. Why? —because we are not yet sure if all the listed complex-errors exhaustively include all the manners in which a counter-example could express non-determinism. We can only be sure however that if we find a complex-error the target is non-deterministic —the contrapositive of Lemma 5.6.4.

We now recall the idea of learning as the reverse of the process of taking effective sub-LTSs —recall Definition 5.4.1. Our main goal is to use this construct to help us visualize the final proof.

Lemma 5.6.7. *Every hypothesis at each stage of the learning algorithm is a sub-tree-LTS of the target.*

Proof. If we get a counter-example formula of the type $[\alpha]\varphi$ for some current state *pt*, then either it is not an error or it is of the form $[\alpha](\bigvee \varphi_i)$. In that case our algorithm uses **OR-elimination** to get, for a fixed α-transition leading to a state *st*, a new counter-example: $\bigvee_{i\in T} \varphi_i$, where each φ_i is satisfied by that fixed state. By determinism and Corollary 5.6.6, this is equivalent to a counter-example of the form $\langle\alpha\rangle(\bigwedge_{i\in T} \varphi_i)$ for state *pt*.

Moreover, because of determinism we can consider all [ ]-modal operators as $\langle\rangle$-operators, except those that appear as $[-]\mathit{ff}$ –which, however, would not be counter-examples in a deterministic LTS. Thus, effectively all counter-examples are ultimately of the form $\langle\alpha\rangle(\bigwedge \varphi_i)$. These then form the only *necessary* counter-examples for deterministic LTS. If you read Algorithm 5.5.1 carefully you will notice that the algorithm is trying to convert (when possible) all counter-examples into the form $\langle\alpha\rangle(\bigwedge_{i\in T} \varphi_i)$. That is, we simplify errors to traces to a single disagreement; *SIMPLE-ERROR*s themselves are just missing traces. Correspondingly, let us only consider such counter-examples (*effective* counter-examples).

First note that every formula of the form $\langle\alpha\rangle(\bigwedge \varphi_i)$ has a tree like structure (Figure 5.5).

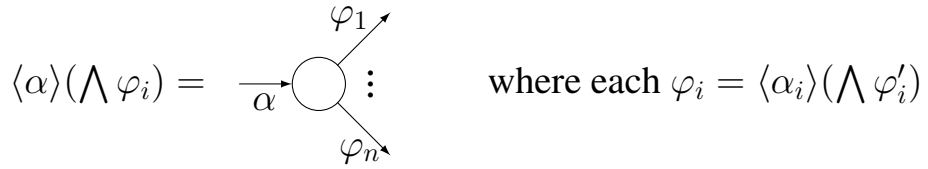
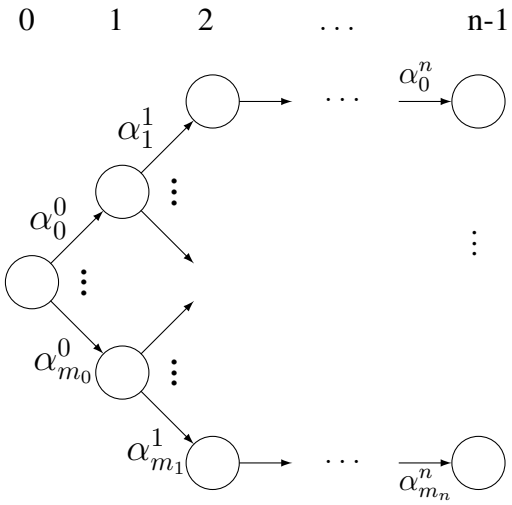


Figure 5.5: Tree-like structure of formulæ

Thus we can transform every tree into a formula in the manner suggested by Figure 5.6,

As long as we use these notational conventions, the set of Γ formulæ described in Figure 5.6 are in a one-to-one correspondence with the set of minimal deterministic tree-LTS. To compute for any LTS its associated Γ-formula, we use the following recursive



$$\Rightarrow \exists\Gamma \;=\; \left(\bigwedge_{i\in I=\{0\ldots m_0\}}\langle\alpha_i^0\rangle\left(\bigwedge_{j\in J}\langle\alpha_j^1\rangle\left(\ldots\left(\bigwedge_{k\in K=\{0\ldots m_n\}}\langle\alpha_k^n\rangle\right)\ldots\right)\right)\right)$$

Figure 5.6: Turning a generic tree into a HML formula

procedure, where we denote the start state *st*:

$$\begin{aligned}\Gamma(\mathrm{st}) &:= \bigwedge_{\forall st\xrightarrow{\alpha}pt}\langle\alpha\rangle\Gamma(pt)\\ \text{where } \bigwedge_{\emptyset}\langle\alpha\rangle\Gamma(pt) &:= tt\end{aligned} \tag{5.20}$$

The key to the one-to-one correspondence is the fact that each character α appears once in the formula $\left(\bigwedge_{\alpha\in\Sigma}\langle\cdot\rangle\Gamma(pt)\right)$. Since targets are deterministic each one refers to a unique branch. Thus the modal operator $\langle\ \rangle$ serves as a list of which transitions leave the current state; and, $\Gamma(pt)$ serves as a description of the sub-LTS that starts from this state. This is analogous to how instead of assuming each new-counter-example is a new branch, our algorithm compares each new counter-example to the existing hypothesis LTS, and tries to find a minimal difference.

Actually, more then analogous: it is *this* conversion from formula to tree that our

algorithm is attempting. We achieve this because the '*effective counter-examples*' in our algorithm contain only the $\bigwedge$-connective; like Γ formulæ. Since every sub-tree of the target is itself a tree-LTS they, as well, have corresponding Γ formula. These formula will be subformula of Γ. For instance if

$$\Gamma = \left(\bigwedge_{i \in I = \{0 \ldots m_0\}} \langle \alpha_i^0 \rangle \left(\bigwedge_{j \in J} \langle \alpha_j^1 \rangle \left(\ldots \left(\bigwedge_{k \in K = \{0 \ldots m_n\}} \langle \alpha_k^n \rangle \right) \ldots \right) \right) \right)$$

then for any effective subtree of the Target there exists $I' \subset I$, $J' \subset J$, $\ldots$ $K' \subset K$ such that

$$\Gamma' = \left(\bigwedge_{i \in I'} \langle \alpha_i^0 \rangle \left(\bigwedge_{j \in J'} \langle \alpha_j^1 \rangle \left(\ldots \left(\bigwedge_{k \in K'} \langle \alpha_k^n \rangle \right) \ldots \right) \right) \right)$$

This is because effective subtrees only include states reachable from the start state, and that removing a state along a branch removes every successor state. If we remove the fixed state with the incoming transition labeled α_i^k we would remove the whole sub-formula:

$$\langle \alpha_i^k \rangle \left(\bigwedge_{j \in J'} \langle \alpha_j^{k+1} \rangle (\cdots) \right)$$

Every counter-example given to our algorithm will be converted to one of these sub-formulæ of the target's Γ formula. We can prove this fact by induction:

Base Case: Our first hypothesis is the empty LTS, which corresponds to the empty HML formula, a subformula of Γ. If in fact the target is non empty then, the first counter example will tell us the first transition. Our next hypothesis will be a single α-transition from the start state to a dead state. This corresponds to $\langle \alpha \rangle tt$. This is the associated Γ-formula of this hypothesis as the procedure of Equation 5.20 would build it. It is also a sub-formula of the target's Γ-formula.

Inductive Step: Assume that for the k^{th} hypothesis and its associated Γ-formula, call it Γ^k, are sub-LTSs and subformulæ respectively. We need to show this property holds for the $k+1^{st}$ hypothesis.

Since we know that the **Interpret** subroutine will only return *SIMPLE-ERROR*, or *NO-ERROR*, we only need to consider two cases —the two ways that *SIMPLE-ERROR* cause changes in the hypothesis. In both cases we compare the counter-example to the hypothesis. From the assumption in the inductive step, and the determinism of the hypothesis we know the error is in some fixed state of the hypothesis corresponding uniquely to some fixed state of the target. This fixed state is uniquely described by the trace to the simple error. Because the counter-example is illuminating actions from this state that the target admits, as long as can show we can add a single action correctly, we can prove the inductive hypothesis. Considering only those updates upon a simple error, the two cases are, one, when the next state is the dead state versus, two, when it is any other state:

Case One:

The current known state transitions on α to the dead state.

If we find ourselves in this position we surmise two possibilities. First, perhaps the counter-example is not illustrating an error down this branch. For instance the counter-example may be of the form $\langle\alpha\rangle tt$ or $\langle\alpha\rangle(\bigwedge_{\beta\in\Sigma}[\beta]ff)$ [1]. Otherwise, second, we have found a leaf of a sub-tree LTS that is not a leaf in the target. Again, this is because for the current counter-example to truly be a counter-example, it must disagree with the fact that α leads to a dead state. The only way that could happen is if the counter-example exhibited the fact that α leads to a state that can do some other β-action, which implies that the

[1] though as discussed, [ ]-operators would never be pertinent to the counter-example in our setup

state satisfies $[\alpha]\left(\bigvee_{\beta\in\Sigma}\langle\beta\rangle tt\right) \wedge \langle\alpha\rangle$. Therefore we know the current state transitions on α to a new found state, which in turn transitions on β to a leaf of the sub-tree-LTS. Here is how the algorithm compensates:

1) We create a new state $\langle\alpha^*_{st}\rangle = \langle\alpha^*_{pt}\rangle\langle\alpha\rangle$ where pt is the current state
2) The new state *st* is given the distinguishing formula $\delta_{st} = \langle\beta\rangle tt$
3) The old state must have had $\langle\alpha\rangle$ as a subformula in its distinguishing formula, as it would have been added in a similar manner to step '2)' or 'Case Two' below. We can replace that with $\langle\alpha\rangle\delta_{st}$

Case Two:

Adding a transition to any other state.

The other way we add to the hypothesis, is if we add a new transition to a previously known state. This new transition would necessarily lead to a leaf of the next sub-tree-LTS; we naturally send it to the dead state for now, knowing of no further behaviour. We modify the current states distinguishing formula to include a '$\wedge\langle\alpha\rangle tt$' clause –where α is the label of the new transition. No new states are added. This new distinguishing formula fits with the Γ-interpretation of distinguishing formula.

This completes both the induction and the proof, as the distinguishing formulæ for the hypotheses all have the form of subformula of Γ. Furthermore the corresponding hypothesis has been built using the same one-to-one correspondence described by equation 5.20 and is itself a sub-tree. □

Theorem 5.6.8 (Correctness of Algorithm for Deterministic Targets)**.** *Algorithm Learn-Tree-LTS correctly learns deterministic tree-LTSs in polynomial time.*

Proof. By Lemma 5.6.7 each hypothesis is a sub-tree of the target. Furthermore, each hypothesis adds either a single state or a single transition. Thus measured in the number of states plus the number of transitions the size of the hypotheses are strictly increasing. Since the target has finite size, we must eventually reach a hypothesis equal to the target.

If the number of states in the target is n, then the total number of states and transition is $2n - 1$. Thus, the number of counter-examples we need is $O(n)$. The time complexity of comparing each counter-example to the hypothesis is proportional to the time it takes to consider each state and transition of the hypothesis, which is on the order of $O(n)$. Therefore the overall time complexity is $O(n^2)$. □

In fact if we were to write an algorithm that only had to learn deterministic Tree-LTS we could take our algorithm and remove all the parts dealing with distinguishing formulæ or complex-errors.

5.6.2 Black Box Split

Our next goal is to prove the algorithm works for any Tree-LTS. This means we must consider the effect of non-determinism. To continue to simplify things we will make another assumption. This assumption is that if we find a non-deterministic error we can correctly update it. We use this to prove, given a working **Split** subroutine, that we maintain the Correctly Distinguishing Property. Once that is done, we only have to prove that **Split** works correctly the first time, while the hypothesis is still deterministic.

First note that the **Interpret** sub-routine has different cases for any possible combination of '*views*' of states and counter-examples. By *views of states* we refer to what all the current outgoing transitions look like to the state. Based on what the counter-example

says we should do and what the state actually does, we decide if the upgrade is complex. In such a case we create new paths, or branches, from some point in the LTS. Figure 5.7 shows the idealized split operation. On the left hand side we have a section in the current hypothesis representing a trace $\langle \omega^* \beta^* \rangle$ to a disagreement. In actuality, this trace is masking a branching path. However, until now we had not yet learned enough actions to distinguish these branches.

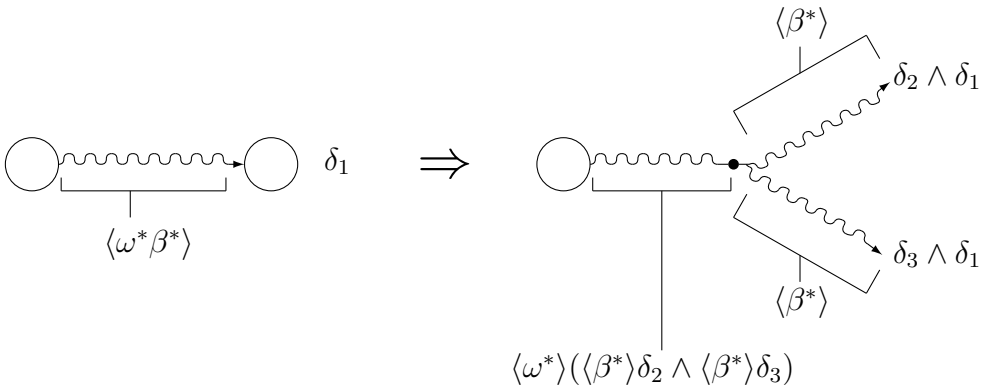


Figure 5.7: Conceptual Drawing of Split

For Figure 5.7, δ_1 represents the distinguishing formula of the state where the disagreement occurred. It encodes the actions that happen in the descendant states of the current state. Note that a *COMPLEX-ERROR* implies that a state has descendant states.

Proposition 5.6.9. *It is not possible that a complex error occurs at a state with no descendants.*

Proof. Assume for a moment that **Interpret** could enter a dead or dummy state, adding any action to a state with no transitions would be considered a simple error. Why? Consider the one instance where it might matter: we have a dead state non-deterministically mirroring another state.

Specifically, consider a target with a dead state (no transitions) and call it state pt. Let us further assume that the target has a state st with descendants and that st and pt have a common trace $\langle\alpha^*\rangle$. It could be the case that the hypothesis has only just learned the actions in that trace. In this case, the last state learned along that trace would have no transitions, accurately representing state pt. If the next counter-example begins to illuminate the behaviour of the descendants of state st, it would be correct to continue adding transitions along the trace $\langle\alpha^*\rangle$ –thus destroying our hypothesis's previously accurate representation of state pt. Had we known state pt had distinguishing formula $[-]\mathit{ff}$, it would also be correct to split. Either change would be correct because in the first choice we will get another counter-example to tell us in fact state pt exists. To have had split earlier we would have had to anticipate that perhaps no transitions should be added at the end of the trace $\langle\alpha^*\rangle$ —which we are not currently doing.

In summary, complex-errors do not need to happen at dead states, because we naturally assume all frontier transitions lead there. That is, the dead states act as placeholder states, as we have emphasized before. □

Coming back to Figure 5.7, we know that $\delta_1 \neq [-]\mathit{ff}$, that is it actually details real actions of the disagreeing state. This next point is key, and goes to the contrived nature of tree-LTS. Since the split states currently disagree momentarily on a single action, and presumably could still agree with most of the actions δ_1 entails, it would make sense to keep the paths split only up to the disagreement. Transitions thereafter should re-merge in the hypothesis because to the best of our knowledge the descendants of the last split state are still bisimilar. Were the target a regular LTS this is what we would do, and we do so in the case of Directed Acyclic target LTSs (Section 6.1.1). However, since we know the

target is a tree, we know split branches stay split –equivalently we know there must exist continued future disagreements.

What does this all mean? That only the behaviour of descendant states matter, though we could have already surmised this from the definition of bisimilarity. Assuming that we can compute formulæ that can correctly distinguish non-deterministically mirrored states we can *determinize* non-deterministic tree-LTS. By determinize we mean that we markup non-deterministically mirrored transitions with distinguishing formulæ. Figure 5.8 shows how to achieve this, where δ_i,δ_j Correctly Distinguish states i and j.

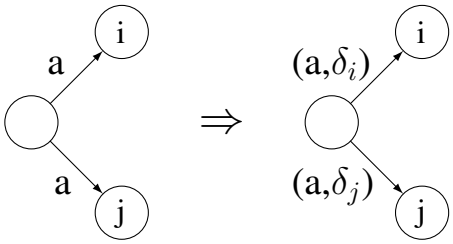


Figure 5.8: Determinization

The key then is to be able to compute the distinguishing formulæ so that determinization works correctly. Our algorithm only creates distinguishing formulæ based on what it knows. As it learns more, these formulæ will be expanded upon —by the **Split** subroutine. We must now consider the problems that could arise from building distinguishing formula in this piecewise manner. Explicitly, can they interfere with each other? Once we are assuming **Split** works as a blackbox the issue becomes proving whether our algorithm can handle multiple splits. Could a single split interfere with multiple, possibly overlapping, future mirrored branches? Consider Figure 5.9. It shows the four cases for which a second split could occur over top an existing split.

For Figure 5.9, assume that the LTS as shown represents the relevant sections of the

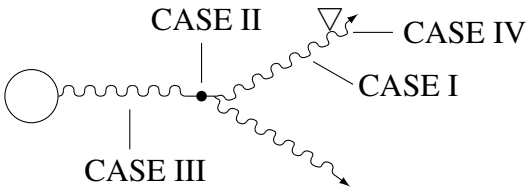


Figure 5.9: Possible Locations for additional splits

current hypothesis and that after running **Interpret** on the current counter-example we find a disagreement at the point marked $\bigtriangledown$. This is symmetric to the case in which we find the disagreement in the mirroring branch. Notice that the location of the CASE I split begins before the point of the disagreement, whereas CASE IV occurs after. CASE VI is not possible as no split would occur after the disagreement –so let us dismiss this possibility outright. Case II is occurring at the branching point of the previous split, and Case III is occurring prior to that point. As for other possible locations for the disagreement, our reasoning on its effects on splits for cases I and II will not change as long as the disagreement $\bigtriangledown$ is occurring at some point after the split. Like CASE IV we can ignore much of CASE III and some of CASE II with the following lemma:

Lemma 5.6.10. *Split will not place a new branch before a non-deterministic choice in the hypothesis for complex-errors discovered after such choices.*

Proof. Passing a non-deterministic choice in the hypothesis invokes the **Which**-subroutine. This subroutine compares whether the LTS, as it is currently known, can accept placing the new behaviour of the counter-example down-stream of the non-deterministic choice. It does this by computing whether after a total trace to the supposed branching point, the counter-example's remaining behaviours are consistent with the distinguishing formula of

the current state. If the counter-example disagreed in such a way as to call **Split** so that the branch was placed before the non-deterministic choice, this suggests that the current counter-example's behaviours do not agree at all with the current states descendants.

We can in fact show the above intuition correct by demonstrating that for a branch to be placed by **Split** prior to a non-deterministic choice, we would require conflicting responses from the membership query oracle.

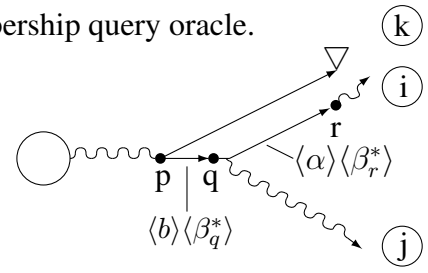


Figure 5.10: Placing a Branch Before the Non-Deterministic Choice

Consider Figure 5.10. If we label the point of the non-deterministic choice q and the current counter-example at this point φ, we can see that it cannot be the case that φ agrees with just the i-branch or both branches, and yet **Split** places the new branch at or before point q. Consider first the two queries that the **Which** subroutine would compute:

– membership$\big(\langle\langle\alpha_q^*\rangle\rangle\ (\delta_q \wedge \langle\alpha\rangle\ (\delta_j \wedge \varphi))\big) = \text{YES}$

– membership$\big(\langle\langle\alpha_q^*\rangle\rangle\ (\delta_q \wedge \langle\alpha\rangle\ (\delta_i \wedge \varphi))\big) = \text{YES}$

In the case in which **Split** places the new branch at point 'q' we would have to see these answers to **Split** queries, where $\langle\alpha\rangle\langle\beta_r^*\rangle$ is the trace (from q) to the disagreeing state, *r*, along the i-branch:

– membership$\big(\langle\langle\alpha_q^*\rangle\rangle\ \ (\delta_q \wedge (\langle\alpha\rangle\langle\beta_r^*\rangle\delta_r \wedge \langle\alpha\rangle\langle\beta_r^*\rangle EP))\big)$ = YES

– membership$\big(\langle\!\langle\alpha_q^*\rangle\!\rangle\langle\alpha\rangle\ (\delta_i \wedge (\langle\beta_r^*\rangle\delta_r \wedge \langle\beta_r^*\rangle EP))\big)$ = NO

Consider the second formula in both cases. First we know state q satisfies the formula $\langle\!\langle\beta_r^*\rangle\!\rangle\delta_{pt}$ because previous counter-examples convinced us to add these behaviours to the hypothesis. Likewise, if any state satisfies φ then it must satisfy $\langle\!\langle\beta_r^*\rangle\!\rangle EP$, because EP was derived from φ as was the total trace $\langle\!\langle\beta_r^*\rangle\!\rangle$, —albeit possibly by turning $[\,]$-operators into $\langle\rangle$-operators and use of the **Which**-subroutine. Thus the converse statement is not necessarily true. These two observations lead us to conclude that we can reduce the aforementioned membership queries as such:

– membership$\big(\langle\!\langle\alpha_q^*\rangle\!\rangle\ (\delta_q \wedge \langle\alpha\rangle\ (\delta_i \wedge \langle\!\langle\beta_r^*\rangle\!\rangle EP))\big) =$ YES

– membership$\big(\langle\!\langle\alpha_q^*\rangle\!\rangle\langle\alpha\rangle\ (\delta_i \wedge (\langle\!\langle\beta_r^*\rangle\!\rangle EP))\big)$ = NO

This result is clearly absurd as it implies YES = NO.

We find similar results if the split is place before the point q. Let p be the state where the new branch is placed (see Figure 5.10). This state has non-determinism on the b-transition. Let p' be the state that p transitions to on an action b, along the path from p to q. For split to have placed the branch a point p we would had to have seen the following two queries:

– membership$\big(\langle\!\langle\alpha_p^*\rangle\!\rangle\ \big(\delta_p \wedge \big(\langle b\rangle\langle\!\langle\beta_q^*\rangle\!\rangle\langle\alpha\rangle\langle\!\langle\beta_r^*\rangle\!\rangle\delta_r \wedge \langle b\rangle\langle\!\langle\beta_q^*\rangle\!\rangle\langle\alpha\rangle\langle\!\langle\beta_r^*\rangle\!\rangle EP\big)\big)\big)$ = YES

– membership$\big(\langle\!\langle\alpha_p^*\rangle\!\rangle\langle b\rangle\ \big(\delta_{p'} \wedge \big(\langle\!\langle\beta_q^*\rangle\!\rangle\langle\alpha\rangle\langle\!\langle\beta_r^*\rangle\!\rangle\delta_r \wedge \langle\!\langle\beta_q^*\rangle\!\rangle\langle\alpha\rangle\langle\!\langle\beta_r^*\rangle\!\rangle EP\big)\big)\big)$ = NO

But again since EP was derived from φ, and since the initial **Which**-subroutine included the query: membership$\big(\langle\!\langle\alpha_q^*\rangle\!\rangle\ (\delta_q \wedge \langle\alpha\rangle\ (\delta_i \wedge \varphi))\big) =$ YES we know that the target LTS satisfies:

$$\big(\langle\!\langle\alpha_p^*\rangle\!\rangle\langle b\rangle\langle\!\langle\beta_q^*\rangle\!\rangle\langle\alpha\rangle\langle\!\langle\beta_r^*\rangle\!\rangle EP\big) \tag{5.21}$$

This is just $\langle\langle \alpha_q^* \rangle\rangle \langle \alpha \rangle \varphi$ in disguise. We also know the target satisfies:

$$\left(\langle\langle \alpha_p^* \rangle\rangle \langle b \rangle \langle\langle \beta_q^* \rangle\rangle \langle \alpha \rangle \langle\langle \beta_r^* \rangle\rangle \delta_r\right) \tag{5.22}$$

because these actions are in the existing hypothesis. Therefore we know the target should satisfy:

$$\langle\langle \alpha_p^* \rangle\rangle \langle b \rangle \ (\delta_{p'} \wedge (\langle\langle \beta_q^* \rangle\rangle \langle \alpha \rangle \langle\langle \beta_r^* \rangle\rangle \delta_r \wedge \langle\langle \beta_q^* \rangle\rangle \langle \alpha \rangle \langle\langle \beta_r^* \rangle\rangle EP)) \tag{5.23}$$

Again contradicting the result of the **Split**-subroutine queries, completing the proof. □

Theorem 5.6.11 (Consistency of Overlapping Splits)**.** *If we have a non-deterministic branching point where both the distinguishing formulæ have the correctly-distinguishing property, then a split involving one of these branches will maintain the property for all branching points involved, be they the same point or not.*

Proof. Clearly from section 5.5.7 we can see that all identified splits maintain the correctly distinguishing property when there have been no previous splits regardless of whether branches are being placed properly.

We consider adding new branches to the points identified in Figure 5.9:

CASE I: On a branch

This case involves splitting along a branch mirrored by a previous split. Since the split disagreement is only appearing on branch 'i' and not the other branch 'j' we can make several assumptions:

– Either at point q the current counter-example's outermost modal operator was $[\alpha]\varphi$ and then:

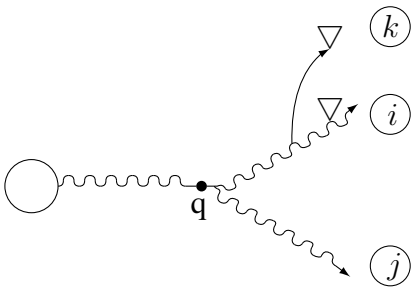


Figure 5.11: CASE I

a) –branch 'j' already modeled φ

– Or at point q the current counter-example's outermost modal operator was $\langle\alpha\rangle\varphi$ and a call to the **Which**-subroutine resulted in either:

b) –branch 'j' failing to agree with φ (through δ_j)

c) –both branches agreeing with φ and branch 'i' was chosen arbitrarily to accommodate the new behaviours

However, for Case I our reasoning for subcases a,b, and c will be essentially the same. These same three subcases will be considered in Case II as well –so take care to remember them. The reason that here a,b, and c are irrelevant is because of the determinization process of Figure 5.8. Once we have passed point q in the **Interpret** subroutine we can effectively ignore the rest of the tree, including branch j, because the total-traces in our membership-queries will affix any queries we make to the sub-tree starting at branch 'i'. In terms of the target this could refer to affixing behaviours to a set of states not yet distinguished in the hypothesis. Thus all we need to be concerned with is whether the 'k' branch disagrees with the 'i' branch so that we can maintain the correctly-distinguishing property. This of course will be the case if a single split works correctly.

CASE II: At the branching point

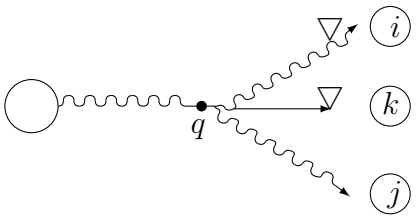


Figure 5.12: CASE II

In this case, the new branch is being added at the point where the previous two branches began. Thus for the correctly-distinguishing property to hold, the i,j, and k branches will have to be in pairwise disagreement. Let us consider the subcases a),b) and c) we identified earlier:

a) This cannot be the case because it does not resolve the $[\,]$-operator since the 'i' and 'j' branches are left unchanged. (Compare this to split TYPE IV)

b) These cannot be the case due to Lemma 5.6.10

c) same as b)

Additionally, CASE II has an additional subcase d); which, as it turns out is the only possible way for a branch to be placed at such a point. This case is where the disagreement occurs at q and the new branch is place at state q. This only happens, if during execution of the **Which**-subroutine, we find no current branches able to accept the changes. In this case the distinguishing formula for the new branch includes its disagreement with all previous branches. Thus, this new branch is distinguished from each previous branch; and, if all previous branches were in pairwise disagreement, then so are all the new branches.

In conclusion, we have enumerated through all possible instances of adding a branch where existing branches already exist, and seen that doing so maintains the correctly-distinguishing property. Repeated applications of the above argument inductively show that this property is maintained throughout the execution of the whole algorithm. This completes the proof.

□

5.6.3 Correctness of Distinguishing Formulæ

The proof of Theorem 5.6.11 implicitly assumed that the algorithm was building and maintaining correct distinguishing formulæ; not only for the current state but for all states preceding it. What we would like to do is come up with an equivalent formulation of counter-examples in all tree LTS similar to the Γ formulæ of Lemma 5.6.7. The class of Γ formulæ not only implied that all counter-examples could be converted into this form, but that in fact this class of formulæ was sufficient to describe deterministic tree-LTSs. The easiest way to do this is to take the distinguishing formulæ from the determinization of a tree-LTS (see Figure 5.8) and add them to the Γ formula. Since the old Γ formula were built only from $\wedge$-connectives and $\langle\rangle$-operators, the new Γ^+ class of formula now contains the original $\wedge$-connectives, $\langle\rangle$-operators and whatever constitutes the distinguishing formulæ δ. Since the δ's are derived from the EP's of the algorithm they contain the $\vee$-connectives and $[\,]$ operators one would expect from a full distinguishing formula. All the Γ^+ formulæ will do is logically organize the formula into deterministic parts and non-deterministic parts, reflecting the determinization of LTSs. Again, it is a way of logically grouping formulæ; it is not of any vital theoretical importance.

We define the Γ^+ formulæ by segmenting an LTS into deterministic parts as follows:

Definition 5.6.12 (Deterministic Segments of an LTS)**.**

1) Given an LTS L with start state s we do the following:

2) Add s to the current segment

3) For every deterministic transition leaving s for a state p,

- *add the deterministic transition to the current segment*
- *return to step 2) considering state p as the start state of some sub-tree LTS within the current segment*

4) For every individual transition leaving s for a state p that is non-deterministically mirrored

- *do not put this non-deterministic transition in any segment*
- *return to step 2) considering state p as the start state of some sub-tree LTS within a* new *segment*

For example, Figure 5.13 shows two similar LTSs with one change in labeling. The left-hand LTS has segments A,B,C,D, and E. The states are labeled with the segment they belong to. The righthand LTS has only three segments A,B, and C. Again the states are labeled with the segment they belong to. The dead state is the only state that belongs to multiple segments; this is a technicality really, as we never care what segment it is in, as we never update it.

Segmenting the LTS this way converts an LTS into a collection of deterministic sub-LTSs, connected by non-deterministic transitions of the original LTS. Thus each segment has a Γ formula that describes it. Recall each Γ formula contains as a subformula a trace to each state. We call those parts of the Γ formula that lead to states which have transitions

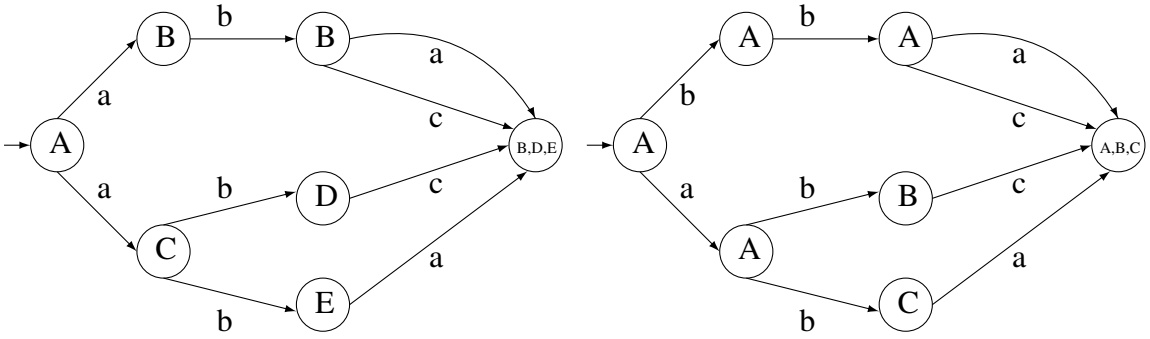


Figure 5.13: Segmentation

that connect segments *contact points*. The idea is we can remove the atomic *tt* formula from the ends of the contact points and replace it with the formula:

$$\bigwedge_{\alpha \in A, \beta \in B} \langle \alpha \rangle \left(\delta_{st_i} \Gamma_\beta \right) \tag{5.24}$$

where A is the set of non-deterministic transitions leaving the contact point state and arriving at state st_i, and B the set of connected segments respectively. Notice the segmentation procedure will create a tree of segments, where the root segment is the one which contains the start state. We can thus define for each segment a level, where level 0 is the root segment, and all other segments statisfy:

(level of segment) = (level of parent segment) $+1$.

Given the segmentation of an LTS L and distinguishing formulæ δ_i we can define Γ^+ formulæ as:

Definition 5.6.13 (Γ^+ Formulæ)**.** *For a segmentation of a LTS L, with segment levels* $0 \ldots n$*:*

- *For each segment at level* $n-1$ *we want to create a* Γ^+ *formula for the non-deterministic tree-LTS rooted at the root state of that segment.*
- *We do this by taking the* Γ *formula* $\Gamma_1 \ldots \Gamma_m$ *for the level* $n-1$ *segments and at their contact points, we attach the associated* Γ *formula for the level n segment, using the* δ_{st_i}*'s (Equation 5.24).*
- *For each segment starting at segments in levels* $n-2$ *through to 0:*
  - *For each segment in the current level create a* Γ^+ *formula for the tree rooted at the root state of that segment by attaching at the contact points of the* Γ *formula for that segment the* Γ^+ *formula created for the descending segments in the previous round.*
- *The* Γ^+ *formula created for the segment at level 0 is the* Γ^+ *formula for L*

Lemma 5.6.14. *Algorithm 5.5.1 maintains and updates distinguishing formula correctly.*

Proof. For non-deterministic targets we are going to view the algorithm in phases. Each phase will represent a period of discovery of simple-errors. These phases are separated by the discovery of complex-errors. The segments represent these phases. We show that the distinguishing formula that the algorithm builds are the Γ^+ formula, which are sufficient to describe non-deterministic Tree-LTS.

Base Case:

The hypothesis is a single segment, therefore it has a Γ formula since it is deterministic. After a split it will become three segments, two new. Thus we have three corresponding

Γ formula: Γ_1, Γ_a, and Γ_b. Γ_a and Γ_b are non-deterministically mirrored. We know our algorithm already correctly creates Γ_a and Γ_b The key to the fact that **Split** creates correct Γ formulæ is line 7 of **Split**. This shows that the distinguishing formulæ we create are attached at the contact point, and new mirroring states are created in a way such that they have corresponding Γ formulæ. If one examines Algorithm 5.5.1 we find that the two states we arrive at from the contact points connecting these two segments are correctly distinguished. Let these correctly distinguishing formulæ be δ_a and δ_b. Furthermore the algorithm stores these states' distinguishing formulæ as their respective Γ-formula conjuncted with their respective distinguishing formula. The distinguishing formula for the start state would be $\Gamma_1|_{a,b}(\Gamma_a \wedge \delta_a), (\Gamma_b \wedge \delta_b)$, where the operator $|_{a_1,\ldots,a_n}\varphi_1 \ldots \varphi_n$ denotes substituting $\varphi_1 \ldots \varphi_n$ at the contact points $a_1 \ldots a_n$, respectively.

Inductive Hypothesis:

We have a hypothesis at stage i of the algorithm with a correct Γ^+-formula

Inductive Step:

Since **Split** does not place the start of new branches before previous non-deterministic choice, we can conclude that the act of splitting either turns a segment into two segments, in the case that the complex error is found at the very beginning of a segment, or into three segments. That last case is exactly like the base case but considering the sub LTS starting with the first state in the segment to be split. The first case is not much different. Such a split is the same as making another substitution at the contact point in the Γ^+ formula where the segment to be split is currently attached. The proof follows by inspection of the Algorithm's **Split** subroutine.

□

5.6.4 Effective Subtrees

We have already proven that if **Split** works correctly then distinguishing formula correctly distinguish among non-deterministic choices at the same state. Once we prove that both: **Split** works (branches place correctly) and that the **Which**-subroutine works (we follow a non-deterministic choice that allows us to make a correct update), we will have proved that the algorithm works essentially the same as if the target were deterministic during deterministic phases, and that in between phases the complex errors are correctly being updated. We have determinized a non-deterministic LTS. Thus our proof of correctness will be similar to the deterministic case. We will show that the algorithm produces a sequence of subtrees as hypothesis. This sequence ends with the target and each previous hypothesis as a subtree of its successor.

To begin, we must first define a a new type of effective sub-tree; this new subtree reflects the same requirements as in Section 5.4.4. We define this new sub-tree by providing a procedure to create such sub-trees:

Procedure 5.6.15.

1) Define 3 types of transitions:
   a) normal $\xrightarrow{\alpha}$ transitions between two states
   b) labeled *no-transitions*, leaving a single state, going nowhere, expressing a state cannot admit an action
   c) labeled *possible-transitions*, leaving a single state, going nowhere, expressing a state which might admit an action, or, is known to eventually be distinguished (Figure 5.3).

2) For every state s, and for all characters $\alpha \in \Sigma$, such that there is no α-transition leaving state s, add an α-labeled no-transition.

3) Note: we define a "*total*" tree as a tree where every state has for every character either a single no-transition labeled with that character or at least one labeled transition with said character. A total tree has no possible-transitions.

4) Given the associated total tree for L, L_{total}, we begin crafting a sub-LTS by arbitrarily removing states with one caveat:

5) We do not remove the start state.

6) For each state we remove we:

   i) remove all transitions (of every variety) leaving that state , and every state descendent from that point onwards; leaving only states reachable from the start state.

   ii) every $\xrightarrow{\alpha}$ transition that entered that state becomes an α-labeled possible-transition.

7) after removing some amount of states we can arbitrarily change some labeled no-transitions into labeled possible-transitions of the same label.

8) given this stripped down LTS L' we combined states that appear bisimilar until no more such minimizing is possible. This operation is done with two caveats:

   i) possible-transitions act as a wild-cards, thus the resulting sub-LTS may not be unique.

ii) during minimization the LTS must remain a tree.

9) Remove all possible-transitions and no-transitions.

Definition 5.6.16 (Effective Subtrees)**.** *We define an effective subtree of an LTS L, as any LTS that can result from using Procedure 5.6.15 on L:*

The above process of making effective subtrees is designed to be the reverse process of learning tree LTSs; for instance the effective subtree will be a single component which includes the start state. Furthermore the act of merging bisimilar states where possible-transitions act as wild cards, is the reverse of using a counter-example to split two states. It is as if, when learning, every new state we create has a possible-transition for all characters. As the counter-examples are received we are determining, in reverse, whether possible-transitions represent no-transitions or actual-transitions.

We want to formalize the notion that the hypothesis our algorithm creates will be a sequence of these effective subtrees:

Conjecture: *Algorithm 5.5.1 computes as hypothesis a series of effective subtrees of the target LTS.*

We are not quite ready to prove this conjecture yet. Informally the proof we want goes like this: The phases of the algorithm are periods of discovery of simple errors, thus all modifications to the hypothesis during these phases are through updates caused by simple errors. Though, to believe this we need to convince ourselves that if a counter-example is not revealing a *COMPLEX-ERROR*, we will not find any such errors. This amounts to proving the **Which**-subroutine works. Given that, these *SIMPLE-ERROR* updates ei-

ther add just a new transition or a new state and new transition. Since these states are the results of counter-examples which reflect behaviour of the target, this represents the reverse operation of removing states from the procedure of Definition 5.6.16. Likewise, as we have tried to motivate previously, the **Split** Subroutine is the reverse of merging apparently bisimilar states in the procedure of Definition 5.6.16. To believe this we need to show **Split** places new branches correctly.

Lemma 5.6.17. *The **Which** subroutine selects a non-deterministic choice that leads to a section of the hypothesis that contains a difference from the counter-example.*

Consider an aside for a moment. Our concern is that the target might be designed in some confusing way such that the distinguishing actions of states overlap in ways making it hard to separate them. However, we are querying the target using total traces containing distinguishing formulæ from the hypothesis. We are assuming the hypothesis is correct at the first step. Furthermore, we have already shown all our updates maintain the correctly-distinguishing property, as well we shall show that updates leave the new hypothesis as effective subtrees of the target. Thus we do not need to worry about future confusion possible in the target —the way that the queries are structured allow us to make correct changes with regards to what we do know.

Proof. The point of this is to prove that errors are corrected in the right spot. Say we have our current hypothesis, and let us assume that it is an effective sub-tree of the target. Let us say we have a state p in the hypothesis where **Interpret** has found an error δ. For the proof we need only assume we have called **Which** at least once, though we likely called the subroutine several times. When called, **Which** proceeds by comparing the behaviours of the distinguishing formulæ of the non-deterministic successor states with the behaviours

in the counter-example. **Which**'s membership queries amount to verifying that $\langle\langle \alpha_p^* \rangle\rangle \delta$ is a satisfied by the target, where $\langle\langle \alpha_p^* \rangle\rangle$ is the total trace to the state p in the hypothesis. This would otherwise amount to a total trace in the target with the exception that all the distinguishing formula within the total trace are culled from the hypothesis LTS. Thus $\langle\langle \alpha_p^* \rangle\rangle$ represents a set of states in the target. Certainly, if $\langle\langle \alpha_p^* \rangle\rangle \delta$ were true in the target then it should be true in the hypothesis; but, since it was derived from a counter-example, this trace is not true. There is currently only one state in the hypothesis that has that total trace, thus only one state we *need* to change. Thus **Which** picks a correct path. □

Now that we know errors are placed correctly, we want to show that we do not miss the errors elucidated in the counter-example.

Lemma 5.6.18. *If δ is a counter-example then Interpret(δ) $\neq$* NO-ERROR.

Proof. The **Interpret** subroutine does a traversal of the entire hypothesis —behaving as a simple model-checking algorithm. We show that if it returns *NO-ERROR*, this would mean the hypothesis satisfied the counter-example, a contradiction. Recall we assume counter-examples are all of the form (δ, YES) demonstrating HML formulæ the LTS should satisfy, but does not. This is equivalent to showing that **Interpret** works correctly as a model checker, returning *NO-ERROR* if and only if the counter-example is satisfied by the hypothesis. Given the format of distinguishing formulæ we can summarize the definition of satisfaction as:

1) $s \models \langle\alpha\rangle(\bigwedge \delta_i)$ if there exists a state p such that $s \xrightarrow{\alpha} p$ and $\forall i,\ p \models \delta_i$

2) $s \models [\alpha](\bigvee \delta_i)$ if for all states p_i such that $s \xrightarrow{\alpha} p_i$, there exists δ_j such that $p_i \models \delta_j$

3) $s \models \langle\alpha\rangle tt$ if there exists a state p such that $s \xrightarrow{\alpha} p$

4) $s \models [\alpha]ff$ if for all states p, there is no transition $s \xrightarrow{\alpha} p$

We deduce from these the equivalent rules for when a state fails to satisfy a formula:

1) $s \not\models \langle\alpha\rangle(\bigwedge \delta_i)$ for all states p_i such that $s \xrightarrow{\alpha} p_i$, $p \not\models \delta_1$ or ... or $p \not\models \delta_n$

2) $s \not\models [\alpha](\bigvee \delta_i)$ if there exists a state p such that $s \xrightarrow{\alpha} p$, and for all δ_j $p \not\models \delta_1$ and ... and $p \not\models \delta_n$

3) $s \not\models \langle\alpha\rangle tt$ if for all states p there is no transition $s \xrightarrow{\alpha} p$

4) $s \not\models [\alpha]ff$ if there exists a state p, such that $s \xrightarrow{\alpha} p$

The Interpret subroutine has four subsections, one for each of the above satisfaction rules. Let us verify that each of the four subsections would only return no error in the correct case. In the order of the four subsections:

- if cce = $\langle\alpha\rangle tt$

  It is clear that we only return NO-ERROR if the current state has one or more α-transitions, that is $\exists p \in$ STATES, such that the current state $s \xrightarrow{\alpha} p$.

- if cce = $\langle\alpha\rangle(\bigwedge_{i=1}^{m} \varphi_i)$

  Firstly, we only return *NO-ERROR* if there is at least one α-transition. In the case of a single α-transition we only return NO-ERROR if for all the φ_i's and the single state p such that the current state s transition to p on α, we find $p \models \varphi_1$ and ... and $p \models \varphi_n$. In the case of multiple α-transitions we use the **Which**-subroutine to check all p_j such that the current state $s \xrightarrow{\alpha} p_j$. The examination the **Which**-subroutine does involves verifying whether the target LTS has a state $p_j \models (\bigwedge \varphi_i) \wedge \delta_{p_j}$. Since the hypothesis must be consistent with the target any $p_j \not\models (\bigwedge \varphi_i) \wedge \delta_{p_j}$ should remain not satisfying this formula –that is, return *NO-ERROR*. Any $p_j \models (\bigwedge \varphi_i) \wedge \delta_{p_j}$ in

the target could satisfy the same formula in the hypothesis. In this case we arbitrarily pick such a p_j, and the algorithm continues as if that α-transition were the only one –as we discussed previously (see Section 5.5.5). Recall the δ_{p_j} such that $p_j \models (\bigwedge \varphi_i) \wedge \delta_{p_j}$ may not yet be mutually exclusive, so we only make a change down one path.

- □ if cce $= [\alpha]ff$

  This returns *NO-ERROR* if there are no other transitions. That is, for every other state there is no transition which leads to it from the current state.

- □ if cce = $[\alpha](\bigvee_{i=1}^{m} \varphi_i)$

  There are several ways we return *NO-ERROR*. First if there is no α-transition then the state trivially satisfies the formula as the statement: $\forall p \in$ STATES such that current state $s \xrightarrow{\alpha} p$, and $p \models \delta_1$ and ... and $p \models \delta_n$, is vacuously true. In the case that we have at least one α-transition, then for each and every state p such that our current state $s \xrightarrow{\alpha} p$, we look at only the δ_j among $(\bigvee \delta_i)$ such that the target has some analogous state s' for which the target LTS $\models \langle\langle \alpha^*_{s'} \rangle\rangle (\delta_p \wedge \varphi_j)$ as these are the φ_i's that each individual p should satisfy. It does not matter if there are some φ_i such that some $p \not\models \varphi_i$. In fact, we only need each p to satisfy one φ_i. So we return *NO-ERROR* if for every state p, there is some φ_i that p satisfies.

Thus, if we compare the instances where **Interpret** returns *NO-ERROR*, they are exactly those instances where we have confirmed that the states of the LTS satisfy the counter-example. Likewise, if any kind of error is returned, it is from exactly those states that do not satisfy a given formula.

□

The final two lemmas prove that if we do indeed find the error, be it simple or complex, then the new hypothesis will be a subtree of the target.

Lemma 5.6.19. *Assuming that the current hypothesis is an effective subtree of the target and given a counter-example δ with Interpret(δ) = SIMPLE-ERROR, then correcting the current hypothesis with Update(SIMPLE-ERROR) produces a new hypothesis which is an effective sub-tree and still maintains the correctly-distinguishing property.*

Proof. There are two ways a Simple Update works, both very similar. In one we are just adding a transition to a known state. In the case of tree LTS, this will always be the dead state —or rather a state that still appears bisimilar to a dead state. Otherwise we are adding both a transition and a new state for that transition to lead to. The new state itself will have a transition to the dead state, to reflect its distinguishing formula as best as it is known.

In the first case, we are building a transition to a state whose further actions we know nothing about. Thus, as best we know it is bisimilar to the dead state. In the second case we know that the new state can do some further actions that distinguish it from the dead state. Let δ be the simple error –the actions that the current state p should do, but does not do. In the second case we can consider δ to additionally have both the actions leading to the new state, and those leaving it for the dead state. What is left to show is that these outlined changes would leave the new hypothesis as an effective subtree of the target. Consider that **Which** computes the total trace, where the distinguishing formulæ within are all drawn from the hypothesis (Section 5.6.17). Furthermore $\langle\langle \alpha_p^* \rangle\rangle$ is part of the current structure of the hypothesis, which we are both assuming to be correct, and assuming to be an effective subtree of the target. Finding the simple error δ amounts to querying $\langle\langle \alpha_p^* \rangle\rangle \delta$ in the target but with distinguishing strings from the hypothesis, and finding it should be

true. Thus $\langle\langle \alpha_p^* \rangle\rangle \delta$ should be true in the hypothesis as well.

Moreover, $\langle\langle \alpha_p^* \rangle\rangle$ is a unique state in the hypothesis (again because the distinguishing formula were in the total trace were drawn from the hypothesis and since the hypothesis is assumed to satisfy the correctly distinguishing property.) Thus we can map the unique state $\langle\langle \alpha_p^* \rangle\rangle$ in the the hypothesis to a set of states $P = \langle\langle \alpha_p^* \rangle\rangle$ in the target which all satisfy δ. Since the current hypothesis is an effective sub-Tree there exists a set of procedures from Definition 5.6.16 that allowed the set of states P to be collapsed into a single state. If we replace these procedures with one that removes all that distinguishes the states of P, except the δ actions, this gives us a new effective sub-Tree equivalent to the new hypothesis we build. Thus the new hypothesis is an effective subtree of the target. We have not added any new non-determinism, so the hypothesis still also satisfies the correctly-distinguishing property.

This completes the proof.

□

Lemma 5.6.20. *Assuming that the current hypothesis is an effective subtree of the target and given a counter-example δ with Interpret(δ)* = COMPLEX-ERROR *then correcting the current hypothesis with Update(*COMPLEX-ERROR*) produces a new hypothesis which is an effective sub-tree, and still maintains the correctly-distinguishing property.*

Proof. We are faced with a similar situation as in the previous proof. **Interpret** has found a total trace $\langle\langle \alpha_p^* \rangle\rangle$ to a unique state in the hypothesis —with distinguishing formula culled from within the hypothesis, which we now denote as $\langle\langle \alpha_p^* \rangle\rangle_{hyp}$. Furthermore, that state has a single disagreeing action in the form of a complex-error δ; that is, the hypothesis should satisfy δ but cannot without removing some actions. Rather, that unique state currently

satisfies $\neg\delta$. Likewise from the assumed correctness of the existing hypothesis we know that in the target we have a set of states $\{\langle\langle\alpha_p^*\rangle\rangle_{hyp}\}$, some of which satisfy at least δ. From the counter-example we know we can divide the set of states $\{\langle\langle\alpha_p^*\rangle\rangle_{hyp}\}$ into two disjoint sets:

1) $\{\langle\langle\alpha_p^*\rangle\rangle_{hyp}\delta\}$

2) $\{\langle\langle\alpha_p^*\rangle\rangle_{hyp}\neg\delta\}$

These states have exactly the same trace, but they satisfy incompatible formulæ and therefore must be disjoint. Thus there must be some earliest prefix $\langle\langle\langle\alpha_p^*\rangle\rangle_{hyp}|_i$ of $\langle\langle\alpha_p^*\rangle\rangle_{hyp}$ where the above two disjoint sets of states start to diverge, in the sense that one of the sets will at some point fail to have any states in it still reachable from the set of states $\{\langle\langle\langle\alpha_p^*\rangle\rangle_{hyp}|_i\}$. By this we mean that the target satisfies:

$$\langle\langle\langle\alpha_p^*\rangle\rangle|_i \left(|\langle\langle\langle\alpha_p^*\rangle\rangle\rangle_i\delta \wedge |\langle\langle\langle\alpha_p^*\rangle\rangle\rangle_i\neg\delta\right) \tag{5.25}$$

but not:

$$\langle\langle\langle\alpha_p^*\rangle\rangle|_{i+1} \left(|\langle\langle\langle\alpha_p^*\rangle\rangle\rangle_{i+1}\delta \wedge |\langle\langle\langle\alpha_p^*\rangle\rangle\rangle_{i+1}\neg\delta\right) \tag{5.26}$$

for some i. Furthermore, we know this because the target satisfies:

$$\langle\langle\langle\alpha_p^*\rangle\rangle|_0 \left(|\langle\langle\langle\alpha_p^*\rangle\rangle\rangle_0\delta \wedge |\langle\langle\langle\alpha_p^*\rangle\rangle\rangle_0\neg\delta\right) \tag{5.27}$$

but not:

$$\langle\langle\langle\alpha_p^*\rangle\rangle|_n \left(|\langle\langle\langle\alpha_p^*\rangle\rangle\rangle_n\delta \wedge |\langle\langle\langle\alpha_p^*\rangle\rangle\rangle_n\neg\delta\right) \tag{5.28}$$

where n is the number of transitions to the disagreeing state in question. This state i must be the location of, at least one, non-deterministic branch in the target, as no state satisfies $\delta \wedge \neg\delta$. In such a case **Split** adds states that correspond to each element of the

trace suggested by the $|\langle\langle\alpha_p^*\rangle\rangle_{hyp})_i$ suffix of Equation 5.25. We know the target has these states, because we know some states must mirror a suffix of the total trace $\langle\langle\alpha_p^*\rangle\rangle$ and yet eventually admit a δ action. Furthermore, since the target is a tree-LTS, these states must therefore all be new, and no current states with corresponding traces exist in the hypothesis, otherwise we would not have a counter-example.

Finally, it is not hard to see that the type of update **Split** would do would leave us with an effective subtree, because if we removed all actions from the sets $\{\langle\langle\alpha_p^*\rangle\rangle\delta\}$ and $\{\langle\langle\alpha_p^*\rangle\rangle\neg\delta\}$ except the δ and $\neg\delta$ actions, these two sets would collapse into two single but distinct states. This is just as we have in the hypothesis, with two separate but mirrored traces leading to them from point i. The sole exception is that the last new state may be currently bisimilar to the dead state. □

Corollary 5.6.21. *The algorithm maintains the Correctly Distinguishing Property.*

Proof. This is a result of the placement of new branches by the **Split** subroutine being consistent with the target, as proved in Lemma 5.6.20 □

Using this as a base case with Lemmas 5.6.19 and 5.6.20 as the inductive step, the proof follows directly. Thus we have all we need to prove the algorithm correct.

Theorem 5.6.22 (Correctness of Algorithm 5.5.1)**.** *Algorithm 5.5.1 correctly learns any tree-LTS given as a target.*

Proof. The first hypothesis we pick is just a start state with no transitions. This is an effective subtree satisfying the correctly-distinguishing property. By the previous lemmas, at every step of the algorithm, if we are given a counter-example, **Interpret** will return either *SIMPLE-ERROR* or *COMPLEX-ERROR*. In both cases, the new hypothesis will be

an effective subtree, and maintain the correctly distinguishing property. Since the target is of finite size and since each hypothesis is both a subtree, but also adds at least one new state or one new transition, eventually one of the hypotheses must equal the target. This terminates the algorithm.

The time complexity of the algorithm is $O(n^3)$ as the number of transitions and states is $O(n)$ in the size of the LTS, measured in the number of states. Thus we will have no more then $O(n)$ iterations. Each iteration considers as many as $O(n)$ states with distinguishing formula whose complexity is no bigger then $O(n)$. We also must compute the total traces, but that does not increase the time complexity of a single round. Thus $O(n) \times O(n) \times O(n) = O(n^3)$. □

5.7 An Example

This section contains an example of the execution of the algorithm. We will pick as a target an LTS very similar to the ones seen in Figure 3.1. We have used this LTS for several examples, and now we shall see how to learn it. Our actual target is illustrated in Figure 5.14. We denote calls to functions in **Boldface**. Recall *COMPLEX-ERROR* and *SIMPLE-ERROR* are not function calls, they are parameterized data types.

Steps:

1) Initial hypothesis $hyp =$ LTS with a single starting state

2) **Equivalent**(hyp) $= (\langle ab \rangle [c] ff, \text{Yes})$

3) let $hyp =$ Figure 5.15

4) **Equivalent**(hyp) $= [a]\langle b \rangle tt$

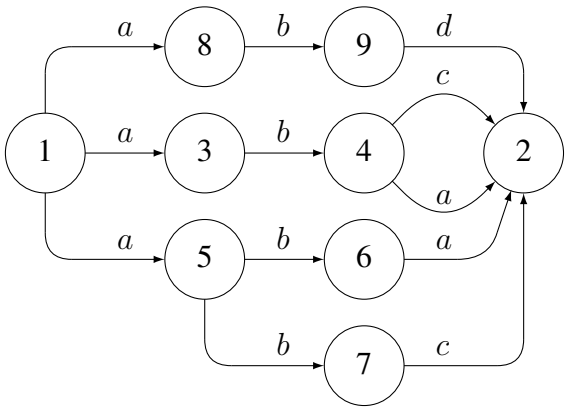


Figure 5.14: The Target LTS

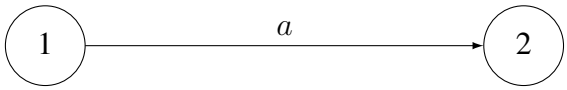


Figure 5.15: The First Hypothesis

5) **Interpret**($\langle\,\rangle$,①, $[a]\langle b\rangle tt$)

6) We start **Interpret** at line 46 with $cce = [a](\langle b\rangle \vee \emptyset)$

7) This is Case II of line 49, as we have at least one a-transition

8) For each a-transition to state st_j, or rather the single a-transition to state②

   8a) Call **Or-Elimination**($\langle\!\langle \alpha^*_{②} \rangle\!\rangle$,②, $(\langle b\rangle tt \vee \emptyset)$)

   8b) $\langle\!\langle \alpha^*_{②} \rangle\!\rangle = \langle\alpha\rangle(\cdot)$

   8c) test **Membership**($\langle a\rangle(\emptyset \wedge \langle b\rangle tt)$) = Yes

9) We return to line 56 of **Interpret** and call **Interpret**($\langle a \rangle$,②, $\langle b \rangle tt$)

10) This leads us to lines 2-3 of **Interpret** where we return *SIMPLE-ERROR*($\langle\,\rangle$,①, a, b)

11) Ultimately this leads us to **Update**(*SIMPLE-ERROR*($\langle\,\rangle$,①, a, b)). This would lead us to construct the LTS of Figure 5.16. It would have trace and distinguishing formula as described in Table 5.1.

Table 5.1: Trace and Distinguishing Formulæ of the Second Hypothesis

State:	1	3
Trace:	$\langle\,\rangle$	$\langle\,\rangle\langle a \rangle$
δ:	$\langle a \rangle \delta_3$	$\langle b \rangle tt$

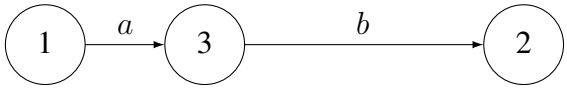


Figure 5.16: The Second Hypothesis

12) **Equivalent**(hyp) = $[a][b](\langle a \rangle tt \vee \langle c \rangle tt \vee \langle z \rangle tt)$

13) This leads to line 49 of **Interpret** where cce = $[a](\underbrace{[b](\langle a \rangle tt \vee \langle c \rangle tt \vee \langle z \rangle tt)}_{\varphi_1} \vee \emptyset)$

13) Call **Or-Elimination**($\langle a \rangle$,③, φ_1)

14) Test **Membership**($\langle a \rangle(\langle b \rangle tt \wedge \varphi_1)$ = Yes

15) We return to lines 5-56 and call **Interpret**($\langle a \rangle$,③, φ_1).

15a) This leads us to line 49 of **Interpret** with cce =$[b](\underbrace{\langle a \rangle tt}_{\varphi_1} \vee \underbrace{\langle c \rangle tt}_{\varphi_2} \vee \underbrace{\langle z \rangle tt}_{\varphi_3})$

15b) Compute **Or-Elimination**($\langle ab\rangle$,②, φ_1) = **Membership**($\langle ab\rangle(\emptyset \wedge \langle a\rangle tt$) = YES

15c) Compute **Or-Elimination**($\langle ab\rangle$,②, φ_2) = **Membership**($\langle ab\rangle(\emptyset \wedge \langle c\rangle tt$) = YES

15d) Compute **Or-Elimination**($\langle ab\rangle$,②, φ_3) = **Membership**($\langle ab\rangle(\emptyset \wedge \langle z\rangle tt$) = NO

15e) We return: $\langle a\rangle tt \vee \langle c\rangle tt$

16) Compute: **Interpret**($\langle ab\rangle$,②, φ_1) and **Interpret**($\langle ab\rangle$,②, φ_2)

17) Both return *SIMPLE-ERROR* from line 5 of **Interpret**: *SIMPLE-ERROR*($\langle a\rangle$,③, b, a) and *SIMPLE-ERROR*($\langle a\rangle$,③, b, c)

18) Since none of these are *NO-ERROR*, we arbitrarily pick *SIMPLE-ERROR*($\langle a\rangle$,③, b, c) to update with.

19) This gives us the Hypothesis of Figure 5.17 with trace and distinguishing formula as described in Table 5.2.

Table 5.2: Trace and Distinguishing Formulæ of the Third Hypothesis

State:	1	3	4
Trace:	$\langle\ \rangle$	$\langle a\rangle$	$\langle ab\rangle$
δ:	$\langle a\rangle\delta_3$	$\langle b\rangle\delta_4$	$\langle c\rangle tt$

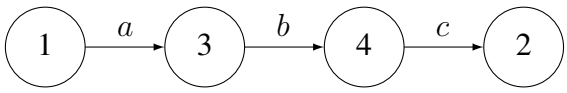


Figure 5.17: The Third Hypothesis

20) **Equivalent**(hyp) = $\langle aba\rangle tt$

21) Leads to line 32 of **Interpret** with cce = $\langle a \rangle (\langle ba \rangle tt \wedge \emptyset)$

22) This would result in a call to **Interpret**($\langle a \rangle$,③, $\langle ba \rangle tt$)

23) This would result in a further call to **Interpret**($\langle ab \rangle$,④, $\langle a \rangle tt$)

24) This would lead to line 6 of **Interpret** where we test the value of **Membership**($\langle ab \rangle (\langle a \rangle tt \wedge \langle c \rangle tt)$) = Yes

25) We would return *SIMPLE-ERROR*($\langle ab \rangle$,④, $\langle a \rangle, \emptyset$)

26) This update would give us the Hypothesis of Figure 5.18 with trace and distinguishing formula as described in Table 5.3.

Table 5.3: Trace and Distinguishing Formulæ of the Fourth Hypothesis

State:	1	3	4
Trace:	$\langle\,\rangle$	$\langle a \rangle$	$\langle ab \rangle$
δ:	$\langle a \rangle \delta_3$	$\langle b \rangle \delta_4$	$\langle c \rangle tt \wedge \langle a \rangle tt$

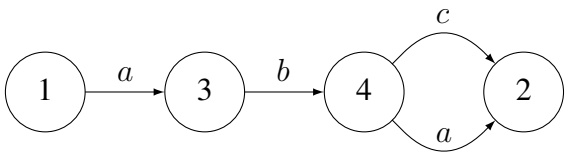


Figure 5.18: The Fourth Hypothesis

27) **Equivalent**(hyp) = $\langle ab \rangle [c] ff$

28) This naturally leads to us considering **Interpret**($\langle ab \rangle$,④, $[c] ff$)

29) This leads to line 34 of **Interpret** and us returning *COMPLEX-ERROR*($\langle ab \rangle$,④, $[c] ff$)

30) This calls **Split**$(\langle ab\rangle, ④, [c]ff)$

31) For each prefix of $\langle ab\rangle$ from $\langle a\rangle$ to $\langle a\rangle$ we compute:

31a) **Membership**$(\langle a\rangle(\delta_3 \wedge (\langle b\rangle\delta_4 \wedge \langle b\rangle[c]ff)))$ = NO

31b) Fails for the first prefix

31c) Create states: $\langle a\rangle$ with distinguishing formula $\langle b\rangle[c]ff$

31d) We send a b-transition to dead state with a tentative distinguishing formula of $[c]ff$

32) This update would give us the Hypothesis of Figure 5.19 with trace and distinguishing formula as described in Table 5.4.

Table 5.4: Trace and Distinguishing Formulæ of the Fifth Hypothesis

State:	1	3	4	5
Trace:	$\langle\ \rangle$	$\langle a\rangle$	$\langle ab\rangle$	$\langle a\rangle$
δ:	$\langle a\rangle\delta_3 \wedge \langle a\rangle\delta_5$	$\langle b\rangle\delta_4$	$\langle c\rangle tt \wedge \langle a\rangle tt$	$\langle b\rangle[c]ff$

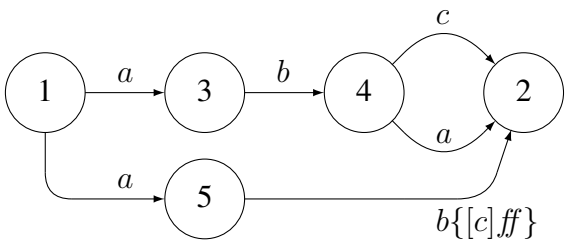


Figure 5.19: The Fifth Hypothesis

33) **Equivalent**(hyp) = $\langle a\rangle\langle b\rangle([c]ff \wedge \langle a\rangle tt)$

34) This leads to line 39 of **Interpret**, **Which**$(a, pt, \langle\ \rangle\alpha^*_{pt}\rangle\rangle, cce)$

35) In the **Which** subroutine:

35a) For both a-transitions to states with distinguishing formulæδ_3 and δ_5, and with $\varphi = \langle b \rangle \left(([c]ff \wedge \langle a \rangle tt) \wedge \emptyset \right)$

35b) Test **Membership**$(\langle\ \rangle(\delta_1 \wedge \langle a \rangle(\delta_3 \wedge \varphi)))$

35c) Test **Membership**$(\langle\ \rangle(\delta_1 \wedge \langle a \rangle(\delta_5 \wedge \varphi)))$

35d) The respective answers are No and Yes. Therefore we pick the path to the δ_5 state since there is only one consistent transition.

36) This gives us a call to **Interpret**$(\langle a \rangle$,⑤$, \varphi)$

37) This leads to line 28 of **Interpret** where we pick $\langle a \rangle tt$ from among $([c]ff \wedge \langle a \rangle tt)$ to get a call to *SIMPLE-ERROR*$(\langle a \rangle$,⑤$, b, a)$

38) This simple update adds the $\langle a \rangle$ action to state⑥, and adds the tentative distinguishing formula $[c]ff$ to state⑥.

39) This update would give us the Hypothesis of Figure 5.20 with trace and distinguishing formula as described in Table 5.5.

Table 5.5: Trace and Distinguishing Formulæ of the Sixth Hypothesis

State:	1	3	4	5	6
Trace:	$\langle\ \rangle$	$\langle a \rangle$	$\langle ab \rangle$	$\langle a \rangle$	$\langle ab \rangle$
δ:	$\langle a \rangle \delta_3 \wedge \langle a \rangle \delta_5$	$\langle b \rangle \delta_4$	$\langle c \rangle tt \wedge \langle a \rangle tt$	$\langle b \rangle([c]ff \wedge \langle a \rangle)$	$[c]ff \wedge \langle a \rangle$

40) **Equivalent**(hyp) = $[a](\langle bc \rangle tt \vee \langle bd \rangle tt)$

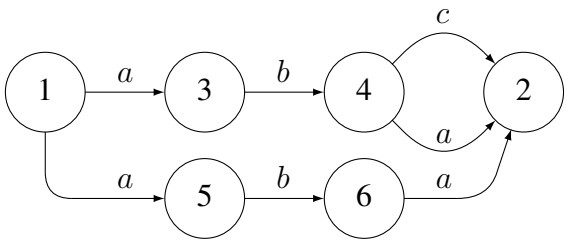


Figure 5.20: The Six Hypothesis

41) We end up on line 49 of **Interpret**, where for each a-transition to states③ and⑤ we consider whether the counter-example is consistent with the known behaviour down these states:

41a) Consider first state③:

41b) Use **Or-Elimination**($\langle\!\langle a_{③} \rangle\!\rangle$,③, $\langle bc\rangle tt \vee \langle bd\rangle tt$)

41c) Compute **Membership**($\langle a\rangle(\delta_3 \wedge \underbrace{\langle bc\rangle tt}_{\varphi_1})$) = YES

41d) Compute **Membership**($\langle a\rangle(\delta_3 \wedge \underbrace{\langle bd\rangle tt}_{\varphi_2})$) = NO

41e) For all queries returning YES consider the associated φ_i in this case just $\varphi_1 = \langle bc\rangle tt$

41f) Compute **Interpret**($\langle\!\langle a_{③} \rangle\!\rangle$,③, φ_1)

42g) We skip the next few steps as they are trivial to check. We will return *NO-ERROR*.

42h) Back to step 41a, we now consider state⑤:

42i) Use **Or-Elimination**($\langle\!\langle a_{⑤} \rangle\!\rangle$,⑤, $\langle bc\rangle tt \vee \langle bd\rangle tt$)

41j) Compute **Membership**($\langle a\rangle(\delta_5 \wedge \underbrace{\langle bc\rangle tt}_{\varphi_1})$) = YES

41k) Compute **Membership**$(\langle a\rangle(\delta_5 \wedge \underbrace{\langle bd\rangle tt}_{\varphi_2}))$ = NO

41l) For all queries returning YES consider the associated φ_i in this case just $\varphi_1 = \langle bc\rangle tt$

41m) Compute **Interpret**$(\langle\!\langle a_{⑤} \rangle\!\rangle, ⑤, \varphi_1)$

41n) This leads to considering **Interpret**$(\langle\!\langle ab_{⑥} \rangle\!\rangle, ⑥, \langle c\rangle)$

41o) This leads to line 20 of **Interpret** where we do the following test:

41p) Compute **Membership**$(\langle\!\langle ab_{⑥} \rangle\!\rangle(\langle c\rangle tt \wedge \delta_6))$ =
Membership$(\langle a\rangle(\underbrace{\langle b\rangle(\langle a\rangle tt \wedge [c]ff)}_{\delta_5} \wedge \langle b\rangle(\langle c\rangle tt \wedge \underbrace{\langle a\rangle \wedge [c]ff}_{\delta_6})))$ = NO

42q) Therefore δ_6 cannot do a c-action, and we return
COMPLEX-ERROR$(\langle\!\langle ab_{⑥} \rangle\!\rangle, ⑥, \langle c\rangle tt)$

43) We next call **Split**$(\langle\!\langle ab_{⑥} \rangle\!\rangle, ⑥, \langle c\rangle tt)$

43a) For the prefix $\langle a\rangle$, of $\langle ab\rangle$, to itself compute:

43b) **Membership**$(\langle a\rangle(\delta_5 \wedge (\langle b\rangle\delta_6 \wedge \langle bc\rangle tt)))$ = YES

43c) Fails first for i = 2

43d) Since EP begins with $\langle c\rangle$ create a new state with trace $\langle ab\rangle$ with a c-transition

44) This update would give us the Hypothesis of Figure 5.21 with trace and distinguishing formula as described in Table 5.6.

45) **Equivalent**(hyp) = $\langle abd\rangle tt$

46) This leads to a call to **Which**, with $\varphi = \langle bd\rangle tt$ to decide to whether to go to state③ or state⑤

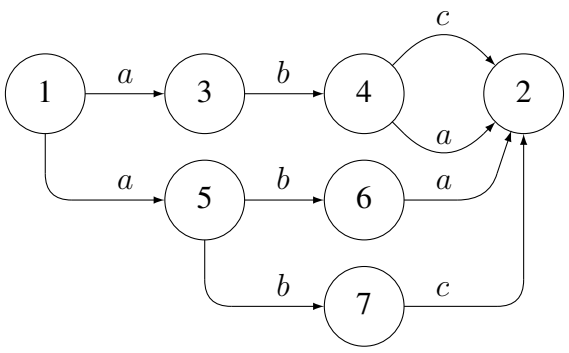


Figure 5.21: The Seventh Hypothesis

47) Compute **Membership**$(\langle\,\rangle(\delta_1 \wedge \langle a\rangle(\delta_3 \wedge \varphi)))$

48) Compute **Membership**$(\langle\,\rangle(\delta_1 \wedge \langle a\rangle(\delta_5 \wedge \varphi)))$

49) Since the answers are NO and NO we test **Membership**$(\langle\!\langle\,\rangle\!\rangle(\delta_1 \wedge \langle a\rangle(\varphi)))$ to see if we can add the new behaviour in place. Since the answer is Yes, we can add behaviour here.

50) We return *SIMPLE-ERROR*$(\langle\,\rangle, ①, a, \emptyset)$

51) This update would give us the Hypothesis of Figure 5.22 with trace and distinguishing formula as described in Table 5.7.

Table 5.6: Trace and Distinguishing Formulæ of the Seventh Hypothesis

State:	1	3	4	5
Trace:	$\langle\,\rangle$	$\langle a\rangle$	$\langle ab\rangle$	$\langle a\rangle$
δ:	$\langle a\rangle\delta_3 \wedge \langle a\rangle\delta_5$	$\langle b\rangle\delta_4$	$\langle c\rangle tt \wedge \langle a\rangle tt$	$\langle b\rangle([c]ff \wedge \langle a\rangle) \wedge \langle b\rangle\langle c\rangle tt$
State:	6	7	–	–
Trace:	$\langle ab\rangle$	$\langle ab\rangle$	–	–
δ:	$[c]ff \wedge \langle a\rangle$	$\langle c\rangle tt$	–	–

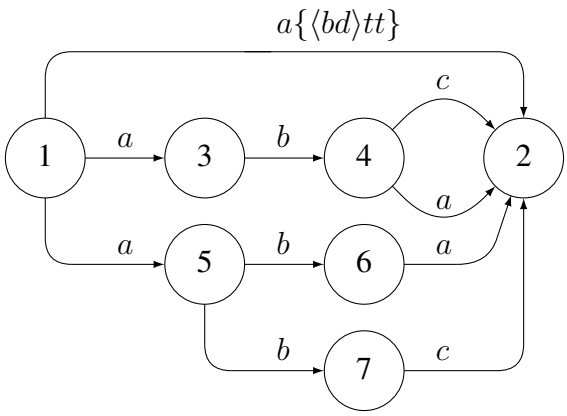


Figure 5.22: The Eighth Hypothesis

52) **Equivalent**(hyp) = $\langle abd\rangle tt$

53) This calls **Which** again, however since the counter-example has not changed we can deduce that the algorithm will find that only **Membership**($\langle a\rangle(\delta_2 \wedge \langle bd\rangle tt)$) returns the answer Yes

54) We treat this as if we were on line 28 of **Interpret**, since we only have one consistent transition.

55) We pick a $\langle\beta\rangle$ from among $\langle bd\rangle tt \wedge \emptyset$, in this case only $\langle b\rangle$

Table 5.7: Trace and Distinguishing Formulæ of the Eighth Hypothesis

State:	1	3	4	5
Trace:	$\langle\,\rangle$	$\langle a\rangle$	$\langle ab\rangle$	$\langle a\rangle$
δ:	$\langle a\rangle\delta_3 \wedge \langle a\rangle\delta_5 \wedge \langle a\rangle tt$	$\langle b\rangle\delta_4$	$\langle c\rangle tt \wedge \langle a\rangle tt$	$\langle b\rangle([c]ff \wedge \langle a\rangle) \wedge \langle b\rangle\langle c\rangle tt$
State:	6	7	–	–
Trace:	$\langle ab\rangle$	$\langle ab\rangle$	–	–
δ:	$[c]ff \wedge \langle a\rangle$	$\langle c\rangle tt$	–	–

56) Return *SIMPLE-ERROR*($\langle\,\rangle$,①, a, b)

57) This update would give us the Hypothesis of Figure 5.23 with trace and distinguishing formula as described in Table 5.8.

Table 5.8: Trace and Distinguishing Formulæ of the Ninth Hypothesis

State:	1	3	4	5
Trace:	$\langle\,\rangle$	$\langle a\rangle$	$\langle ab\rangle$	$\langle a\rangle$
δ:	$\langle a\rangle\delta_3 \wedge \langle a\rangle\delta_5 \wedge \langle ab\rangle tt$	$\langle b\rangle\delta_4$	$\langle c\rangle tt \wedge \langle a\rangle tt$	$\langle b\rangle([c]ff \wedge \langle a\rangle) \wedge \langle b\rangle\langle c\rangle tt$
State:	6	7	8	–
Trace:	$\langle ab\rangle$	$\langle ab\rangle$	$\langle a\rangle$	–
δ:	$[c]ff \wedge \langle a\rangle$	$\langle c\rangle tt$	$\langle b\rangle tt$	–

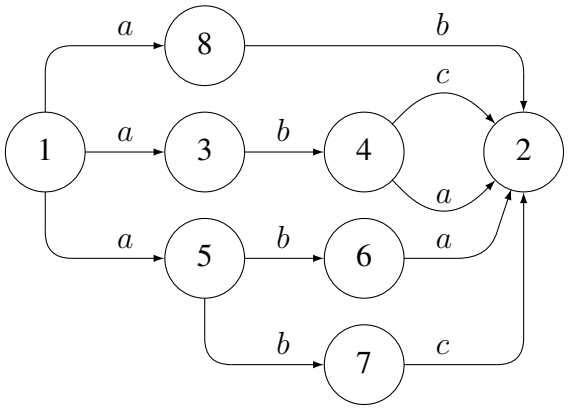


Figure 5.23: The Ninth Hypothesis

58) **Equivalent**(hyp) = $\langle abd\rangle tt$

59) It is trivial to check that this counter example will result in a *SIMPLE-ERROR*

60) This update would give us the Hypothesis of Figure 5.14 with trace and distinguishing formula as described in Table 5.9.

Table 5.9: Trace and Distinguishing Formulæ of the Tenth Hypothesis

State:	1	3	4	5
Trace:	$\langle\,\rangle$	$\langle a\rangle$	$\langle ab\rangle$	$\langle a\rangle$
δ:	$\langle a\rangle\delta_3 \wedge \langle a\rangle\delta_5 \wedge \langle abd\rangle tt$	$\langle b\rangle\delta_4$	$\langle c\rangle tt \wedge \langle a\rangle tt$	$\langle b\rangle([c]ff \wedge \langle a\rangle) \wedge \langle b\rangle\langle c\rangle tt$
State:	6	7	8	9
Trace:	$\langle ab\rangle$	$\langle ab\rangle$	$\langle a\rangle$	$\langle ab\rangle$
δ:	$[c]ff \wedge \langle a\rangle$	$\langle c\rangle tt$	$\langle bd\rangle tt$	$\langle d\rangle tt$

61) Equivalent(hyp) = YES

62) We can return our current hypothesis as the answer as it is equal to the target.

Chapter 6

Partial Results, Future Work and Conclusion

Having completed the algorithm for learning tree-like LTSs we are left to develop algorithms for the larger classes of LTSs, namely DAG LTSs and full LTSs. This chapter presents some partial work towards this goal.

6.1 Partial Results

In this section we present some unfinished work on developing algorithms for other classes of LTSs. We start with a simple method for determining where to place a transition. Say we have a partial hypothesis and we know it satisfies the correctly distinguishing property. Further suppose we discover a new transition; we are now tasked with finding what state that transition goes to. Here is how we do it.

If $\langle \alpha_p^* \rangle$ is the trace to the state where we want to add a β-transition, then for each state i in the hypothesis test the following:

$$\text{membership}(\langle \alpha_p^* \rangle \, (\delta_p \wedge \langle \beta \rangle \delta_i)) \tag{6.1}$$

Where δ_j denotes the distinguishing formula for j. For every state i such that the membership query returns YES, we add that transition. The only catch is that, as behaviour at state i is updated we need to re-evaluate these transitions. Unlike the case of tree learning, where all the transitions we put in were known to be actual transitions of the target, transitions placed this way are purely hypothetical. We can separate such hypotheses into two halves: *anchored* states and transitions, and *unanchored* states. By anchoring a state

we mean that the hypothesis knows of a trace to some already known state, usually the dead state. Anything that is anchored represents a known aspect of the target, and serves as scaffolding to hold the unanchored parts. This is the advantage learning languages has over learning LTSs. We can anchor all states to accepting states; here we have to try and rig it so that we can anchor states to the dead state. To expand on this work let us consider the case of learning DAG-LTSs.

6.1.1 Learning DAG LTSs

By DAG-LTS we mean directed acyclic graphs which are also LTSs. Learning these LTSs have one advantage over trees: distinguishing formulæ fully distinguish states. Thus the issue of Figure 5.3 is no longer relevant. This does not mean we no longer need total traces, since now states have multiple access traces. Moreover, we will need to make one concession: *that counter-examples will need to exhibit actions to the dead state*. Or, rather simple-errors are made clear by anchoring them to the dead state, while complex-errors are made clear until the place where they occur —the irreconcilable difference. For now, this ensures we will only have anchored states.

We need this concession because if we come to a simple disagreement about where a state should do an $\langle aa\rangle tt$ action, but the state has no a-transition, and further, we have states with distinguishing formula $\langle a\rangle[-]tt$ and $\langle ab\rangle[-]tt$, then which state should we send the transition to? What if it created a loop? Equation 6.1 shows how we can achieve this, but it creates these hypothetical unanchored transitions. And while it might seem fine for a hypothesis to have hypothetical transitions when we know the targets are DAGs, we will make the aforementioned assumption about the power of counter-examples for now, so that we do not have to bookkeep which transitions are anchored or not. By *anchoring*

simple errors to the dead state, we know they form a sub-LTS of the target.

For DAG-LTSs, as we did with tree LTSs, we will first consider deterministic DAGs. Like we saw for tree-LTSs, HML formulæ involving only $\wedge$ and $\langle\ \rangle$ operators are sufficient to describe deterministic systems. This is not surprising, since strings describe DFAs. We again must convince ourselves that we can transform formula containing $[\ ]$ and $\vee$ connectives into $\langle\ \rangle$ and $\wedge$ connectives where it is safe to do so, —when the transitions are deterministic. Yet, we do not want to change them in a way that would destroy their expressiveness in the later case —when the transitions are not deterministic. In other words, we want to design the first part of the algorithm that will learn Deterministic DAGs in a way that can be updated to learn non-deterministic ones. First, we show that we can effectively get rid of $[\ ]$ and $\vee$ connectives. For deterministic DAG-LTSs we can achieve this with three kinds of replacements:

1) **Formula or sub-formula ending in** $[\cdot]ff$**.** Since there are no cases of non-determinism, it will never be the case that the oracles suggest that both the formulæ $\langle\alpha^*\rangle\phi$ and $\langle\alpha^*\rangle\neg\phi$ are possible, since by determinism that would suggest that there exists a state with trace $\langle\alpha^*\rangle$ that satisfies $\phi \wedge \neg\phi$. Since the hypothesis will only add behaviours that counter-examples express, if we have an action β at some state, it is because the target satisfies $\langle\alpha^*\rangle\langle\beta\rangle tt$ for a trace to that state. Therefore, it is not possible that the target satisfies $\langle\alpha^*\rangle\neg\langle\beta\rangle tt = \langle\alpha^*\rangle[\beta]ff$. However, since we are requiring that counter-examples which illuminate simple errors anchor these errors to the dead state, we expect every counter-example to contain the sub-formula $[-]ff$. The point, however, is that this will not be a point of error. Simple errors can always be expressed using $\langle\ \rangle$ operators, and in fact must, since states can vacuously satisfy

$[\alpha]$-operators by having no α transitions. Thus we can exclude considering all formulæ ending in $[\cdot]ff$ where $[\cdot]ff$ represents the error (Think of formulæ built only with $\wedge$, $\langle\ \rangle$, and $\neg$).

2) **Formula containing [ ] operators.** If we get a counter-example $\delta_1[\alpha]\delta_2$ it is clear that if the target is deterministic, then it would also satisfy $\delta_1\langle\alpha\rangle\delta_2$. Even if the target were not deterministic, but we had been interpreting the counter-example and reached a state p such that we found $p \models [\alpha]\delta_2$ but p only had one α-transition, then it would be sufficient to adjust the hypothesis so that $p \models \langle\alpha\rangle\delta_2$.

3) **Formula containing $\bigvee \varphi_i$.** As discussed earlier we can use a procedure similar to **OR-Elimination** to change $\bigvee_{i\in I} \varphi_i$ into $\bigwedge_{\imath\in T\subset I} \varphi_i$ for deterministic targets.

Once we believe we can effectively consider formulæ built only using $\wedge$, $\langle\ \rangle$, and tt, it is not hard to believe that we can parse any counter-example as $\langle\alpha^*\rangle\langle\beta\rangle\delta$. Where $\langle\alpha^*\rangle$ is a trace to a unique state admitting a simple-error $\langle\beta\rangle$ leading to a state which statisfies δ. Furthermore, δ contains the sub-formula $[-]ff$, as it highlights behaviour up to the dead state.

We continue with some relevant definitions. First, a new feature of DAG LTSs is that, unlike trees, paths converge. The same state can have different traces. We give a special name to locations where paths join together:

Definition 6.1.1 (Join)**.** *If we have two distinct states x and y transitioning to the same state z, we say these states form a join. Specifically x and y* join *z. We call it a non-deterministic join if the transitions from x and y have the same label. Otherwise we call it a deterministic join.*

We are going to modify the definition of what it means when states are correctly-distinguished to take into account the non-deterministic joins. The problem is, if state x and y join z non-deterministically on an α-transition, then both $x, y \models \langle\alpha\rangle\delta_z$.

Definition 6.1.2 (Correctly Distinguishing Property)**.** *We add to Definition 5.2.1 the following condition:*

4) If states x and y non-deterministically join z, then δ_x and δ_y are correctly distinguished.

This new condition is easy to maintain. Since both $x, y \models \langle\alpha\rangle\delta_z$, the fact that we require δ_x and δ_y to be distinguished simply means all the other actions descendant from x and y are known to be different.

Now, since we are dealing with Deterministic DAG-LTSs for the moment, we only expect to get *SIMPLE-ERRORS*. We can parse these as:

$$\langle\alpha_p^*\rangle\langle\beta\rangle\delta$$

where:

- $\langle\alpha_p^*\rangle$ is a trace to a state
- $\langle\beta\rangle$ is the missing actions
- δ is the trace to the dead state

By convention all traces to the dead state will end in $[-]ff$, thus $\not\delta)_n = [-]ff$. It would seem once we compute this parsing we need only attach state p, by β-transition, to the unique state that satisfies δ. Ignoring for the moment how to find the state that satisfies δ

we consider another problem. If states x and y join an ancestor, z, of the state p where we are adding the new action, it might be the case that one of x and y should transition to state z. The new β action might not occur at a descendant state of the target for both states x and y. For this we introduce the ***Deterministic-Split***.

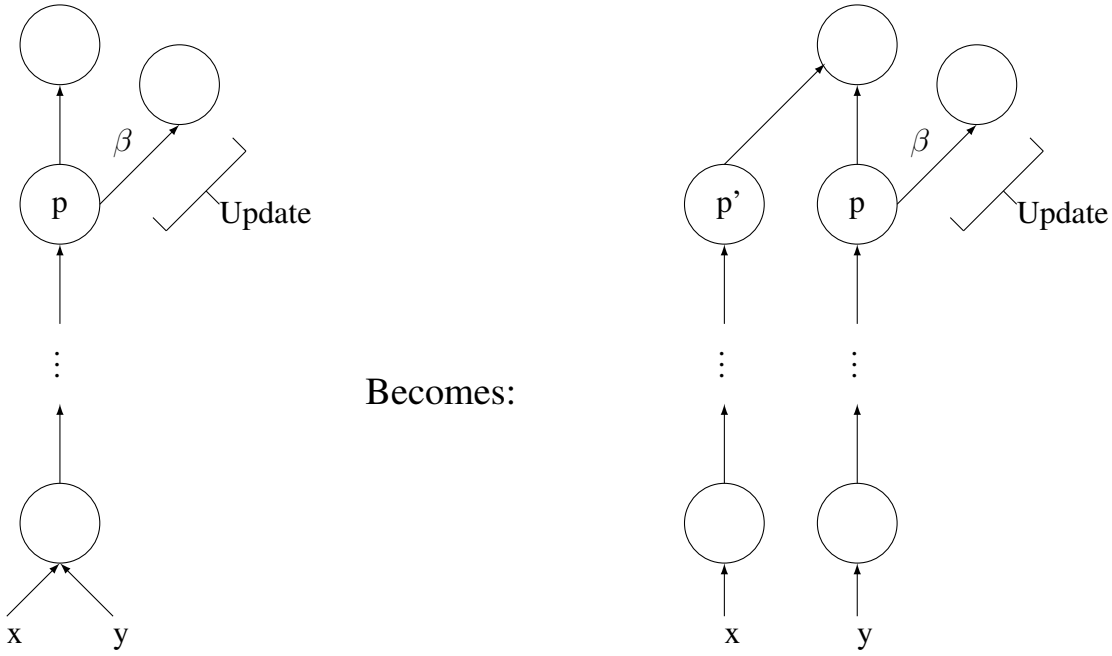


Figure 6.1: **Deterministic-Split**

Definition 6.1.3 (**Deterministic-Split**). *Say we have states x and y, with traces* $\langle \alpha_x^* \rangle$ *and* $\langle \alpha_y^* \rangle$. *Say x and y join state z by an a-transition and b-transition respectively. Let* $\langle \alpha_{z-p}^* \rangle$ *represent the trace from state z to a state p, where we have just added a new* β*-transition. A deterministic split tests the following:*

membership($\langle \alpha_x^* \rangle \langle a \rangle \left(\delta_z \wedge \langle \alpha_{z-p}^* \rangle \langle \beta \rangle tt \right)$ *)*

membership($\langle \alpha_y^* \rangle \langle b \rangle \left(\delta_z \wedge \langle \alpha_{z-p}^* \rangle \langle \beta \rangle tt \right)$*)*

For all states not returning 'YES' on the above queries, we must create a path of 'mirror states' for all the states along the trace $\langle \alpha^*_{z-p} \rangle$, to redirect the 'NO' state to. If we call that last mirrored state (mirroring p) p', then p and p' join at all the original successor states of p. See Figure 6.1.

This definition is easy to expand to a collection of states joining a single state z, just consider it as two collections of transitions: one collection which answers NO on the membership query and one collection wich answers YES. These are the only two answers, so we know we can divide all the transitions joining z along these lines. We use this to justify considering only joins of two transitions henceforth. We show this operation in Figure 6.1. The added state might force the path from a join to the addition to be split into two paths —which then rejoin afterwards. The figure does not show the case where both x and y are sent down the new mirroring path and we create a new trace to accept the new β-behaviour. Note, p cannot be the dead state since simple errors are expressed to the dead state; thus the above definition is consistent. When we do a **Deterministic-Split** the distinguishing formula of either x or y, (without loss of generality say x) changes from the old δ_x to include that it can now do β actions. The other one adds a $[\beta]ff$ sub-formula. The $[\beta]ff$ sub-formula exists only to deny that the current state never eventually travels along some known branch.

Definition 6.1.4 (clones)**.** *When we deterministically split states, we note that they were once the same by calling them a* cloned *pair.*

For instance in Figure 6.2 we represent the cloned pair by connecting them with a dashed line. This says these states were once the same, but were split because they disagreed on some added branch.

We now need to prove that **Deterministic-Split** maintains the new Correctly-Distinguishing Property.

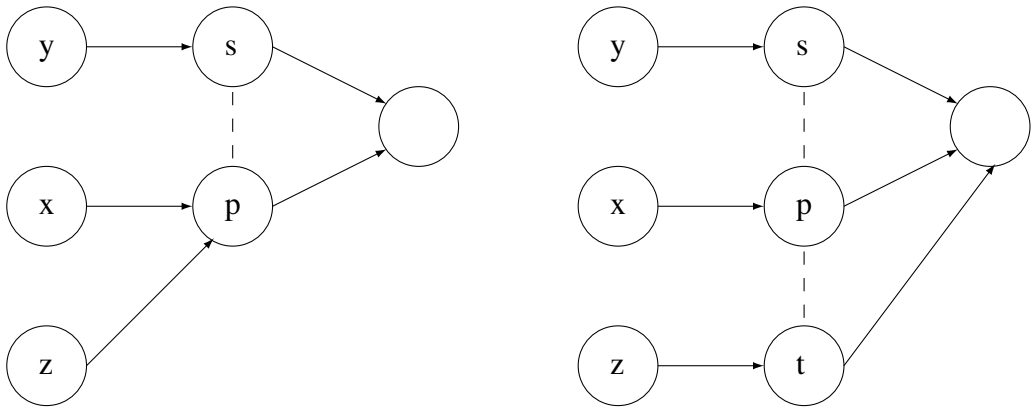


Figure 6.2: Deterministic Split Maintains Distinguishing Formulæ

Let us break down a sketch of the proof: Adding a change requires us to do a **Deterministic-Split**. For this we travel backwards to all ancestor states; and, for any joins we find we call **Deterministic-Split** to see if they need to be unjoined. Call the change β. Say it is followed by actions δ, and say that the changed state originally had the distinguishing formula δ', then the deterministic split creates two branches which induce the following changes on δ': A *positive change* $\delta' \rightarrow \delta' \wedge \langle\beta\rangle\delta$ and a *negative change*: $\delta' \rightarrow \delta' \wedge [\beta]\neg\delta$. This is important as earlier we had said that we need only $\wedge$ and $\langle\ \rangle$ operators to describe the DAG-LTS. However we are using $[\ ]$ in a limited way, to deny the existence of a branch in a mirroring formula. We now want to prove that **Deterministic-Split** maintains the Correctly-Distinguishing property.

Lemma 6.1.5. ***Deterministic-Split*** *maintains the new Correctly-Distinguishing Property*

Proof. Consider Figure 6.2. Let us say we have states s and p. Furthermore they have

already been split once so that they now have distinguishing formula:

$$\begin{aligned} \delta_s &= \delta \wedge \langle\alpha\rangle\delta' \\ \delta_p &= \delta \wedge [\alpha]\neg\delta' \end{aligned} \tag{6.2}$$

They already satisfy the Correctly-Distinguishing Property, as will all single splits. The question is then, will it still be satisfied if we split again at the same location? Let us consider what happens if we split state p to get a state t, as seen in Figure 6.2. Assuming state p accepts the positive change on some action β we would get two new distinguishing formulæ:

$$\begin{aligned} \delta_p &= \delta \wedge ([\alpha]\neg\delta') \wedge \langle\beta\rangle\delta'') \\ \delta_t &= \delta \wedge ([\alpha]\neg\delta') \wedge ([\beta]\neg\delta'') \end{aligned} \tag{6.3}$$

By inspection the property is still maintained.

□

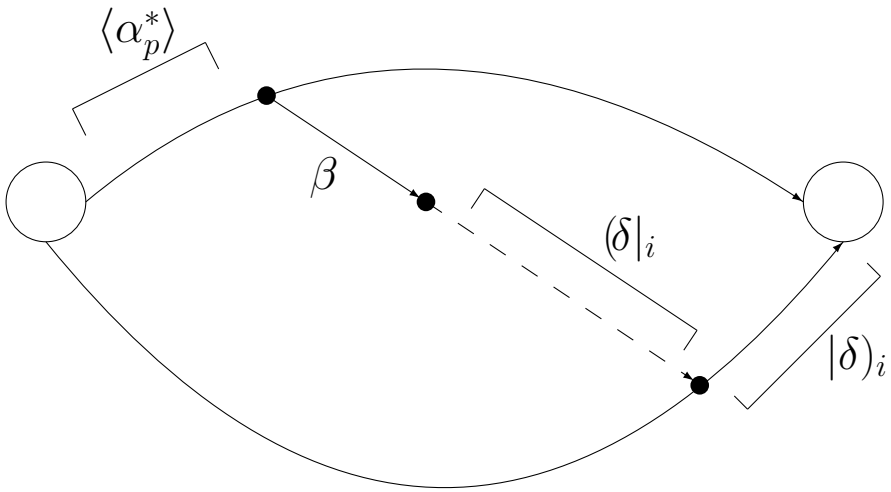


Figure 6.3: Adding a Link

Let us move on to the idea of linking. Once we parse an equation as $\langle\alpha^*\rangle\langle\beta\rangle\delta$, the idea is to connect, by actions δ, the state p that the trace $\langle\alpha^*\rangle$ ends at, to a state that satisfies δ. However because of the **Deterministic-Split** routine, there may be many formulæ which satisfy δ —or there could be none. Here is where our assumption that for simple errors δ illuminates actions to the dead state simplifies things. We can compute the best place for β to transition to by working backwards from the dead state comparing our path we trace in reverse to the formula δ. This was why we needed to adjust our definition of the Correctly Distinguishing Property. The determinism property is only for following transitions forwards. There is nothing stopping a join from containing the same labels. For this we need the idea of a **Deterministic-Which** routine —for when we work backwards.

Given a **Deterministic-Which** subroutine, Figure 6.3 suggests what we will do. We parse the distinguishing formula as:$\langle\alpha^*\rangle\langle\beta\rangle(\delta|_i|\delta)_i$, for some fixed i. Thus $|\delta)_i$ represents the part of δ reachable in reverse from the dead state, and $(\delta|_i$ represents a set of new states that need to be added to create the *link*.

Definition 6.1.6 (**Deterministic-Which**). *Parse the counter-example as $\langle\alpha_p^*\rangle\langle\beta\rangle(\delta|_{i-1}\alpha|\delta)_i$, for every possible i. If when computing a link we have worked backwards through the trace $|\delta)_j$, and we we find a non-deterministic join on the character α joining into a state x, then for all states $y_k \xrightarrow{\alpha} x$ we test:*

$$\langle\alpha_p^*\rangle\langle\beta\rangle\left((\delta|_{i-1}\left(\delta_{y_k} \wedge \langle\alpha\rangle\delta_x\right)\right) \tag{6.4}$$

Note that we know $x \models \delta_x \wedge |\delta)_i$ and for $\delta = (\delta|_{i-1}\alpha|\delta)_i$ if the answer to the query is not YES, we link at the current point in question; otherwise, we continue backwards on any states that returned YES in the above query.

Lemma 6.1.7. ***Deterministic-Which*** *correctly determines where a link should be put.*

Proof. Given that all the hypothesis will be deterministic, Equation 6.4 is sufficient to determine which path to take because the actions $\langle\alpha_p^*\rangle\langle\beta\rangle(\!|\delta|\!)_i$ only only lead to one state in the deterministic target. So whenever we are backtracking from the dead state, and are faced with a choice, if the path we have followed backwards traces $|\!\delta\!)_{i-1}$ in reverse, and that leads to multiple states, we know that $\langle\alpha_p^*\rangle\langle\beta\rangle(\!|\delta|\!)_i$ must lead to only one of those multiple states. Thus for any y_k only one of Equations 6.4 can be satisfied. □

Lemma 6.1.8. *Linking maintains the Correctly-Distinguishing Property*

Proof. Again, parse the counter-example as $\langle\alpha_p^*\rangle\langle\beta\rangle(\!|\delta|_i|\!\delta\!)_i$. First note, we reverse backwards through $|\!\delta\!)_i$ until we can go no further. When we stop, we create the linking bridge between the β action and the stopping point, which we will call state x. We know that the i^{th} character of δ, the one joining at state x, cannot create a non-deterministic join —otherwise we would not have stopped. Thus we do not need to worry about breaking the Correctly-Distinguishing Property. As Figure 6.4 shows, we might be concerned that on a deterministic join the two joining states could satisfy the same behaviours after action γ, but this would imply that δ_x and δ_y are not correctly distinguished.

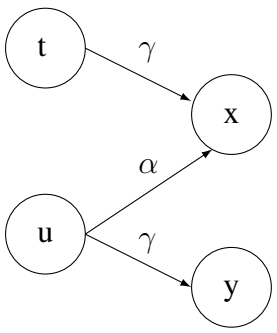


Figure 6.4: Linking Maintains Distinguishing Property

□

6.1.2 Learning Deterministic DAG-LTS Algorithm

We are now ready to consider the algorithm for Learning DAG-LTSs. We present the algorithm in less detail then the previous algorithm. Also, it is tailored to the deterministic case. We then present a less detailed proof of correctness.

Algorithm 6.1.9. Learn_DAG_LTS:

—Initialize Hypothesis as empty LTS

—While **Equivalent**(Hypothesis) = NO

— Else let δ be the counter-example

— parse $\delta = \langle \alpha_p^* \rangle \langle \beta \rangle \delta'$

— Use **Deterministic-Which** to find unique state x for linking

— parse $\delta' = (\delta'|_i | \delta')_i$ where $|\delta')_i$ is the trace from x to the dead state

— create states $(\delta'|_i$, for all j, $0 \leq j \leq i$, where we '*link*' state p to state x via state $(\delta'|_0 \ldots (\delta'|_i$.

— Use **Deterministic-Split** on all ancestor '*joins*' of state p

— End While

— return Hypothesis

To facilitate a Proof of Correctness of the Algorithm we must again consider what it means to be an effective sub-DAG-LTS of the target. In the deterministic case this definition is very simple:

Definition 6.1.10 (Effective Sub-DAG LTS)**.** *An effective sub-DAG LTS of a target LTS is any DAG-LTS that is the result of the following process:*

- *We arbitrarily pick a state to remove*
- *We remove every state and transition that follows the removed state until we reach a join*
- *We delete every state and transition that preceded the removed state, that can no longer reach the dead state.*
- *We repeat the above three steps for an arbitrary number of repetitions*
- *When this process is complete we merge any states which appear bisimilar, until we reach a minimal DAG-LTS.*

With this idea of sub-DAG we are now ready to prove correctness.

Theorem 6.1.11. *Each hypothesis is a sub-DAG LTS of the target.*

Proof. The first hypothesis we create is the empty LTS. Either this is correct or we get our first counter-example. In this counter-example, by our presupposition, we must be able to find a trace to the dead state. Thus in our parsing of the counter-example we find $\langle \alpha_p^* \rangle = \epsilon$, leaving us with our first action $\langle \beta \rangle$ and the remainder of the trace to the dead state δ'. If $\langle \beta \rangle = \epsilon$, then the start state equals the dead state, and we build an empty trace, otherwise we build a normal trace. The trace is necessarily a trace in the target, as the target has no cycles. Thus the first hypothesis is a sub-DAG LTS of the target.

Given a hypothesis which is a sub-DAG LTS of the target, which also adheres to the Correctly-Distinguishing Property, and a counter-example parsed as $\langle \alpha_p^* \rangle \langle \beta \rangle \delta'$, we find that since the target is deterministic, $\langle \alpha_p^* \rangle$ represents a unique state; since the counter-example says the target satisfies $\langle \alpha_p^* \rangle \langle \beta \rangle \delta'$, this unique state must have a β-action. That β action goes to a state that can satisfy δ'. By working backwards from the dead state, we can find a set of states that satisfy each suffix of δ' (for $\lfloor \delta' \rfloor_n$ the dead state is in this set). There must exist a last non-empty set containing states which satisfy the increasingly longer suffixes, $\lfloor \delta' \rfloor_i$ (which may be $\lfloor \delta' \rfloor_0$). The path we are looking for must pass through at least one of the states in this last non-empty set, because otherwise when we created the linking states to match the corresponding prefix as shown in Figure 6.3, we would be creating states bisimilar to other states known to exist. We also know it must pass through at most one, since the target is deterministic. By Lemma 6.1.7 we know we can find this unique state, by Lemma 6.1.8 we know this operation maintains the Correctly-D distinguishing Property.

We then call **Deterministic-Split** to see if all the joins that are ancestors of the added action β are still forming correct joins. For each ancestor join, with state x and y joining state z on transitions α and γ respectively, we call **Deterministic-Split**. Let $\langle \alpha_{z-p}^* \rangle$ be the trace from the ancestor join at state z to the changed state p. Without loss of generality, if the membership query says the target cannot do actions $\langle \alpha_x^* \rangle \langle \alpha_{z-p}^* \rangle \beta \delta'$ but can do actions $\langle \alpha_y^* \rangle \langle \alpha_{z-p}^* \rangle \beta \delta'$ (one of them must be possible) then since we also know that the target can do some actions $\langle \alpha_x^* \rangle \langle \alpha_{z-p}^* \rangle \beta \delta''$, where δ'' is also a trace to the dead state, we know that there is a mirrored set of states $\langle \alpha_{z-p}^* \rangle$ that are distinguished by δ' and $\neg \delta'$. None of these new states are bisimilar to any other states, as the states they were split from were not. However, these two paths to the best of our knowledge still agree after the changed state.

So we keep all successor states the same. Thus whenever the changed state transitions on a character to another state, so transitions its clone. These paths are in the target since the descendants of x passed through these states. Thus this new hypothesis is still a sub-DAG of the target and by Lemma 6.1.5 we know the new hypothesis maintains the Correctly-Distinguishing Property.

□

6.2 Non Deterministic DAG-LTS

Because of the assumption we made about the power of the counter-examples, we can harness their power to easily extend new non-deterministic branches. Think of it as a delayed linkage. We see this idea in Figure 6.5. Here we have used a tentative distinguishing formula to markup the newly learned split location; assuming that the dead state is indeed not the state it should transition to, we will eventually get a counter-example expressing as much. By assumption this counter-example will express behaviour to the dead state because it is a counter-example elucidating a simple error. We must get a simple error because the fact is that the state $1'$ of Figure 6.5 leads to a state that can at least do some actions.

As a final note, it seems like it should be easy to remove the restriction on the efficacy of the simple error counter-examples (expressing actions to the dead state) if we marked known anchor states, and re-evaluated all transitions leaving unanchored states using the idea of Equation 6.1.

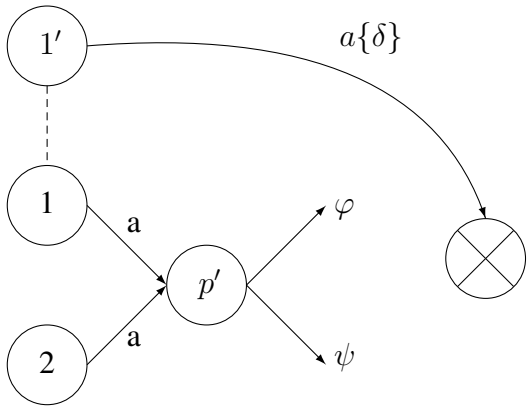


Figure 6.5: How We Use Non-Deterministic Split

6.3 Future Work

Certainly this thesis suggests much more work to be done. Here is a partial list of possible topics:

1) Explore how we can learn LTSs based on other notions of equivalence. By this we refer to trace and failure equivalence. It is known that there exist subsets of HML formulæ that are sufficient to describe trace and failure equivalence. One might wonder what kind of targets we would learn if we restricted our counter-examples only to those types. How then would we deal with membership queries? Only allow queries that showed trace or failure behaviour?

2) Explore canonical forms for LTSs. This idea is based on the idea that, for Angluin's algorithm, the accepting states allowed us to get access to strings exhibiting long-term behaviour —behaviour that passed through cycles. We can fix LTSs so that we only consider LTSs with the dead state. Furthermore, if we add the requirement that

every state must exhibit all behaviours, we can in fact guarantee that all interesting LTSs have the dead state. This is because a subsection of an LTS that can do all actions infinitely often would collapse under bisimilarity into a single state that self-looped on all actions. Otherwise it has states that can eventually enter a dead state. We can transform LTSs into this canonical form fairly easily. Create a new dead state. For any state that cannot do an α action, send an α-transition to that state. For instance, the old dead state has a transition on all characters to this new dead state. Could this procedure transform two non-bisimilar LTSs into the same LTS? Does this help us learn the LTS?

3) Create a learning algorithm that uses μ-calculus queries to learn LTS. Thus if when placing a transition that would form a cycle starting at state p with the trace $\langle abaad\rangle tt$, we can do a membership query from state p on the equation $\nu Z.\langle abaad\rangle tt \wedge [-]Z$. If YES, the cycle should be formed; otherwise we query the following until we get the answer YES:

   1) membership($\langle \alpha_p^* \rangle \langle abaad\rangle tt$)

   2) membership($\langle \alpha_p^* \rangle \langle abaad\rangle \langle abaad\rangle tt$)

   ⋮)

   k) membership($\langle \alpha_p^* \rangle \langle (abaad)^k\rangle tt$)

   Since our running time is only judged on the number of states in the target, and further, we know that the series of actions "abaad" eventually runs out, we are safe in making all those queries because the number of such queries is on the same order as the size of the target.

4) Let us further consider the idea that the problem with HML formula is that they can only express behaviour up until some unknown state, not long term behaviour to a well defined accepting state. There is a version of Angluin's algorithm that works in what is called the no reset model [12]. In this model, every time we do a membership query, the next query begins in the state the previous one ended in. This is in line with the idea of an oracle being a black box version of a DFA where we can feed in characters and the only information we get back is whether, after the current character, we are in an accepting state or not. The reason this seems similar to learning LTS is that it encounters the same problem of not knowing exactly when it is stuck in a cycle. Thus it is worthwhile to explore the similarities further.

5) In Chapter 5 we considered learning tree-like transition systems. We should note the similar problem of learning regular tree languages. This is an area with a bit of research, including that done by Drews and Högberg [6]. Tree Automata accept tree languages, and in some sense one could think of transition systems as accepting the modal logics they satisfy. The parallels are made more interesting when one considers that tree like transitions systems can be fully modeled by tree like modal logics. However, without further research the two models appear quite different. For instance, tree languages generalize strings in a way that makes the characters the states of a tree (think branching strings). In transition systems, the characters are labeling the transitions. We might imagine that there is a way to transform between the two instances; suppose we could encode all the transitions leaving a state as a single character which represents the set of labeled transitions leaving that state. Although it would bring the two models closer, this would exponentially increase

the size of the character alphabet.

The set of all subsets of characters is exponentially larger in size then the list of characters. In fact, as best we can tell the two notions are not very similar, and without further research it is not clear how close this two problems are, but consider this: learning DFA was based on the notion of the famous Myhill Nerode Theorem; a version of this theorem caries over to tree automata. The fact that this theorem caries over to tree automata is the key to learning regular tree languages. The Myhill Nerode Theorem seems not to fully carry over to LTSs. Ultimately, any similarities between the two models is grounds for future research, as it stands they appear to require different approaches.

6) It would be fascinating to try and cast traditional algorithms under the model of learning. By this we mean, how would we state a classic algorithm like graph colourability as a learning algorithm. Traversing an edge could be considered like a membership query telling us this edge existed. The difference being that this membership query is only providing information about O(1) amount of the size of the target. Whereas strings provided O(n) amount of information about target DFAs of size O(n^2). There already is a lot of work about what happens to problems in certain complexity classes when they are provided oracles of different powers.

   Consider further an adversary trying to fool a learning algorithm making queries with information O(n) in the size of the target. That is, the adversary answers queries in a way as to maximize the number of queries the algorithm must make. It does so by changing the target, but keeps it consistent with all previous queries. Can such an adversary force any learning algorithm to make an exponential number of queries?

Certainly any learning algorithm could try to take a tour of the input to make sure the adversary is not changing it later to force greater computation, but then the learning algorithm would seemingly need to remember an exponential amount of information at any given time (by pre-computing the problem).

- Can our idea of looking at learning as the reverse of taking subsets of some target be used to inspire other learning algorithms? Does it have other practical applications, or can it be replaced by something theoretically simpler.

6.4 Conclusion

The goal of the thesis was twofold: we wanted to see if we could extend Angluin's algorithm with HML formulæ to learn a generalization of DFA. Secondly, we wanted to use the search for such an algorithm as an exercise in examining the theoretical importance of learning theory, and explore a bit of its history. We fell short of accomplishing the first goal, however we were able to use HML formula to learn a non-deterministic target. This is a step forward. We also understand very well the obstacles that need to be overcome to extend this result to more complicated LTS. Breaking the class of LTSs into the three increasingly complex components also taught us something about HML formulæ. There is a clear division in which formulæ are necessary for describing non-deterministic choice, and those that are sufficient to describe deterministic choice. Moreover this division is fairly simple. We would group $\langle\ \rangle$ and $\wedge$ as being associated with determinism, and [ ] and $\vee$ as being associated with non-determinism. We also devised a method of proof for the learning algorithm that associated the loss of information from taking subsets of the target, and showed how it could be offset by a net gain in information through membership

queries. As mentioned in the future work section, perhaps this notion could be formalized. There seems to be great potential in measuring how perturbations to the target, or input to an algorithm could affect the computation of the associated algorithm. Does the learning model help us understand this at all? If we could make one prescription for the future, it would seem that information guides computation, and computer science is really information science.

Bibliography

[1] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[2] Dana Angluin. Computational learning theory: survey and selected bibliography. In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 351–369, New York, NY, USA, 1992. ACM Press.

[3] Dana Angluin and Michael Kharitonov. When won't membership queries help? In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 444–454, New York, NY, USA, 1991. ACM Press.

[4] Anselm Blumer, A. Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Learnability and the vapnik-chervonenkis dimension. *J. ACM*, 36(4):929–965, 1989.

[5] Julian Charles Bradfield. *Verifying temporal properties of systems*. Birkhauser Boston Inc., Cambridge, MA, USA, 1992.

[6] Frank Drewes and Johanna Hogberg. Query learning of regular tree languages: How to avoid dead states. *Theor. Comp. Sys.*, 40(2):163–185, 2007.

[7] Jean-Claude Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2–3):219–236, May 1990.

[8] Dimitra Giannakopoulou, Jamieson Cobleigh, and Corina Pasareanu. Learning assumptions for compositional verification. Technical report, November 01 2002.

[9] E.M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37:302–320, 1978.

[10] Paris C. Kanellakis and Scott A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Inf. Comput*, 86(1):43–68, May 1990.

[11] Michael Kearns and Leslie Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *J. ACM*, 41(1):67–95, 1994.

[12] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. The MIT Press, second edition, 1997.

[13] Dexter C. Kozen. *Automata and Computability*. Springer-Verlag, first edition, 1997.

[14] Robin Milner. *Communication and Concurrency*. Prentice Hall, first edition, 1989.

[15] Paige and Tarjan. Three partition refinement algorithms. *SICOMP: SIAM Journal on Computing*, 16, 1987.

[16] Leonard Pitt and Leslie G. Valiant. Computational limitations on learning from examples. *J. ACM*, 35(4):965–984, 1988.

[17] Leonard Pitt and Manfred K. Warmuth. The minimum consistent dfa problem cannot be approximated within any polynomial. *J. ACM*, 40(1):95–142, 1993.

[18] Alban Ponse, Maarten de Rijke, and Yde Venema. *Modal Logic and Process Algebra: A Bisimulation Perspective*. CSLI Publications, first edition, 1995.

[19] Colin Stirling. Modal and temporal logics for processes. In *Banff Higher Order Workshop*, pages 149–237, 1995.

[20] L. G. Valiant. A theory of the learnable. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 436–445, New York, NY, USA, 1984. ACM Press.

[21] T. Yokomori. Learning non-deterministic finite automata from queries and counterexamples. pages 169–189, 1995.