FINITE AUTOMATA FOR VARIABLE ASSIGNMENT LANGUAGES AND THEIR ALGORITHMIC INFERENCE

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ATAKAN GARIPLER

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2023

Approval of the thesis:

FINITE AUTOMATA FOR VARIABLE ASSIGNMENT LANGUAGES AND THEIR ALGORITHMIC INFERENCE

submitted by **ATAKAN GARIPLER** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _____

Prof. Dr. Halit Oğuztüzün
Supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Assoc. Prof. Dr. Ebru Aydın Göl
Computer Engineering, METU _____

Prof. Dr. Halit Oğuztüzün
Computer Engineering, METU _____

Assoc. Prof. Dr. Hürevren Kılıç
Computer Engineering, Atılım University _____

Date:07.09.2023

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Atakan Garipler

Signature :

ABSTRACT

FINITE AUTOMATA FOR VARIABLE ASSIGNMENT LANGUAGES AND THEIR ALGORITHMIC INFERENCE

Garipler, Atakan
M.S., Department of Computer Engineering
Supervisor: Prof. Dr. Halit Oğuztüzün

September 2023, 52 pages

Finite automata for variable assignment languages (FA-VA), a subclass of deterministic finite automata, are defined. Properties of FA-VAs are analyzed. An active learning algorithm that infers FA-VA-based models of interactive constraint-based systems is devised. Advantages and limitations of the proposed representation method and the algorithm are discussed.

Keywords: Automata Theory, Active Automata Learning, Model Learning, Constraint Acquisition, Representation of Constraint-based Systems

ÖZ

DEĞİŞKEN ATAMA DİLLERİNE ÖZGÜ SONLU DURUM DEVİNGENLERİ VE BUNLARIN ALGORİTMİK ÇIKARSANMASI

Garipler, Atakan
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Halit Oğuztüzün

Eylül 2023 , 52 sayfa

Sonlu durum devingenlerinin bir alt sınıfı olan değişken atama dillerine özgü sonlu durum devingenleri tanımlanmıştır. Bu devingenlerin özellikleri incelenmiştir. Koşul temelli sistemlerin değişken atama dillerine özgü sonlu durum devingeni temelli modellerini çıkarsayan bir aktif devingen öğrenme algoritması geliştirilmiştir. Önerilen temsil metodu ve algoritmanın avantajları ve kısıtları tartışılmıştır.

Anahtar Kelimeler: Devingenler Teorisi, Aktif Devingen Öğrenme, Model Öğrenme, Koşul Edinimi, Koşul Temelli Sistemlerin Temsili

To the beloved memory of my grandmother

ACKNOWLEDGMENTS

I would like to thank my supervisor Prof. Dr. Halit Oğuztüzün for his support, guidance and for acquainting me with the field of active automata learning.

I also thank to Ege, our computer scientific discussions with whom helped me to develop some of the initial ideas eventually led this thesis; and to Rojda, our philosophical discussions with whom helped me to better conceive the implications of the representation method I proposed in this thesis.

Finally, I would like to express how grateful I am to my family for their unconditional love and support.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vi

ACKNOWLEDGMENTS viii

TABLE OF CONTENTS ix

LIST OF TABLES xi

LIST OF FIGURES xii

LIST OF ALGORITHMS xiii

LIST OF ABBREVIATIONS xiv

CHAPTERS

1 INTRODUCTION 1

1.1 Motivation and Problem Definition 1

1.2 The Outline of the Thesis 7

2 FINITE AUTOMATA FOR VARIABLE ASSIGNMENT LANGUAGES . . 9

2.1 Variable Assignment Languages 9

2.2 Finite Automata for Variable Assignment Languages 13

2.3 Fundamental Finite Automata for Variable Assignment Languages . . 22

3 ALGORITHMIC INFERENCE 33

3.1 Overview 33
3.1.1 Observation Tables 34
3.2 Inference Algorithm 35
3.2.1 LearnFAVA 36
4 DISCUSSION 45
REFERENCES 51

LIST OF TABLES

TABLES

Table 3.1 Observation Table for L_{simple} 35

Table 3.2 Observation Table for $L(SUL)$ 43

LIST OF FIGURES

FIGURES

Figure 2.1 M_1 28

Figure 2.2 M_2 29

Figure 3.1 M_{SUL} 44

LIST OF ALGORITHMS

ALGORITHMS

Algorithm 1 LearnFAVA - Baseline FA-VA Inference Algorithm 36

LIST OF ABBREVIATIONS

DFA	Deterministic Finite Automaton
FA-VA	Finite Automaton for Variable Assignment Languages
SUL	System Under Learning

CHAPTER 1

INTRODUCTION

1.1 Motivation and Problem Definition

This thesis focuses on designing a finite-automaton-like abstract machine that can compactly and comprehensively represent *interactive constraint-based systems*, and an algorithm that can learn those systems in terms of those abstract machines. More specifically, as a means of representing interactive constraint-based systems, this work introduces a new class of abstract machines, *finite automata for variable assignment languages (FA-VA)*, which is a is subclass of deterministic finite automata (DFA) [1]; moreover, so as to enable algorithmic inference of FA-VAs, it devises the algorithm LearnFAVA, which is a active automaton learning algorithm [2] that learns FA-VA-based representations of interactive constraint-based systems by actively making queries to the system under learning.

The concept of constraint can most easily be defined by defining "a set of constraints", which is a set of allowable valuations of a set of given variables [3]. Under this definition, an interactive constraint-based system is a system which operates on the basis of responding inputs given by the user *(interactive)* and behaviour of which is determined by a set of constraints *(constraint-based)*. In fact, an extended definition of constraint, where the time and order in which values of variables are assigned are also restricted (e.g. value of the variable x can only be changed right after the value of the variable y changed), can also be made (i.e. temporal constraints) [4]. However, in this work, the constraints with the classical definition given at the beginning of this paragraph are considered. This considered type of constraints can be also be referred as variable assignment constraints so as to prevent confusion (even so, in the

following chapters, the terms constraint and variable assignment constraint are used interchangeably).

Programmable signal generators are a good examples of interactive constraint-based systems. The main function of those devices is generating different electric signals according to the input taken from the user. Inputs are in the form of commands each of which define a property/feature of the signal to be generated. From a simplified perspective, a signal is defined by its waveform, frequency, and amplitude. Programmable signal generators have one command for each of those features. To illustrate, say the command SHAP sets the waveform, FREQ sets the frequency, and VOLT sets the amplitude. Then, so as to generate a square wave with 200 kHz frequency and 10 Volts amplitude, the required sequence of commands is "SHAP:Square, FREQ:200kHz, VOLT:10V" (or any other permutation of this sequence). Inputs may be given by the user one by one or may be read by the device from a file containing sequential inputs. Anyway, commands are applied/processed one by one. Nevertheless, some command combinations are erroneous/forbidden due to the limits of the capabilities of the specific signal generator. A simple example of such a limitation may be that the signal generator can generate a square wave with 500kHz frequency but cannot generate a triangular wave with 500kHz frequency. In this case, assume the user wants to get a square wave with 500kHz frequency and the current waveform is triangular. If the user gives the commands in the order of "FREQ:500kHz, SHAP:Square", the device will give an error since the application of the first command (FREQ:500kHz) results the signal being a triangular wave with 500kHz frequency, which is forbidden. However, if the commands are given in the order "SHAP:Square, FREQ:500kHz", no erroneous signal is formed. From an abstract perspective, commands can be considered as variables and the set of non-error-causing valuations of the set of commands (i.e. valuations defining the signals that are not forbidden) can be considered as the set of constraints. However, as the last example (about error behavior) shows, even if their behavior can simply be defined by only a set of variables and a set of constraints, signal generators (and general interactive constraint-based systems) show non-trivial state-based behavior.

Even though this thesis totally consists of theoretical work and it does not directly mention how to model signal generators, the initial concrete problem behind that

theoretical work is the problem of algorithmically inferring the behavior of signal generators [5]. Therefore, the signal generator analogy may help the reader, at least at some parts of the thesis.

In fact, behavior of signal generators [5] and general interactive constraint-based systems can be modeled by deterministic finite automata (under the condition that the domains of their variables are finite). The most simplest way of constructing the finite automata that models an interactive constraint-based system is first creating a state for each valuation of variables and then placing transition between pairs of states (i.e. valuations) which differ only in the value of a single variable. However, such a model would be huge, hard-to-trace and would hardly help to understand the system. Luckily, a minimal finite automaton representing the system can be learned by the methods of active automata learning [2], specifically, by L^* Algorithm [6].

As already underlined, the problem of inferring behavioral models of interactive constraint-based systems can be formulated as an active automata learning problem. Active automata learning is the general name for algorithmic methods of inferring behavior of a system in terms of automata, typically finite automata or register automata [7] (or their variants), by actively making experiments on the system (under the assumption that the system under learning can be modeled by that type of automata). Currently, there are several application areas of active automata learning such as constructing behavioral models of software systems for verification purposes, checking communication protocols for leaks, or modelling legacy software [2].

L^* Algorithm is the baseline active automata learning algorithm devised by Dana Angluin and presented at her seminal paper "Learning regular sets from queries and counterexamples" [6], which can be considered the starting point of the field of active automata learning. The main approach of the L^* algorithm is based on constructing a candidate automaton (a conjecture) according to the observations made on the system that far and making an equivalence query to the system under learning (in fact, to an oracle which knows the model of the system), which is equivalent to asking whether the learned model is correct or not. If it is not, the system (in fact, the oracle) gives a counterexample, which is a string that is either accepted by the conjecture but not by the system or the reverse. In practice, equivalence queries are replied by conformance

testing or brute force search, both of which are expected to require long times [2][5]. Once the counterexample is obtained, the algorithm processes it by adding the path traced by the counterexample string to the conjecture and then constructing a consistent deterministic finite automaton (which will be the next conjecture) by making membership queries to the system, which are equivalent to asking whether a given string is accepted by the actual automaton/system or not. This procedure continues until the automaton is correctly learned. Finally, it is guaranteed that the DFA learned by L^* Algorithm is minimal.

Even though the model (of an interactive constraint-system) learned by the L^* Algorithm will be compact, there are problems with the efficiency of the learning method (in the scope of learning interactive constraint-based systems) and effectiveness/informativeness of the produced model:

Effectiveness. When the system under learning is an interactive constraint system, the model learned by the L^* Algorithm, or DFA-based models of interactive constraint-based systems in general, correctly model state-based behaviour of those systems. However, in contrast to FA-VAs, they neither keep track of values of variables nor associate states with the valuations. This makes it difficult for a human observer to associate the valuation-based nature of the constraint-based systems with the representation and to reason about the system by investigating/analyzing the model. On the other hand, FA-VAs both keep track of values of variables and associate states with the valuations. Thus, they enable human observers to see the effect of each variable assignment at each state; and for each legal valuation, to see the sequences of variable assignments that convert that legal valuation to an illegal valuation. Further, they categorize the valuations and the values (of each variable) into uniformly behaving groups/clusters under the existing set of constraints (they form groups of valuations and groups of values separately), thus enable human observers to see each group (of values or of valuations) as a single entity. This abstraction helps the human observers to understand the modeled system, which will be discusses at Chapter 4.

Efficiency. L^* Algorithm assumes that the system under learning is an arbitrary finite automaton. However, finite automata representing interactive constraint-based systems (i.e. FA-VAs) are a specialized type of DFA such that each of their transitions

is equivalent to a variable assignment. This specialization/restriction implies many other properties that distinguish those automata from general finite automata, as will be shown in Chapter 2. L^* Algorithm, naturally, does not exploit those properties. Therefore, it cannot use all existing sources of information, further, cannot escape equivalence queries. The algorithm LearnFAVA, which will be presented in Chapter 3, makes heavy use of the distinguishing properties of FA-VAs and does not require equivalence queries.

Given all variables of the system are categorical/qualitative variables (i.e. each value can be considered as a separate symbol) [8] (or given there exist no linear, quadratic or higher polynomial constraints defined on multiple variables even if the variables are numeric variables) the proposed algorithm (i.e. LearnFAVA) is able to infer the interactive constraint-based system under learning; moreover, generates a compact representation of the learned system. In case linear or polynomial constraints exist, even though the algorithm is able to generate a FA-VA correctly representing the system, the compactness of representation is lost (i.e. the state space explodes). This is the main limitation of this work. Possible solutions that may overcome this limitation is discussed at the end (i.e. Chapter 4). Moreover, effectiveness and other current limitations of the proposed representation method and learning algorithm are also discussed. Potential solutions that are expected to overcome the current limitations are sketched. Potential future works are proposed.

Additionally, it is important to note that there exists other active automata learning algorithms that infer the systems under learning in the form of register automata [7][9][10]. Those algorithm can be considered as adaptations/extensions of L^* Algorithm to the problem of inferring register automata; and they are designed to learn systems that either cannot be represented or cannot be compactly and effectively represented with finite automata. Since they are able to learn (and model) not only the control flow but also the data flow of the systems, at the first sight such algorithms may seem to provide a perfect solution to the problem on which this thesis focuses. However, this is not correct since systems considered in this work do not have complex data flow given they do not have additional properties to be modeled such as some additional actions triggered according to the results of some operations between some values, which are out of the scope of this work. To be more specific, the systems

considered in this work have (and driven by) a single type of data operation: variable assignment operation. On the other hand, as already underlined, keeping track of data values (i.e. values assigned to variables) is important for the effectiveness of the representation and greatly helps while inferring systems under consideration. Nevertheless, register automata learning algorithms, naturally, are designed to learn general register automata, therefore they use relatively higher amounts of computational resources. However (interactive) constraint-based systems can be represented with FA-VAs, which are computationally less powerful and easier to infer. The algorithm LearnFAVA provides a more efficient way of learning/modelling interactive constraint-based systems by making use of the properties special to FA-VAs, as will be shown in the following chapters.

Furthermore, there exist an earlier study [11] which suggests modelling constraint satisfaction problems using finite automata. However, the focus of that study is neither learning nor effectively modelling constraint-based systems but solving constraint satisfaction problems. In parallel, it proposes a relatively restricted representation which does not explicitly keep track of values and assumes an order in which assignments to the variables are made. Even so, it reaches compact models which have some similarities with FA-VAs. However, that study does not investigate the special properties of the subclass of finite automata representing constraint problems/systems (It does not define such a subclass). Moreover, it does not consider the problem of inferring/learning finite automata which represent an unknown constraint-based system but only models already specified constraint problems in terms of finite automata. In accordance, the method proposed in that study is based on methods/algorithms developed for general deterministic finite automata, in essence, on product automata construction and on a general state minimization algorithm. It does not provide special methods exploiting special properties of the automata representing constraint problems/systems while generating the models.

Lastly, the algorithmic inference part of this thesis intersects with the area of constraint acquisition [12][13][14][15]. Constraint acquisition may be defined as the study of methods of identifying and grouping the constraints on a system either passively from the existing data or actively by making experiments on (i.e. queries to) the system. The algorithm LearnFAVA presented in Chapter 3 implicitly does a sim-

ilar identification. On the other hand, there is a significant difference between this work and general constraint acquisition studies: In constraint acquisition, the goal is to acquire the underlying constraint satisfaction problem(s) defined on the system under learning, there is no significant interest in further modelling/representing the system. On the other hand, the main focus of this study (and the algorithm Learn-FAVA) is creating effective representations of interactive constraint-based systems by modelling them in terms of FA-VAs.

1.2 The Outline of the Thesis

The following chapters are organized as follows:

- Ch2. defines finite automata for variable assignment languages (FA-VA) and analyses their properties.
- Ch3. introduces and explains the algorithm LearnFAVA, which is the baseline active automata learning algorithm for inferring finite automata for variable assignment languages (FA-VAs).
- Ch4. discusses effectiveness, potential, and limitations of the proposed representation method and algorithm.

CHAPTER 2

FINITE AUTOMATA FOR VARIABLE ASSIGNMENT LANGUAGES

2.1 Variable Assignment Languages

Definition 2.1.1. Given a variable v, $domain(v)$ denotes the set of all values the variable v can take.

Definition 2.1.2. Given a set of variables V and the mapping D from variables to their domains (i.e. $D = \{v_i \rightarrow domain(v_i) | v_i \in V\}$), $D(v_i)$ denotes $domain(v_i)$.

Definition 2.1.3. The operation that assigns a value $val \in domain(v)$ to a variable v is denoted as $v \leftarrow val$ and named as a **variable assignment** operation.

Definition 2.1.4. [] is the **index operator**. Applied on a tuple t, $t[i]$ denotes the i^{th} element of the tuple where $1 \leq i \leq |t|$ (i.e. indices start from 1). Further, $t[i, j]$ denotes the tuple consisting of elements $t[i]$ (inclusive) to $t[j]$ (inclusive) where $i \leq j \leq |t|$. If $i > j$, $t[i, j]$ is the empty tuple.

Definition 2.1.5. Given a finite set V of variables with non-empty domains, and the mapping D from variables in V to their domains, each element c of the set $C = \prod_{v_i \in V} D(v_i)$ is a **complete assignment** (of variables in V). Notably, C, the set of all complete assignments over (V, D), is finite when $D(v_i)$ is finite for each $v_i \in V$.

In simpler terms, a complete assignment is a set of assignments which contains exactly one assignment for each of the defined variables. Further, in accordance with the complete assignment definition, and borrowing the concept of constraint from the field artificial intelligence [3], the concept of variable assignment constraint is defined as follows:

Definition 2.1.6. Given $C = \prod_{v_i \in V} domain(v_i)$, a **variable assignment constraint** is a subset A of C where each $a \in A$ is a legal/allowable complete assignment of variables. On the other hand, given a set of constraints $X = \{A_1, A_2, ..., A_n\}$, each element a' of the set $A' = C \setminus \bigcup_{A_i \in X} A_i$ is an **illegal assignment**; in other words, each $a' \in A'$ violates one or more constraints.

Definition 2.1.7. Given a finite set V of variables with finite non-empty domains and the mapping $D = \{v_i \to domain(v_i) \,|\, v_i \in V\}$ from those variables to their domains, any *finite* sequence over an alphabet $\Sigma = \{a[1] \leftarrow a[2] \,|\, a \in \bigcup_{v_i \in V}(\{v_i\} \times D(v_i))\}$ consisting of variable assignment operations over (V, D) is a **variable assignment sequence**.

Definition 2.1.8. Where v is a variable and w is a variable assignment sequence, $last_v(w)$ denotes the last assignment of the variable v at the sequence w. If there is no assignments to the variable v in the sequence, $last_v(w)$ is either$\perp$ or equals to a previously specified default value $val \in D(v)$.

Definition 2.1.9. $LAST_V(w)$ denotes the tuple $(last_{v_1}(w), last_{v_2}(w), ..., last_{v_n}(w))$ where $V = \{v_1, v_2, ...v_n\}$ ($n \in \mathbb{Z}^+$). In other words, $LAST_V(w)$ denotes the final complete assignment formed by the sequential application of the all variable assignments at the variable assignment sequence w.

Definition 2.1.10. When the index operator is applied on a sequence w, $w[i]$ (where $1 \leq i \leq |w|$) denotes the i^{th} element of the sequence w. Further, $w[i, j]$ (where $1 \leq i \leq j \leq |w|$) denotes the subsequence of w starting from its i^{th} element (inclusive) and ending at its j^{th} element (inclusive).

Definition 2.1.11. Given a variable assignment sequence w, $pre_j(w)$ denotes the j-long prefix of the string w.

Definition 2.1.12. Given a quintuple (V, D, Σ, p, A') where

- V is a finite set of variables with non-empty finite domains,
- $D = \{v_i \to domain(v_i) \,|\, v_i \in V\}$ is the mapping from variables in V to their domains,
- $\Sigma = \{a[1] \leftarrow a[2] \,|\, a \in \bigcup_{v_i \in V}(\{v_i\} \times D(v_i))\}$ is the alphabet consisting of all variable assignment operations over (V, D),

- $p \in \prod_{v_i \in V} D(v_i)$ is an initial complete assignment,
- $A' \subseteq \prod_{v_i \in V} D(v_i)$ is the set of illegal complete assignments;

a **variable assignment language** $L = (V, D, \Sigma, p, A')$ is the subset of all variable assignment sequences over Σ such that for each prefix w' of each sequence $w \in L$ and for each illegal assignment $a' \in A'$, $LAST_V(w') \neq a'$ holds, where default (i.e. initial) value for each $last_{v_i}$ is $p[i]$ (i.e. initially $LAST_V$ is set to p).

To be more clear, since an assignment to a variable v_i overwrites the previous assignments to that variable by definition, $last_{v_i}$ in fact "reads" the lastly assigned, thus the current, value of the variable v_i. For a prefix $w' = pre_j(w)$, $LAST_V(w') \neq a'$ holding means that last assignments to each of the variables in the sequence w up to and including its j^{th} index do not form an illegal complete assignment. Then, that condition holding for all j's means that the sequence w never forms an illegal assignment. That is, intuitively, a variable assignment language is the set of all variable assignment sequences on a given (V, D) pair which never form any of the illegal assignments in a given A' when initiated with the initial complete assignment p.

Theorem 2.1.1. *The class of variable assignment languages is a subclass of the class of regular languages.*

Proof. Given a variable assignment language $L = (V, D, \Sigma, p, A')$ and a variable assignment sequence w, let $p_w = LAST_V(w)$. Then, a new variable assignment language $L_w = (V, D, \Sigma, p_w, A')$ which has the same set of illegal assignments as L but has a (potentially) different initial assignment p_w, can be defined. Thus, for any two variable assignment sequences $w \in L$ and w', $ww' \in L$ if and only if $w' \in L_w$ and $ww' \notin L$ if and only if $w' \notin L_w$. On the other hand, if $w \notin L$, then $ww' \notin L$ for any w' by the definition of variable assignment languages (i.e. the sequences that are not in the language are in the same single equivalence class). This concludes that any two variable assignment sequences $w_1, w_2 \in \Sigma^*$ are equivalent with respect to L if and only if either $w_1, w_2 \notin L$ or $L_{w_1} = L_{w_2}$ holds. Consequently, since L_w is defined by $LAST_V(w)$ and remembering for any w, $LAST_V(w)$ is a complete assignment over (V, D), the number of equivalence classes of $\sim_L$, where $\sim_L$ is the Nerode Congruence [16][17], is bounded from above by the number of complete assignments over (V, D). Since $|V|$ and $D(v_i)$ for any $v_i \in V$ are finite, there exist a finite number of different

complete assignments over (V, D); therefore, any variable assignment language L, $\sim_L$ has a finite number of equivalence classes. ■

The proof of the Theorem 2.1.1. implies there is a mapping from the set of all complete assignments over (V, D) to the set of equivalence classes of $\sim_L$ where $L = (V, D, \Sigma, p, A')$ is a variable assignment language. However, a crucial remark is, this mapping is not necessarily one-to-one. That is, given a variable assignment language L, for two different variable assignment sequences $w_1 \neq w_2$, L_{w_1} and L_{w_2} may be equivalent languages, thus constitute the same equivalence class of $\sim_L$, even if $c_1 \neq c_2$ where $c_1 = LAST(w_1)$ and $c_2 = LAST(w_2)$. Formally:

Proposition 2.1.2. *Given $L = (V, D, \Sigma, p, A')$, C is the set of all complete assignment over (V, D), $c_1, c_2 \in C$, and E is the set of equivalence classes of $\sim_L$; there exists a function f mapping each element c of C to an element e of E such that $f(c_1) = f(c_2)$ may hold even if $c_1 \neq c_2$ (i.e. f is not necessarily one-to-one).*

Proof. Say $c_1 = LAST_V(w_1)$, $c_2 = LAST_V(w_2)$, $c_1 \neq c_2$, $w_1 \neq w_2$. Given $L_1 = (V, D, \Sigma, c_1, A')$ and $L_2 = (V, D, \Sigma, c_2, A')$, $L_1 = L_2$ may still hold, thus c_1 and c_2 may be mapped by f to the same equivalence class e. A very simple example can be given as such:
Let $L = (V, D, \Sigma, p, A')$ where $V = \{v_1, v_2\}$, $D = \{v_1 \rightarrow \{a\}, v_2 \rightarrow \{0, 1, 2, 3\}\}$, $A' = \{(a, 3)\}$, and $p = (a, 1)$.
Let $w_1 =$ "$v_1 \leftarrow a, v_2 \leftarrow 2,$" and $w_2 =$ "$v_2 \leftarrow 1, v_2 \leftarrow 0$".
Then, $c_1 = LAST_V(w_1) = (a, 2)$, thus the equivalence class to which c_1 is mapped is $L_1 = (V, D, \Sigma, c_1, A')$; and $c_2 = LAST_V(w_2) = (a, 0)$, thus the equivalence class to which c_2 is mapped is $L_2 = (V, D, \Sigma, c_2, A')$. However, L_1 and L_2 are equivalent; they both are the set of variable assignment sequences over Σ which do not include the assignment "$v_2 \leftarrow 3$".■

The next section will define the class of automata that are acceptors for variable assignment languages: A special type of deterministic finite automata which keeps track of not only the input sequence w but also complete assignments $c = LAST_V(w')$ produced by each prefix w' of the input sequence w.

Further, given a variable assignment language L, in order to conclude which complete assignments are mapped to the same equivalence classes, checking each pair of

complete assignments in terms of equivalence under $\sim_L$ is not required; that mapping is directly implied by the language definition, specifically by the set of illegal assignments. This will also be elaborated on at the following sections of this chapter, and underlying structure and rules will be shown.

2.2 Finite Automata for Variable Assignment Languages

Before defining the Finite Automata for Variable Assignment Languages (FA-VA), which are acceptors for Variable Assignment Languages, let us define the apply operator which will be used in the definition of FA-VA:

Definition 2.2.1. The **apply operator** $\alpha(c, i, \sigma)$ takes a complete assignment c, an index i, and a variable assignment $\sigma = v_i \leftarrow val$; yields a complete assignment c' such that $c'[k] = c[k]$ for all $1 \leq k \leq |c|$ except $k = i$, and $c'[i] = val$. This is, the apply operator applies the variable assignment declared by its argument $\sigma = v_i \leftarrow val$ to its argument c, thus, alters the value at the i^{th} index of c to val, leaving other indices unchanged. Moreover, σ in the $\alpha(c, i, \sigma)$ may in short be denoted by only the value val, since the index i already indicates that the value is assigned to the i^{th} variable.

Definition 2.2.2. A **finite automaton for variable assignment languages (FA-VA)** is an 8-tuple $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$ where

- Q is the finite set of states,
- V is a finite non-empty set of variables,
- $D = \{v_i \rightarrow domain(v_i) | v_i \in V\}$ the mapping from variables in V to their domains.
- $\Sigma = \{\text{“}a[1] \leftarrow a[2]\text{”} \mid a \in \bigcup_{v_i \in V}(\{v_i\} \times D(v_i))\}$ is the alphabet,
- $s \in Q$ is the starting state,
- $E \subseteq Q$ is the set of error states,
- $\rho : \prod_{v_i \in V} D(v_i) \rightarrow Q$ is an onto function mapping each complete assignment of variables in V to a state in Q.
- δ is the transition function from $Q \times \Sigma$ to Q.
  δ is restricted such that $(q, v_i \leftarrow val, q') \in \delta$ if and only if
  $\rho^{-1}(q') \supseteq \{c' | c' = \alpha(c, i, v_i \leftarrow val) \ for \ c \in \rho^{-1}(q)\}$.

In other words, $(Q, \Sigma, \delta, s, E)$ is a deterministic finite automaton with three modifications:

- Instead of a set of final states, E, the set of error states, is included. This does not syntactically alter the traditional finite automaton structure; however, it does semantically alter, which will be addressed later.
- Σ is restricted from being a set of arbitrary string characters/symbols to the set of all possible (single) variable assignment operations defined over (V, D).
- There is an function ρ which restricts δ.

Further, the additional elements of the 8-tuple, in essence $V, D,$ and ρ, are the components that are necessary to retain the relation with variable assignment languages. Among those, the function ρ enforces a restriction on the transition relation δ in a way analogous to V and D do on Σ. In fact, ρ does not only restrict but determines the transition relation δ. Thus, the function ρ is the core component of FA-VA definition.

Elaborating on the nature of ρ, it must be noted that ρ is not necessarily one-to-one (but it is onto). Inverse of the function ρ, denoted by ρ^{-1}, is a one-to-many relation from the set of states to the set of all possible complete assignments of the given variables over the given domains (i.e. from Q to $\prod_{v_i \in V} D(v_i)$). In plain English, each state has at least one (or more) associated complete assignment, whereas each of the complete assignments defined over (V, D) are associated to exactly one state.

Let us denote the variable assignment language accepted by a FA-VA M as $L(M)$. Since FA-VAs are designed to be acceptors for variable assignment languages, given $L(M) = L_1$ where M is a FA-VA and L_1 is a variable assignment language, there must be a function that maps each state q of M to an equivalence class e of L_1. As shown in the proof of Theorem 2.1.1., each equivalence class (except $\Sigma^* - L$) of sequences under a variable assignment language L is equivalent to another variable assignment language $L_w(V, D, \Sigma, c', A')$ where $c' = LAST_V(w)$. Notably, the only component of L_w that differs from L is c'. Then, the complete assignment $c' = LAST_V(w)$ that is produced by a sequence w inputted to a FA-VA indicates the equivalence class L_w to which w belongs. The role of ρ^{-1} is indicating that complete assignment - equivalence class, thus complete assignment - state relation. That is, $\rho^{-1}(q)$ is equivalent to the set of complete assignments

$C_q = \{c | c = LAST_V(w)\ for\ w \in W_q\}$ where W_q is the set of variable assignment sequences that end up at state q when inputted to the M.

The defining characteristic of FA-VAs follow from the dependency between δ and ρ. Therefore, that dependency requires further explanation. Moreover, some other critical properties derive from that property. That is, it will often be addressed in the following. Therefore, let us rephrase and name it for the ease of reference:

Property 1. *For any FA-VA $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$, the two propositions below are equivalent given $q \in Q$, $q' \in Q$, $\sigma = v_i \leftarrow val \in \Sigma$:*

1. $(q, v_i \leftarrow val, q') \in \delta$
2. $\rho^{-1}(q') \supseteq \{c' | c' = \alpha(c, i, v_i \leftarrow val)\ for\ c \in \rho^{-1}(q)\}$.

Intuitively, proposition (1) entailing proposition (2) implies that in a transition, the variable assignment $v_i \leftarrow val$ (i.e. the alphabet symbol on which the transition is made) applies itself to each complete assignments c associated to the current state q (i.e. to each of the elements of $\rho^{-1}(q)$), thus at each c, the value of the variable v_i changes to val while the values of all other variables are preserved.

On the other hand, proposition (2) entailing proposition (1) implies that for any two state q and q', if all complete assignments c' that are generated by applying the assignment $v_i \leftarrow val$ to each $c \in \rho^{-1}(q)$ are included in $\rho^{-1}(q')$, then there is a transition $(q, v_i \leftarrow val, q') \in \delta$.

Importantly, Property 1 is consistent with the proof of the Theorem 2.1.1.:
Say M accepts the variable assignment language L. Then, the equivalence class (of variable assignment sequences) under L which corresponds to the state q of M is $L_w = (V, D, \Sigma, c, A')$ where c is any element from $\rho^{-1}(q)$ (i.e. L_w is the same for different complete assignments $c \in \rho^{-1}(q)$). Concatenating a sequence $\sigma = v_i \leftarrow val$ to the end of all sequences $w \in L_w$, another equivalence class $L_w\sigma$ is obtained (At the automaton-side, this corresponds M making a transition from q with σ). Since the only last assignment affected by that concatenation is $last_{v_i}$, $LAST_V(w\sigma) = \alpha(LAST_V(w), i, \sigma))$. Thus, $L_{w\sigma} = (V, D, \Sigma, \alpha(c, i, \sigma), A')$, which is the equivalence class corresponding to the state q' of M.

At this point, we are ready to define operational semantics of Finite Automata for

Variable Assignment Languages. Given a FA-VA $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$ and variable assignment sequences w and w':

Definition 2.2.3. Denoted as (q, w) where $q \in Q$, a **configuration** of M represents the current state of M and the currently unread part w of the sequence inputted to M.

Definition 2.2.4. $q \xrightarrow{\sigma} q'$ denotes a **step** of M where $q, q' \in Q$ and $\sigma \in \Sigma$. A step is valid if a transition $(q, \sigma, q') \in \delta$ exists. While the current configuration of M is (q, w), M takes a step by reading the first element of w (i.e. $w[1]$) and yields the new configuration (q, w') where $w' = w[2, |w|]$.

Definition 2.2.5. A **run** of M is defined over a variable assignment sequence w. When inputted with w, M starts with the initial configuration (s, w) and takes steps until it reaches a final configuration (q', ϵ) where $q' \in Q$ and ϵ is the empty string. The sequence of configurations yielded at each of the steps, from initial configuration (inclusive) to the final configuration (inclusive), constitute a **run** of M.

Definition 2.2.6. A run of M is an **accepting run** if for none of the configurations (q', w') constituting that run, $q' \in E$ holds. Intuitively, any run in which a FA-VA M does not visit (and does not end up at) an error state is an accepting run.

Corollary 1. *An obvious result of the Definition 2.2.6 is that for any two variable assignment sequences w_1, w_2, if M reaches a configuration (q_e, ϵ) by reading w_1 where $q_e \in E$, that is, if M rejects (i.e. does not accept) w_1, then it also rejects $w_1 w_2$ since the condition of rejection (i.e. visiting an error state) is already satisfied by w_1.*

Definition 2.2.7. The **maximal FA-VA** $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$ accepting a variable assignment language $L = (V, D, \Sigma, p, A')$, is the FA-VA which has exactly one state for each complete assignment c defined over (V, D). That is, ρ is one-to-one, and $E = \{\rho(c_a) | c_a \in A'\}$ (i.e. the only error states are states q_e for which $\rho(q_e)$ is an erroneous assignment).

Proposition 2.2.1. *For any variable assignment language L, there exists a maximal FA-VA M accepting L (i.e. $L(M) = L$).*

Proof. For any variable assignment language $L = (V, D, \Sigma, p, A')$, the maximal FA-VA M accepting L can be defined as such:
$M = (Q, s, \Sigma, \delta, E, V, D, \rho)$,

- $Q = \{q_c | c \in C\}$ where $C = \prod_{v_i \in V} D(v_i)$ is the set of all complete assignments over (V, D).

  That is, there is a state for each complete assignment c over (V, D). Since the set of all complete assignments over (V, D) is finite, there are finitely many states.
- $\rho = \{(c, q_c) | c \in C\}$.

  Notice that ρ is one-to-one since there is exactly one q_c for each $c \in C$, by the definition of Q.
- $\delta = \delta_1 \cup \delta_2$ where δ_1 and δ_2 are defined as such:

  $\delta_1 = \{(q_c, v_i \leftarrow val, q_{c'}) \mid q_c \neq q_{c'} \ \wedge \ q_c, q_{c'} \in Q \ \wedge \ c'[i] = val \ \wedge \ \forall j \neq i . c[j] = c'[j] \ \wedge \ i, j \in [1, |V|]\}$

  That is, if two complete assignments c and c' differ at the value of only one variable v_i, and say $c'[i] = val$, then there is a transition from q_c to $q_{c'}$ with $\sigma = v_i \leftarrow val$. Notably, for any such transition, there is also a transition from $q_{c'}$ to q_c with $\sigma' = val'$ where $c[i] = val'$ and $val \neq val'$, which is already implied by the definition of δ.

  $\delta_2 = \{(q_c, v_i \leftarrow val, q_c) \mid c[i] = val \ \wedge \ i \in [1, |V|]\}$, which is the set of self-transitions. For each alphabet symbol σ, there is an outgoing transition from each state q_c. In case $c[i] = val$ and $\sigma = v_i \leftarrow val$, the transition ends up at q_c itself. This follows from that: Given a variable assignment sequence w and given $w' = pre_j(w)$, $w'' = pre_{j+1}(w)$, if $LAST_V(w')[i] = w''[j+1]$ holds, then $LAST_V(w')[i] = LAST_V(w'')[i]$ also holds since the assignment at $w''[j+1]$ does not change the value of a variable.
- $E = \{q_c | c \in A'\}$ (i.e. $E = \{\rho(c) | c \in A'\}$)
- $s = q_p$ (i.e. $\rho^{-1}(s) = p$). ■

The term "maximal" follows from the fact that the number of states of a FA-VA $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$ is bounded from above by the number of complete assignments defined over (V, D) since ρ is an onto function. That is, $\rho(c)$ yields an exact, single element of its image set; moreover, its image set consists only of the elements q for which there exist a c in the domain of ρ such that $\rho(c) = q$ (by the definition of onto function). When ρ is one-to-one, it yields a different state q for each complete assignment c from its domain, thus the cardinality of its image set (i.e.

Q) reaches its upper bound with $|Q| = |C|$ where C is the set of all complete assignments over (V, D). In other words, $|C|$ is the upper bound on the number of states $|Q|$ and that upper bound is reached by maximal FA-VA.

Now, let us focus on some properties derived from the Property 1, which will help us to understand and define FA-VAs that are not maximal. However, in advance we should define an extended version of the index operator that operates on sets of complete assignments, which will help us explain those properties.

Definition 2.2.8. Given C' is a set of complete assignments over (V, D) and $1 \leq k \leq |V|$, $C'[k]$ yields the set $d'_k = \{c[k] | c \in C'\}$ (Notice that $d'_k \subseteq D(v_k)$ where v_k is the k^{th} variable).

Proposition 2.2.2. *Given a FA-VA $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$, for each state $q \in Q$,* $\rho^{-1}(q) = \prod_{1 \leq i \leq |V|} \rho^{-1}(q)[i]$.

Proof. Recall that from each state exactly one outgoing transition with each of the alphabet symbols exists. That is, for each state q and each alphabet symbol $\sigma = v_i \leftarrow val$, there is an outgoing transition (q, σ, q'). Then, let us focus on the transitions in the form $(q, v_i \leftarrow val, q)$ (i.e. self-transitions):

According to Property 1, a transition $(q, v_i \leftarrow val, q)$ exists if and only if $\rho^{-1}(q) \supseteq \{c | c = \alpha(c, i, v_i \leftarrow val) \; for \; c \in \rho^{-1}(q)\}$. Then, focusing on the self-transitions $(q, v_1 \leftarrow val, q)$, $\rho^{-1}(q) \supseteq (\{val\} \times \rho^{-1}(q)[2, |V|])$ for each val such that $val \in \rho^{-1}(q)[1]$, by Property 1 and the definition of the apply operator.
Then, since $\rho^{-1}(q)$ cannot have an element $\{val'\} \times \rho^{-1}(q)[2, |V|]$ such that $val' \notin \rho^{-1}(q)[1]$, $\bigcup_{val \in \rho^{-1}(q)[1]}(\{val\} \times \rho^{-1}(q)[2, |V|]) = \rho^{-1}(q)$ holds. Since $\bigcup_{val \in \rho^{-1}(q)[1]}(\{val\} \times \rho^{-1}(q)[2, |V|]) = \rho^{-1}(q)[1] \times \rho^{-1}(q)[2, |V|]$, $\rho^{-1}(q) = \rho^{-1}(q)[1] \times \rho^{-1}(q)[2, |V|]$ also holds.

By the same reasons, and since $\rho^{-1}(q) = \rho^{-1}(q)[1] \times \rho^{-1}(q)[2, |V|]$ is already shown, $\bigcup_{val \in \rho^{-1}(q)[2]}(\rho^{-1}(q)[1] \times \{val\} \times \rho^{-1}(q)[3, |V|]) = \rho^{-1}(q)$ also holds. Therefore, $\rho^{-1}(q) = \rho^{-1}(q)[1] \times \rho^{-1}(q)[2] \times \rho^{-1}(q)[3, |V|]$ holds.

Then, the same proof can be iteratively repeated for each $k > 2$ to show that $\rho^{-1}(q) = \rho^{-1}(q)[1] \times \rho^{-1}(q)[2] \times ... \times \rho^{-1}(q)[k] \times \rho^{-1}(q)[k+1, |V|]$ also holds.

Thus, $\rho^{-1}(q) = \prod_{1 \leq i \leq |V|} \rho^{-1}(q)[i]$ holds. ■

Proposition 2.2.3. *For any two state $q, q' \in Q$, if $c \in \rho^{-1}(q)$, $c' \in \rho^{-1}(q')$ such that $c[i] \neq c'[i]$ for exactly one i and $c[j] = c'[j]$ for all $j \neq i$, where $i, j \in [1, |V|]$; then the transitions $(q, v_i \leftarrow c'[i], q')$ and $(q', v_i \leftarrow c[i], q)$ necessarily exist.*

Proof. For notational convenience, let $c[i] = val$ and $c'[i] = val'$. Under given conditions, $\alpha(c, i, v_i \leftarrow val')$ equals c'. Since $v_i \leftarrow val'$ is an alphabet symbol, the transition $(q, v_i \leftarrow val', q'') \in \delta$ for some $q'' \in Q$, by the definition of FA-VAs. Then, by Property 1, $\rho^{-1}(q'') \supseteq \{c''|c'' = \alpha(c''', i, v_i \leftarrow val)\ for\ c''' \in \rho^{-1}(q)\}$ necessarily holds, which implies $c' \in \rho^{-1}(q'')$. Since $\rho(c') = q'$ and ρ is a function, $q'' = q'$ necessarily holds. The same proof where q and q', c and c', val and val' are interchanged is valid for the existence of the transition $(q', v_i \leftarrow c[i], q)$.■

Proposition 2.2.4. *All transitions of a FA-VA M are invertible at one step. That is, for any transition (q, σ, q'), there is an inverse transition (q', σ', q) where $q, q' \in Q$ and $\sigma, \sigma' \in \Sigma$.*

Proof. Obviously, each self-transition (each transition in the form (q, σ, q)) is the inverse of itself, since the sequence of transitions $(q, \sigma, q)(q, \sigma, q)$ starts and ends at the state q. When it comes to the other transitions (q, σ, q'), where $q \neq q'$, we can proceed as such: Say there is a transition $(q, v_i \leftarrow val', q') \in \delta$. Randomly select an element $c \in \rho^{-1}(q)$ where $c[i] = val$. By Property 1, $c' = \alpha(c, i, v_i \leftarrow val') \in \rho^{-1}(q')$ necessarily holds. By the definition of FA-VAs, there exists a transition $(q', v_i \leftarrow val, q'') \in \delta$ from q' to some state $q'' \in Q$; and according to Property 1, the existence of that transition requires $\alpha(c', i, v_i \leftarrow val) \in \rho^{-1}(q'')$ to hold. It is known that $\alpha(c', i, v_i \leftarrow val) = c$ and $c \in \rho^{-1}(q)$ (i.e. $\rho(c) = q$). Then, since ρ is a function, $q = q''$ necessarily holds.■

Proposition 2.2.5. *For any state $q \in Q$,*
the set of self-transitions from/to q is $\delta'_q = \{(q, \sigma', q) \mid \sigma' \in \Sigma'\}$
where $\Sigma' = \{\sigma' \mid \sigma' = v_i \leftarrow val\ for\ 1 \leq i \leq |V|\ and\ val \in \rho^{-1}(q)[i]\}$.

Proof. For a $c \in \rho^{-1}(q)$ and a variable $v_i \in V$, say $c[i] = val$. Then, the transition $(q, v_i \leftarrow val, q')$, necessarily exists (by the definition of FA-VAs). According to Property 1, the existence of that transition requires $\alpha(c, i, v_i \leftarrow val) \in \rho^{-1}(q')$ to hold. Since $c[i] = val$, $\alpha(c, i, v_i \leftarrow val)$ is equivalent to c itself; and since $\rho(c) = q$

and ρ is a function, $q' = q$ necessarily holds. That is, the transitions of the type described in the proposition are necessarily self-transitions.

Further, no other transitions are self-transitions:
By Property 1, if there is a transition $(q, v_i \leftarrow val', q) \in \delta$, $\alpha(c, i, v_i \leftarrow val') \in \rho^{-1}(q)$ necessarily holds. That is, for any such $v_i \leftarrow val'$, $val' \in \rho^{-1}(q)[i]$ holds, therefore $v_i \leftarrow val' \in \Sigma'$ holds by definition of the Σ'.■

Proposition 2.2.6. *If $(q, v_i \leftarrow val, q') \in \delta$, where $q \neq q'$, then $\rho^{-1}(q)[i] \neq \rho^{-1}(q')[i]$ and $\rho^{-1}(q)[j] = \rho^{-1}(q')[j]$ for each $j \neq i$, where $i, j \in [1, |V|]$. That is, if there is a transition from a state q to another state q' on an assignment to the variable v_i, $\rho^{-1}(q)$ and $\rho^{-1}(q')$ differ from each other only at the i^{th} index.*

Proof. Firstly, under the given conditions, it is straightforward that $\rho^{-1}(q)[i] \neq \rho^{-1}(q')[i]$ since val is not in $\rho^{-1}(q)[i]$ due to Proposition 2.2.5 (if it were, q and q' would necessarily be the same state); moreover, $\rho^{-1}(q') \supseteq \alpha(c, i, v_i \leftarrow val)$, which implies $val \in \rho^{-1}(q')[i]$, necessarily due to Property 1.

Further, to show that $\rho^{-1}(q)[j] = \rho^{-1}(q')[j]$ holds for each $j \neq i$, assume there is an index $k \neq i$ such that $\rho^{-1}(q)[k] \neq \rho^{-1}(q')[k]$ (proof by contradiction). That is, there is least one value val_k of the k^{th} variable v_k such that either $val_k \in \rho^{-1}(q)[k]$ and $val_k \notin \rho^{-1}(q')[k]$ or $val_k \notin \rho^{-1}(q)[k]$ and $val_k \in \rho^{-1}(q')[k]$ holds. Assume the former. Then, there is at least one $c \in \rho^{-1}(q)$ such that $c[k] = val_k$ and $c[i] \neq val$. The existence of the transition $(q, v_i \leftarrow val, q')$ implies $c' = \alpha(c, i, v_i \leftarrow val)$ is in $\rho^{-1}(q')$ (by Property 1); and this implies $c'[k] = c[k] = val_k$ (i.e. k^{th} indice is left unchanged). However, $val_k \notin \rho^{-1}(q')$ is already given. Contradiction.

When the latter is assumed, the same proof (with q and q' interchanged) is valid since $(q, v_i \leftarrow val, q') \in \delta$ guarantees $(q', v_i \leftarrow val', q) \in \delta$ due to the Proposition 2.2.4. Thus, under the given conditions, the assumption that there is an index $k \neq i$ such that $\rho^{-1}(q)[k] \neq \rho^{-1}(q')[k]$ yields a contradiction.■

Proposition 2.2.7. *For any pair of states $q, q' \in Q$ such that $q \neq q'$, if there are multiple transitions (q, σ_k, q') from q to q' on different alphabet symbols σ_k, all such σ_ks are assignments to the same variable.*

Proof. For $|V| = 1$, the proposition trivially holds. For $|V| \geq 2$, let us randomly

select two variables, v_i and v_j where $v_i \neq v_j$. Consider a random $q \in Q$ and a random $c \in \rho^{-1}(q)$, and say $c[i] = val_i$, $c[j] = val_j$. Now, assume there is a transition $(q, v_i \leftarrow val'_i, q')$ where $q' \neq q$ (Please note that, by Proposition 2.2.5, $val'_i \notin \rho^{-1}(q)[i]$ since $q' \neq q$). Then, by Property 1, $c' = \alpha(c, i, v_i \leftarrow val'_i) \in \rho^{-1}(q')$, which also implies $c'[j] = val_j$ (same as $c[j]$) and $c'[i] = val'_i$. Also assume that there is another transition $(q, v_j \leftarrow val'_j, q')$ (Again, $val'_j \notin \rho^{-1}(q)[j]$ since $q' \neq q$). By Property 1, $c'' = \alpha(c, j, v_j \leftarrow val'_j) \in \rho^{-1}(q')$, which also implies $c''[i] = val_i$ (same as $c[i]$) and $c''[j] = val'_j$. Consequently, c' and c'' differ from each other exactly at i^{th} and j^{th} indices; and both are elements of $\rho^{-1}(q')$.

Then, by Proposition 2.2.5, there exists a transition $(q', v_i \leftarrow val_i, q')$ (since $c'' \in \rho^{-1}(q')$ and $c''[i] = val_i$). By Property 1 and since $c' \in \rho^{-1}(q')$, the existence of that transition requires $c''' = \alpha(c', i, v_i \leftarrow val_i) \in \rho^{-1}(q')$ to hold. However, $c''' = c$ and $\rho(c) = q$. Since $q \neq q'$ is given and since ρ is a function, contradiction is reached. That is, given $v_i \neq v_j$, $q' \neq q$, and $(q, v_i \leftarrow val'_i, q') \in \delta$; $(q, v_j \leftarrow val'_j, q') \in \delta$ cannot hold. In plain English, if there are multiple transitions from a state q to another state q' exist, those transitions cannot be assignments to the different variables. ■

Lemma 2.2.8. *For any state $q \in Q$, in case $\rho^{-1}(q)[i] \neq D(v_i)$, $val \in \rho^{-1}(q)[i]$ if and only if there exists a transition $(q', v_i \leftarrow val, q) \in \delta$, where $q' \in Q$ and $q \neq q'$.*

Proof. If part:
If there exists a transition $(q', v_i \leftarrow val, q) \in \delta$, for each $c' \in \rho^{-1}(q')$ there exist an element c in $\rho^{-1}(q)$ such that $c = \alpha(c', i, v_i \leftarrow val)$ by Property 1. This directly implies $val \in \rho^{-1}(q)[i]$ if such a transition exists.

Only if part:
Given $C = \prod_{v_i \in V} D(v_i)$; for each i satisfying the condition $\rho^{-1}(q)[i] \neq D(v_i)$, there is a nonempty subset C' of C such that $C'[i] = D(v_i) \backslash \rho^{-1}(q)[i]$ and $C'[k] = \rho^{-1}(q)[k]$ for each $k \in [1, |V|]$ except $k = i$. Moreover, by the definition of FA-VAs, for each $c' \in C'$, there exist a state q' such that $\rho(c') = q'$ (obviously, $q' \neq q$). Proposition 2.2.3 ensures that for each such q' and for each $c \in \rho^{-1}(q)$, there exist a transition $(q', v_i \leftarrow c[i], q)$ since $\rho^{-1}(q')$ has at least one element c' which differs from c only at the i^{th} index. Since $\rho^{-1}(q)[i] = \bigcup_{c \in \rho^{-1}(q)} c[i]$ by definition, it is concluded that for each $val \in \rho^{-1}(q)[i]$, there is at least one transition $(q', v_i \leftarrow val, q)$.■

Corollary 2 below is a direct implication of Lemma 2.2.8:

Corollary 2. *For any $q \in Q$, $\rho^{-1}(q)$ is inferable from transitions incoming to q from other states.*

2.3 Fundamental Finite Automata for Variable Assignment Languages

Definition 2.3.1. Given a variable assignment language L, a **minimal FA-VA** accepting L is a FA-VA with the minimum number of states that accepts L.

Definition 2.3.2. A minimal FA-VA with 1 error state is a **fundamental finite automaton for variable assignment languages**.

Before presenting the method of construction of minimal FA-VAs, let us explore another important property of general FA-VAs, which will be useful while proving the minimality of FA-VAs that are generated by the construction that will be given in the proof of the Lemma 2.3.2:

Proposition 2.3.1. *Given a FA-VA $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$, for any pair of states $q, q' \in Q$ and for each i such that $1 \leq i \leq |V|$, either $\rho^{-1}(q)[i] \cap \rho^{-1}(q')[i] = \emptyset$ or $\rho^{-1}(q)[i] = \rho^{-1}(q')[i]$ holds.*

Proof. To begin with, let us prove a relaxed version of the proposition: Given a FA-VA $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$, for any pair of states $q, q' \in Q$ such that $q \neq q'$ and for each i such that $1 \leq i \leq |V|$, exactly one of the following propositions holds:

1. $\rho^{-1}(q)[i] \cap \rho^{-1}(q')[i] = \emptyset$
2. $\rho^{-1}(q)[i] \subset \rho^{-1}(q')[i]$
3. $\rho^{-1}(q)[i] \supset \rho^{-1}(q')[i]$
4. $\rho^{-1}(q)[i] = \rho^{-1}(q')[i]$.

Assume there is a pair of states $q, q' \in Q$ (s.t. $q \neq q'$) and an index k for which none of the propositions 1,2,3,4 given above hold. That is, there exist at least 3 values of v_k, say val_1, val_2, val_3, such that $val_1 \in \rho^{-1}(q)[k]$ and $val_1 \notin \rho^{-1}(q')[k]$, $val_2 \notin \rho^{-1}(q)[k]$ and $val_2 \in \rho^{-1}(q')[k]$, $val_3 \in \rho^{-1}(q)[k]$ and $val_3 \in \rho^{-1}(q')[k]$. Consider a sequence w such that w contains exactly one assignment to each variable $v \in V \backslash \{v_k\}$ but does not contain any assignment to v_k. Then, consider the states q_1 and q_2 such that the partial run of M started with the configuration (q, w) ends up at

q_1 and the partial run of M started with the configuration (q', w) ends up at q_2. Since there is a complete assignment $c \in \rho^{-1}(q)$ such that $c[k] = val_3$, by Property 1, there exist a complete assignment $c_3 \in \rho^{-1}(q_1)$ such that $c_3[k] = val_3$ and $c_3[i]$ equals to the value assigned by w for each other index $i \neq k$ (where $1 \leq i \leq |V|$). Notice that, $c_3 \in \rho^{-1}(q_2)$ also holds due to the existence of a complete assignment $c' \in \rho^{-1}(q')$ such that $c'[k] = val_3$ and due to Property 1. Then, since ρ is a function, q_1 and q_2 are necessarily the same state (will be referred as q_1 from now on). Similarly and importantly, again due to Property 1, there is at least one other complete assignment $c_1 \in q_1$ such that $c_1[k] = val_1$ and $c_1[i]$ equals to the value assigned by w for each other index $i \neq k$; and at least one another complete assignment $c_2 \in q_1$ such that $c_2[k] = val_2$ and $c_2[i]$ equals to the value assigned by w for each other index $i \neq k$; respectively since $val_1 \in \rho^{-1}(q)$ and since $val_2 \in \rho^{-1}(q')$.

Now, consider any such sequence w': w' does not contain an assignment to v_k and for each i such that $1 \leq i \leq |V|$ except $i = k$, w' contains exactly one assignment $v_i \leftarrow val_i$ such that $val_i \in \rho^{-1}(q)[i]$. Given the three complete assignment c'_1, c'_2, c'_3 such that

- $c'_1[k] = val_1$ and $c'_1[i]$ equals to the value assigned by w' (i.e. $c'_1[i]$ equals to some value val_i such that $val_i \in \rho^{-1}(q)[i]$) for each other index $i \neq k$,
- $c'_2[k] = val_2$ and $c'_2[i]$ equals to the value assigned by w' (i.e. $c'_2[i]$ equals to some value val_i such that $val_i \in \rho^{-1}(q)[i]$) for each other index $i \neq k$,
- $c'_3[k] = val_3$ and $c'_3[i]$ equals to the value assigned by w' (i.e. $c'_3[i]$ equals to some value val_i such that $val_i \in \rho^{-1}(q)[i]$) for each other index $i \neq k$;

the partial run of M started with the configuration (q_1, w') ends up at a state q'' for which, according to the Property 1, $c'_1 \in \rho^{-1}(q'')$, $c'_2 \in \rho^{-1}(q'')$, and $c'_3 \in \rho^{-1}(q'')$ necessarily holds. Notice that $c'_3 = c$; therefore and since $c \in \rho^{-1}(q)$ is previously given (and since ρ is a function), $q'' = q$ must hold. However, that implies
$c'_2 \in \rho^{-1}(q)$, which is impossible since $c'_2[k] = val_2$ and $val_2 \notin \rho^{-1}(q)[k]$ is initially given. Contradiction. No such pair of states $q, q' \in Q$ as defined at the initial assumption can exist. That is, at least one of the propositions 1,2,3,4 necessarily holds for each index i and each pair of states $q, q' \in Q$. Lastly, if one of those four propositions holds for two states q, q' and an index i, the other three cannot hold by the definitions of intersection, subset relation and set equality. Therefore, it is concluded that given a pair of states q, q', for each index i, exactly one of the propositions 1,2,3,4 always

holds.

At this point, so as to prove the main proposition, we need to prove $\rho^{-1}(q)[i] \subset \rho^{-1}(q')[i]$ cannot hold for some states q, q' and some index i, which can be done as such:

Assume there is a pair of states $q, q' \in Q$ (s.t. $q \neq q'$) and an index k for which $\rho^{-1}(q)[k] \subset \rho^{-1}(q')[k]$ holds. Then, there exist at least one value of v_k, say val', such that $val' \in \rho^{-1}(q')[k]$ but $val' \notin \rho^{-1}(q)[k]$. Therefore, there is at least one complete assignment $c' \in \rho^{-1}(q')$ such that $c'[k] = val'$, thus for which $c' \in \rho^{-1}(q)$ does not hold.

For later use, let us also show that "$\rho^{-1}(q)[i] = \rho^{-1}(q')[i]$ for each i except $i = k$" cannot hold: Assume it holds. Then, since $\rho^{-1}(q)[k] \subset \rho^{-1}(q')[k]$ is given, for each complete assignment $c \in \rho^{-1}(q)$, $c \in \rho^{-1}(q')$ must also hold, which cannot be true since ρ is a function (i.e. a complete assignment c cannot be assigned to more than one states by ρ).

Now, consider any such sequence w: w does not contain an assignment to v_k and for each i such that $1 \leq i \leq |V|$ except $i = k$, w contains exactly one assignment $v_i \leftarrow val_i$ such that $val_i \in \rho^{-1}(q)[i]$. Let $q'' \in Q$ be the state at which the partial run started with the configuration (q', w) ends up; and consider a complete assignment $c \in \rho^{-1}(q')$ for which both $c[k] \in \rho^{-1}(q)[k]$ and $c[k] \in \rho^{-1}(q')[k]$ holds. By Property 1, there exist a complete assignment $c_1 \in \rho^{-1}(q'')$ such that $c_1[k] = c[k]$ and $c_1[i]$ equals to the value assigned by w' (i.e. $c_1[i]$ equals to some value val_i such that $val_i \in \rho^{-1}(q)[i]$) for each other index $i \neq k$. Then, since $c[k] \in \rho^{-1}(q)[k]$ is also given, for each index i, $c[i] \in \rho^{-1}(q)[i]$ holds. Therefore, (and since it is already shown that "$\rho^{-1}(q)[i] = \rho^{-1}(q')[i]$ for each i except $i = k$" cannot hold), by Proposition 2.2.2, $c \in \rho^{-1}(q)$ necessarily holds. Therefore and since ρ is a function, $q'' = q$ must hold. However, since $c' \in \rho^{-1}(q')$ is given, by Property 1, there exist a $c_2 \in \rho^{-1}(q'')$ such that $c_2[k] = val'$ and $c_2[i]$ equals to the value assigned by w' for each other index $i \neq k$. Since $val' \notin \rho^{-1}(q)$, $q'' = q$ cannot hold. Contradiction. There cannot exist some pair of states $q, q' \in Q$ (s.t. $q \neq q'$) and an index k for which $\rho^{-1}(q)[k] \subset \rho^{-1}(q')[k]$ holds.

As the result, for each pair of states $q, q' \in Q$ such that $q \neq q'$ and for each i such that $1 \leq i \leq |V|$, either the proposition 1 or the proposition 4 holds.■

Following definitions are needed in the proof of Lemma 2.3.2:

Definition 2.3.3. Given a FA-VA $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$, a variable $v \in V$ is an **effective variable** of an error state $q_e \in E$ if and only if there exist at least one transition $(q, v \leftarrow val, q_e)$ from at least one non-error state $q \in Q \setminus E$ to q_e on some assignment to the variable v.

Definition 2.3.4. Given a FA-VA $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$ and given M_q is the FA-VA obtained by changing the M's initial state to $q \in Q$, the language accepted by M_q is denoted as L_q. That is, L_q is the language accepted by the FA-VA $M_q = (Q, q, \Sigma, \delta, E, V, D, \rho)$.

Definition 2.3.5. Given a FA-VA $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$, any two states $q, q' \in Q$ are **distinguishable** if and only if there is at least one sequence w over Σ such that either $w \in L_q$ and $w \notin L_{q'}$ or $w \in L_{q'}$ and $w \notin L_q$. In accordance with the proof of the Theorem 2.1.1, given w_1 is a sequence such that the run starting with the configuration (s, w_1) ends up at q (without visiting an error state) and w_2 is a sequence such that the run starting with the configuration (s, w_2) ends up at q' (without visiting an error state); w_1 and w_2 are not equivalent with respect to L if and only if q and q' are distinguishable from each other.

Lemma 2.3.2. *For each variable assignment language $L = (V, D, \Sigma, p, A')$ such that $A' = \prod_{1 \leq i \leq |V|} d'_i$ where $d'_i \subseteq D(v_i)$, there is a fundamental FA-VA M that accepts L.*

Proof. First of all, without loss of generality we can assume that $D_i \setminus d'_i \neq \emptyset$ holds for each i. If it does not hold for a d'_i, we can initially ignore v_i while constructing the minimal automaton since it obviously does not affect the branching behaviour; then rectify ρ and δ by replacing $\rho^{-1}(q)$ by $\rho^{-1}(q) \times d'_i$ for each $q \in Q$ and by adding $(q, v_i \leftarrow val'_i, q)$ to δ for each $q \in Q$ and each $val'_i \in d'_i$. The same construction also works for the case where $d'_i = D_i$ holds. Therefore, for the sake of simplicity, we can also assume that $d'_i \neq D_i$ for each i. To sum up, without loss of generality, we can assume each variable is an effective variable of the error state.

Since there must be a single error state, $\rho^{-1}(q_e)$ must be equal to A' where q_e is the error state. Notice that for any $L = (V, D, \Sigma, p, A')$ such that $A' = \prod_{1 \leq i \leq |V|} d'_i$ where $d'_i \subseteq D(v_i)$, A' (equivalently $\rho^{-1}(q_e)$) being equal to $\prod_{1 \leq i \leq |V|} d'_i$ naturally partitions the domain of each variable v_i such that $D(v_i) = d_i \cup d'_i$ where $d_i \cap d'_i = \emptyset$. Proposition 2.3.1 ensures those partitions are minimal (i.e. consist of minimum number of subdomains) for each effective variable. To be more specific, since Proposition 2.3.1 guarantees that for any pair of states q, q' and each index i, either $\rho^{-1}(q)[i] \cap \rho^{-1}(q')[i] = \emptyset$ or $\rho^{-1}(q)[i] = \rho^{-1}(q')[i]$ holds, and since $\rho^{-1}(q_e)[i] = d'_i$ for each i; for any other state $q \in Q$ and each index $1 \leq i \leq |V|$, either $\rho^{-1}(q')[i] = d'_i$ or $\rho^{-1}(q')[i] = d_i$ necessarily holds. Therefore, so as to be able to construct the minimal FA-VA, while constructing ρ (or ρ^{-1}), treating each d_i as a whole (as the construction below does) is necessary. Further, as shown at the beginning of this proof, domains of the variables that are not effective are not partitioned and do not affect the number of states. Then, a FA-VA $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$ accepting L can be constructed as such:

- $Q = \{q_p | p \in P\}$ where $P = \prod_{v_i \in V} \{d_i, d'_i\}$.
- $\rho = \{(c, q_p) | c \in C(p) \; and \; p \in P\}$ where $C(p) = \prod_{v_j \in V} p[j]$ is the set of all complete assignments over (V, p).
  (Follows from the definition of ρ -i.e. ρ is a function and covers all complete assignments over (V, D)- and from Proposition 2.3.1.)
- $\delta = \delta_1 \cup \delta_2$ where δ_1 and δ_2 are defined as such:
  $\delta_1 = \{(q_p, v_j \leftarrow val, q_{p'}) \mid q_p, q_{p'} \in Q \; \wedge \; q_p \neq q_{p'} \; \wedge \; val \in p'[j] \; \wedge \; \forall k \neq j . p[k] = p'[k] \; \wedge \; k, j \in [1, |V|]\}$
  $\delta_2 = \{(q_p, v_j \leftarrow val, q_p) \mid val \in p[j] \; \wedge \; j \in [1, |V|] \; \wedge \; q_p \in Q\}$
  (δ_1 follows from Proposition 2.2.3 and δ_2 follows from Proposition 2.2.5)
- $E = \{q_e\}$, given $e = \prod_{v_i \in V} d'_i$ (i.e. $\rho^{-1}(q_e) = \prod_{v_i \in V} d'_i$)
- $s = q_p$ (i.e. $\rho^{-1}(s) = p$)

At this point, so as to prove the minimality of M, showing that any two states $q, q' \in Q$ are distinguishable from each other is sufficient. In fact, since ρ must cover all complete assignments over (V, D) by its definition, and since for any pair of states q, q' and each index i, either $\rho^{-1}(q)[i] \cap \rho^{-1}(q')[i] = \emptyset$ or $\rho^{-1}(q)[i] = \rho^{-1}(q')[i]$ must hold according to Proposition 2.3.1, there is no other possible construction that can satisfy the condition that there is single error state. Even so, a distinguishability-based

proof is also provided:

Please observe that, in M, for any pair of states $q, q' \in Q$ where $q \neq q'$, $\rho^{-1}(q)[i] \neq \rho^{-1}(q')[i]$ holds at least for one i. That is, there exists at least one i for which either $\rho^{-1}(q)[i] = d_i$ and $\rho^{-1}(q')[i] = d'_i$ or $\rho^{-1}(q)[i] = d'_i$ and $\rho^{-1}(q')[i] = d_i$ holds. Assume the former; then, any sequence w that produces a partial run that starts from q (i.e. starts with the configuration (q, w)) and ends up at the error state must include an assignment $v_i \leftarrow val'_i$ where $val'_i \in d'_i$, whereas there exist some sequences w' that do not include such an assignment but produce a partial run that starts from q' (i.e. starts with the configuration (q', w')) and ends up at the error state. When the latter is assumed, the same proof with q and q' interchanged is valid. That is, any two states $q, q' \in Q$ are distinguishable from each other. Therefore, no two (or more) states of M can be merged into a single state.

In conclusion, since either $\rho^{-1}(q)[i] = d_i$ or $\rho^{-1}(q)[i] = d'_i$ necessarily holds for each q and each i, and since all of the states of M are distinguishable from each other, M is necessarily minimal.■

Below are some example fundamental FA-VAs. Those examples use the shorthand notation $v_i \leftarrow d_i$ (where $v_i \in V$ and $d_i \subseteq D(v_i)$), which represents a set of alphabet symbols $\Sigma_i \subset \Sigma$ such that $\Sigma_i = \{\text{“}a[1] \leftarrow a[2]\text{”} \mid a \in \{v_i\} \times d_i\}$. In brief, $v_i \leftarrow d_i$ represents all assignments $v_i \leftarrow val_i$ where $val_i \in d_i$. Moreover, in graphical notation, each state q is labeled not by its name q but by $\rho^{-1}(q)$. To illustrate, say $\rho^{-1}(q) = d_1 \times d_2$; then, the state q has the label $d_1 \times d_2$ on it. Lastly, error states are indicated via a square inside the circle representing the state.

Example 2.3.1. Given $L_1 = (V, D, \Sigma, p, A')$ where $V = \{v_1\}$, $D = \{v_1 \rightarrow d_1 \cup d'_1\}$ (s.t. $d_1 \cap d'_1 = \emptyset$), $\Sigma = \{v_1 \leftarrow d_1, v_1 \leftarrow d'_1\}$, $p = val_i$ such that $val_i \in d_i$, and $A' = d'_i$; the fundamental FA-VA M_1 accepting L_1 is

$M_1 = (Q, s, \Sigma, \delta, E, V, D, \rho)$ such that
$Q = \{q_0, q_1\}$, $s = q_0$, $E = \{q_0\}$,
$\delta = \{(q_0, v_1 \leftarrow d_1, q_0), (q_0, v_1 \leftarrow d'_1, q_1), (q_1, v_1 \leftarrow d_1, q_0), (q_1, v_1 \leftarrow d'_1, q_1)\}$,
$\rho = \{(c, q_0) | c \in d_1\} \cup \{(c, q_1) | c \in d'_1\}$.

M_1 in graphical notation is given at Figure 2.1.

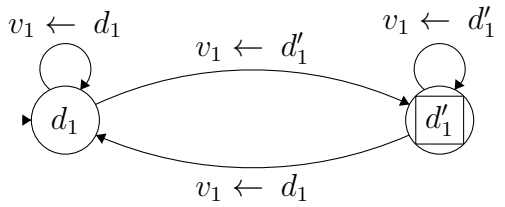


Figure 2.1: M_1

Example 2.3.2. Given $L_2 = (V, D, \Sigma, p, A')$ where
$V = \{v_1, v_2\}$,
$D = \{v_1 \to d_1 \cup d_1', v_2 \to d_2 \cup d_2'\}$ (s.t. $d_i \cap d_i' = \emptyset$ for each i),
$\Sigma = \{v_1 \leftarrow d_1, v_1 \leftarrow d_1', v_2 \leftarrow d_2, v_2 \leftarrow d_2'\}$,
$p = val_i$ such that $val_i \in d_1 \times d_2$,
$A' = d_1' \times d_2'$;
the fundamental FA-VA M_2 accepting L_2 is $M_2 = (Q, s, \Sigma, \delta, E, V, D, \rho)$ such that

$Q = \{q_0, q_1, q_2, q_{11}\}$, $s = q_0$, $E = \{q_{11}\}$,
$\delta = \{(q_0, v_1 \leftarrow d_1, q_0), (q_0, v_1 \leftarrow d_1', q_1), (q_0, v_2 \leftarrow d_2, q_0), (q_0, v_2 \leftarrow d_2', q_2)$
$(q_1, v_1 \leftarrow d_1, q_0), (q_1, v_1 \leftarrow d_1', q_1), (q_1, v_2 \leftarrow d_2, q_1), (q_1, v_2 \leftarrow d_2', q_{11})$
$(q_2, v_1 \leftarrow d_1, q_2), (q_2, v_1 \leftarrow d_1', q_{11}), (q_2, v_2 \leftarrow d_2, q_0), (q_2, v_2 \leftarrow d_2', q_2)$
$(q_{11}, v_1 \leftarrow d_1, q_2), (q_{11}, v_1 \leftarrow d_1', q_{11}), (q_{11}, v_2 \leftarrow d_2, q_1), (q_{11}, v_2 \leftarrow d_2', q_{11})\}$,
$\rho = \{(c, q_0) | c \in d_1 \times d_2\} \cup \{(c, q_1) | c \in d_1' \times d_2\}$
$\cup \{(c, q_2) | c \in d_1 \times d_2'\} \cup \{(c, q_{11}) | c \in d_1' \times d_2'\}$.

M_2 in graphical notation is given at Figure 2.2.

Theorem 2.3.3. *For each fundamental FA-VA $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$, there exists a minimal deterministic finite automaton (DFA) M' such that $L(M) = L(M')$.*

Proof. Consider the DFA [1] $M'' = (Q, \Sigma, \delta', s, F)$ such that $F = E$ and $\delta' = \delta_1 \cup \delta_2$ where $\delta_1 = \{(q_e, \sigma, q_e) |\ q_e \in E\ and\ (q_e, \sigma, q') \in \delta\ for\ some\ q' \in Q\}\}$, and $\delta_2 = \{(q, \sigma, q') |\ q \notin E\ and\ (q, \sigma, q') \in \delta\}$. That is, ρ is discarded, each transition (q_e, σ, q) outgoing from the error state q_e are modified such that they end up at q_e itself (i.e. changed to (q_e, σ, q_e)) and the error state is selected as the final state.
The language L'' accepted by M'' is the complement of the language L accepted by M, since all of the accepting runs (i.e the runs starting with (s, w) and not visiting the

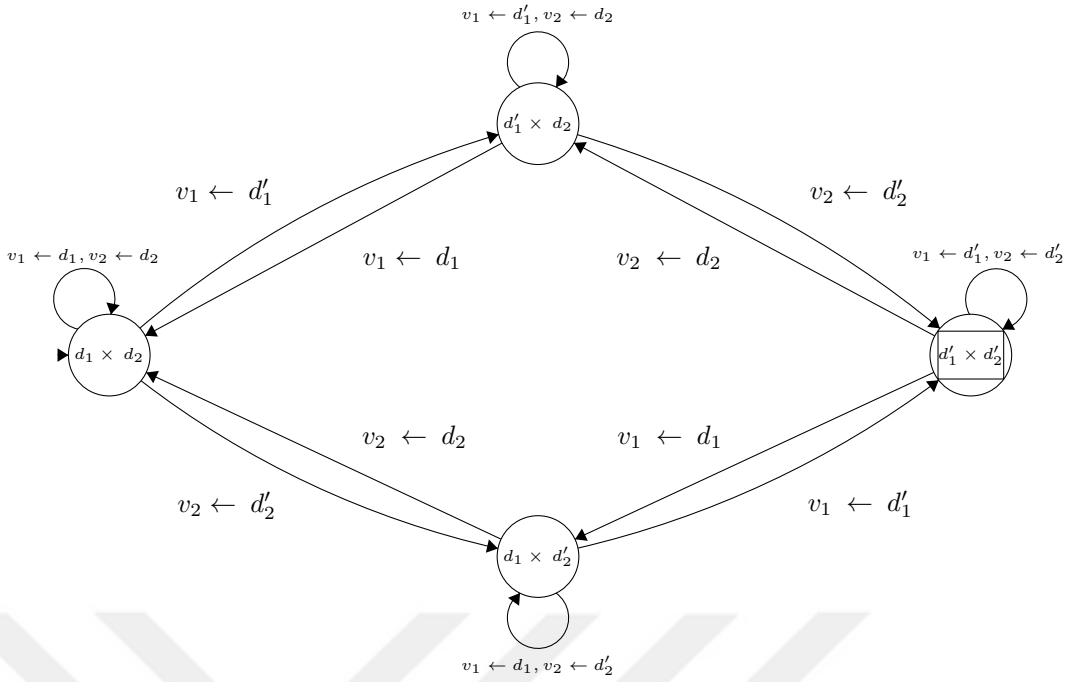


Figure 2.2: M_2

error state) end up at a non-final state of M'' and all of the non-accepting runs (i.e the runs starting with (s, w') and visiting the error state at least once) end up at the final state of M''.

Furthermore, M'' is minimal since in the proof of the Lemma 2.3.2 it is already shown that for any pair of states $q, q' \in Q$ where $q \neq q'$ there exists at least one variable v_i such that any sequence w that produces a partial run that starts from q (i.e. starts with the configuration (q, w)) and ends up at the error state (i.e. the final state of M'') must include an assignment $v_i \leftarrow val_i'$ where $val_i' \in d_i'$ (s.t. $d_i' = \rho^{-1}(q_e)[i]$, q_e is the error state), whereas there exist some sequences w' that do not include such an assignment but produce a partial run that starts from q' (i.e. starts with the configuration (q', w')) and ends up at the error state (i.e. the final state of M'').

Then there exist another DFA $M' = (Q, \Sigma, \delta', s, F')$ where $F' = Q \backslash F$ which accepts the complement of the language accepted by M'' (i.e. M'' is obtained by swapping final and non-final states of M) [1]. That is, the language M' accepts is L.

The DFA obtained by swapping final and non-final states of a minimal DFA is also minimal which can be shown as such: There exist a minimal DFA D. D' is the DFA obtained by swapping final and non-final states of D, therefore $L(D')$ is the comple-

ment of $L(D)$. Assume D' is not minimal. Then, there must be a D'' with fewer states than D such that $L(D'') = L(D')$. Since $L(D'') = L(D')$ is the complement of $L(D)$, the complement of D'', say D''', which can be obtained by swapping final and non-final states of D'' must be an acceptor of $L(D)$. Then, D''' has fewer states than D and $L(D''') = L(D)$, which yields contradiction since it is initially given that D is minimal. The assumption that D' is not minimal is incorrect.
Thus, it can be concluded that M' is a minimal DFA and $L(M')$=$L(M)$.■

Theorem 2.3.4. *For each variable assignment language $L = (V, D, \Sigma, p, A')$, there exists at least one collection S of fundamental FA-VAs such that S accepts L if and only if each fundamental FA-VA $M \in S$ accepts L.*

Proof. There exist at least one set $X = \{X_1, X_2, ..., X_n\}$ such that $\bigcup_{X_j \in X} X_j = A'$ and each $X_j \in X$ equals $\prod_{v_i \in V} d'_{ij}$ where $d'_{ij} \subseteq D(v_i)$ (at the worst case,
$X = \{\{c\} | c \in A'\}$). Moreover, there exist a procedure that computes that set X, which can be shown as such: Since each $D(v_i)$ is finite, there are finite number of subsets of each $D(v_i)$ (i.e. number of possible different d'_{ij}s is finite for each i). V is also finite. Then, the set $X' = \{\prod_{v_i \in V} d'_i | v_i \in V \ \wedge \ d'_i \subseteq D(v_i)\}$ is a finite set with finite elements. Thus, the powerset of X' is also a finite set with finite elements. Therefore, the number of subsets of the powerset of X' is also finite; furthermore, each of those subsets are finite sets with finite elements. Then, since X is a subset of the powerset of X' by definition, there exists a procedure that finds X in finite time, at the worst scenario, by checking each subset of the powerset of X' one by one.

Consider such a collection of variable assignment languages:
$I = \{L_j | L_j = (V, D, \Sigma, p, X_j) \ \wedge \ X_j \in X\}$.
For each $L_j \in I$, there exist a fundamental FA-VA M_j since each $X_j = \prod_{1 \leq i \leq |V|} d'_i$ where $d'_i \subseteq D(v_i)$. Showing that a sequence w is accepted by M if and only if it is accepted by each E_j proves the theorem, which can be done as such:
Consider the maximal FA-VA M' accepting L and each maximal FA-VA M'_j accepting L_j. For each of those FA-VAs; $Q, \Sigma, \delta, V, D, \rho$ are the same. Further, each have the same initial state $s \in Q$ such that $\rho^{-1}(s) = p$. That is, those maximal FA-VAs differ from each other only in their sets of error states as such: $E' = \{q | \rho^{-1}(q) \in A'\}$ where E' is the error state of M' and $E_j = \{q | \rho^{-1}(q) \in X_j\}$. That is, $E' = \bigcup_{j \in X_j} E'_j$ since $A' = \bigcup_{j \in X_j} \{X_j\}$ and ρ is a bijection between the set of states and the set of

complete assignment over (V, D) (for the case of maximal FA-VA). Then, any initial configuration (s, w) produces the same run on each of those maximal FA-VAs. Importantly, since $E' = \bigcup_{j \in X_j} E'_j$, if a run visits an error state on M', it also visits an error state on at least one of the M_j's; moreover, if a run does not visit an error state on M', it does not visit an error state on any of the M_j's. That is, a sequence w is accepted by M' if and only if it is accepted by each E'_j. ■

Lastly, the following proposition underlines a valuable set theoretic / order theoretic property of fundamental FA-VAs:

Proposition 2.3.5. *For each fundamental FA-VA $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$, the set of states Q is a partially ordered set.*

Proof. For a fundamental FA-VA $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$, given $d'_i = \rho^{-1}(q_e)[i]$ for each i where q_e is the error state, given and $d_i = D(v_i) \backslash d'_i$ for each i, ρ^{-1} is a bijection between Q and P where $P = \prod_{v_i \in V} \{d_i, d'_i\}$, as can be quickly verified by checking the construction at the proof of the Lemma 2.3.2. Moreover, where $K = \{d_i | v_i \in V\}$, there is another obvious bijection, say f, between P and the powerset of K such that $f : P \to K^2$, $f(p) = \bigcup_{1 \leq i \leq |V|} g(p[i])$ where $g(p[i])$ yields $\{p[i]\}$ if $p[i] = d_i$ and yields $\emptyset$ if $p[i] = d'_i$. Then, $f(\rho^{-1}(q))$ is also a bijection mapping each state $q \in Q$ to a subset k_2 of K (i.e. to a $k_2 \in K^2$). Then, since elements of K^2 are partially ordered under subset relation, and since $f(\rho^{-1}(q))$ is a bijection between Q and K^2, elements of Q are also partially ordered under the same relation. ■

CHAPTER 3

ALGORITHMIC INFERENCE

3.1 Overview

We will consider an Angluin-style [6] learning scenario. In difference with the Angluin's learning setting, in which an algorithmic Learner infers the system under learning (SUL) by making membership and equivalence queries to a Teacher which knows the SUL (and processing Teacher's replies to those queries) [6], we will employ a more concrete setting where queries are directly made to the SUL. Moreover, a single type of query, **legality queries**, will be used. We will show that, given the set of strings accepted by the SUL is equivalent to a variable assignment language $L(SUL) = (V, D, \Sigma, p, A')$, and given V, D, and one complete assignment c such that $\rho(c) = s$ (where s is the starting state), are initially known, there is an algorithmic Learner able to infer the minimal FA-VA M representing the SUL (i.e. $L(M) = L(SUL)$).

Given $L(SUL) = (V, D, \Sigma, p, A')$, a *legality query* consists a single complete assignment c over (V, D) sent by the Learner to the system under learning. The SUL replies *True* if c is a legal assignment; replies *False* if c is an illegal assignment. It is assumed that the SUL has a function, represented as $LEG(c)$ where c is a complete assignment, that replies legality queries. Moreover, so as to be able to form those queries, the Learner must initially know V and D. At least one complete assignment from $\rho^{-1}(s)$ is also needed to be able to mark the initial state at the end. Apart from these, the Learner does not need an additional information or additional sources of information about the SUL.

What the inference algorithms mainly does is first constructing an observation table,

final form of which keeps all legal complete assignments and the complete assignments reached by applying each alphabet symbol to each of those legal complete assignments; and then partitioning the domains of each variable by considering the groups of values which produce an illegal assignment when applied to the same complete assignments (i.e. for each value pair val_1, val_2 of the variable v_i, the algorithm implicitly checks if $\alpha(c, i, val_2)$ is also an illegal assignment for each the complete assignments c such that $\alpha(c, i, val_1)$ is an illegal assignment).

3.1.1 Observation Tables

Even though the concept of observation table is borrowed from Angluin [6], the type of observation table that will be used in this context is specifically designed according to the characterizing properties of FA-VAs. Therefore, before exploring the inference algorithm, let us formally define the concept of observation table, which will be the central data structure of the inference algorithm:

Formally, an observation table is a quadruple $O = (C, E, T, X)$ where C is the set of all legal complete assignments over (V, D), $E = \bigcup_{v_i \in V}(\{v_i\} \times D(v_i))$ which is equivalent to the alphabet (i.e. it is the alphabet elements without the explicitly shown variable assignment operator " $\leftarrow$ "), T is a function that maps each element $(c, (v_i, val))$ of $C \times E$ to $c' = \alpha(c, i, val)$, and X is a function from $C \times E$ to $\{True, False\}$ such that $X(c, (v_i, val)) = LEG(T(c, (v_i, val)))$. In the visual table representation that will be used in examples, each row will correspond to a complete assignment $c \in C$, each column will correspond to a $(v_i, val) \in E$, and each cell will contain a complete assignment $c' = T(c, (v_i, val)) = \alpha(c, i, val)$. For the ease of perception, the values of $X(c, (v_i, val)) = LEG(c')$ (where $c' = \alpha(c, i, val)$) will be represented by the background color of the cell in which c' resides: The background will be *white* if $X(c, (v_i, val)) = True$ (i.e. if c' is a legal complete assignment), it will be *red* if $X(c, (v_i, val)) = False$ (i.e. if c' is an illegal complete assignment). To illustrate, Table 3.1 below depicts the final form of the observation table that will be generated by the baseline inference algorithm while learning a SUL equivalent to a very simple variable assignment language L_{simple} with 2 variables each having 2 possible values, and with only 1 erroneous assignment such that $L_{simple} = (V, D, \Sigma, p, A')$

where $V = \{v_1, v_2\}$, $D(v_1) = \{1, 2\}$, $D(v_2) = \{a, b\}$, $A' = \{(2, b)\}$, $p = (1, a)$. Since the inference algorithm only stores rows corresponding to the legal complete assignments, the example observation table does not have a row corresponding to the single illegal complete assignment of the example language.

Table 3.1: Observation Table for L_{simple}

	v_1		v_2	
	1	2	a	b
$(1, a)$	$(1, a)$	$(2, a)$	$(1, a)$	$(1, b)$
$(2, a)$	$(1, a)$	$(2, a)$	$(2, a)$	$(2, b)$
$(1, b)$	$(1, b)$	$(2, b)$	$(1, a)$	$(1, b)$

3.2 Inference Algorithm

Before exploring the algorithm, let us prove the following proposition that is needed to ensure the correctness of the algorithm:

Proposition 3.2.1. *Given a FA-VA $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$, if there exists a non-error state $q \in Q \backslash E$ that cannot be reached from the starting state s without visiting an error state, adding q to E (i.e. converting q to an error state) does not affect the language accepted by M.*

Proof. The state q being not reachable from s without visiting an error state directly implies that each run of M that visits q must have already visited an error state. Therefore, each run visiting q is an erroneous run regardless of q being an error state or not. That is, adding q to E does not alter the set of sequences that produce an accepting run.■

Proposition 3.2.1 ensures that the inference algorithm traverses each of the non-error states (that affect the language accepted by the system), as will be underlined in the following.

3.2.1 LearnFAVA

Algorithm 1: LearnFAVA - Baseline FA-VA Inference Algorithm

```
1  Algorithm LearnFAVA(V, D, c_in, LEG)
2      initialize empty table O, empty list<compl> newRows;
3      add c_in to newRows;
4      foreach compl c ∈ newRows do
5          initialize empty row row_c;
6          row_c.label ← c;
7          foreach variable v_i ∈ V do
8              foreach value val ∈ D(v_i) do
9                  compl c' ← α(c, i, val);
10                 row_c[(v_i, val)] ← c';
11                 if LEG(c') == True then
12                     X(row_c.label, (v_i, val)) ← True;
13                     if c' ∉ newRows then
14                         add c' to newRows;
15                     end
16                 else
17                     X(row_c.label, (v_i, val)) ← False;
18                 end
19             end
20             add row_c to O;
21         end
22     end
23     initialize empty list<set<set<value>>> partitions;
24     foreach variable v_i ∈ V do
25         initialize empty set<set<value>> P_i;
26         int j ← 1;
27         while j ≤ |D(v_i)| do
28             value val ← D(v_i)[j];
29             initialize empty set<value> part;
30             add val to part;
31             foreach value val' ∈ D(v_i)[j + 1 :] do
32                 if X(v_i, val) == X(v_i, val') then
33                     add val' to part;
34                     delete val' from D(v_i);
35                 end
36             end
37             add part to P_i;
38             delete part;
39             j ← j + 1;
40         end
41         append P_i to partitions;
42     end
43     return completeFAVA(generateSemiFAVA(V, D, partitions), c_in, LEG);
```

The algorithm LearnFAVA, pseudocode of which is given at Algorithm 1, is the baseline FA-VA inference algorithm. In the pseudocode, in addition to the basic data

types, some special data types are also used. These are $variable$, $value$, $compl$ (stands for complete assignment), $table$ (stands for observation table), and row (stands for a row of the observation table). Furthermore, $row.label$ denotes the complete assignment $c \in C$ to which the row row corresponds, $row[(v, val)]$ denotes the value of $T(c, (v, val))$ for the row row, $X(row.label, (v, val))$ denotes the value of $X(c, (v, val))$ for the row row (where $row.label = c$). Additionally, $X(v, val)$ denotes the bit sequence consisting of the values of X at the column corresponding to (v, val) (value of the corresponding bit is 0 for $False$, 1 for $True$). To illustrate, at the Table 3.1, $X(v_1, 2) =$ "110" and $X(v_2, b) =$ "101"; at the Table 3.2, $X(v_1, 11) =$ "1111111111" and $X(v_2, 3) =$ "1111110000". Lastly, $len(list)$ returns the number of elements of the list (or ordered set) $list$; and $list[k:]$ denotes the sublist of the list (or ordered set) $list$ that starts from the k^{th} index (inclusive) and ends at the last element of the $list$ (inclusive). If $k > len(list)$, $list[k:]$ returns the empty list.

The algorithm requires the following inputs: set of variables V, mapping/dictionary from variables to domains D, an initial complete assignment c_{in} (s.t. $c_{in} \in \rho^{-1}(s)$ where s is the starting state), the function LEG of the SUL. At the end, the algorithm outputs the minimal FA-VA representing the SUL.

How does LearnFAVA work?: It starts with filling the row corresponding to the c_{in} and then adds the filled row to the observation table. That is the first iteration of the outer loop between the lines 4-22. At the end of this first iteration, all legal complete assignments that are reachable in one step from c_{in} (i.e. each c' such that $(\rho(c_{in}), v_i \leftarrow val, \rho(c')) \in \delta$) in the maximal automaton equivalent to the SUL are added to the list $newRows$. The list $newRows$ holds the complete assignments for which a row will be added to the observation table by the later iterations of the outer loop. That outer loop;

1. Fills the rows corresponding to the complete assignments added to the list $newRows$ at its previous iterations (lines 7,8,9,10,12,17),

2. Adds legal complete assignments that are produced while filling the current row (i.e. that are one-step reachable from the current complete assignment) to the $newRows$ (lines 11-15).

Importantly, that outer loop terminates when all of the non-error states of the maximal

FA-VA equivalent of the SUL are added to the observation table. That is, each legal complete assignment that affects the set of sequences accepted by the SUL and thus by the FA-VA that is being inferred is certainly added to the observation table due to the Proposition 3.2.1.

The loop between the lines 27-40 partitions the domain of the variable v_i into the list P_i (for the given i) such that none of the elements of P_i, which are subsets of $D(v_i)$, intersect each other at the end of the partitioning process and $\bigcup_{part \in P_i} part = D(v_i)$. The outer loop between the lines 24-42 makes that loop (i.e. the loop between the lines 27-40) repeat that partitioning process for each variable in V. Ultimately, these loops partition the domain of each variable on such a basis: Any two value $val_1, val_2 \in D(v_i)$ are in the same subdomain (i.e. in the same $part$ of P_i) at the end if and only if $LEG(\alpha(c, i, val_1)) = LEG(\alpha(c, i, val_2))$ holds for each legal complete assignment c. This is implemented by comparing the bit sequences $X(v_i, val_1)$ and $X(v_i, val_2)$ and adding those two values to the same $part$ if and only if $X(v_i, val_2)$ and $X(v_i, val_2)$ are equal (lines 28-36). The reasoning behind that is as such:

If $LEG(\alpha(c, i, val_1)) = LEG(\alpha(c, i, val_2)) = ... = LEG(\alpha(c, i, val_n)) = False$ for a distinct complete assignment c, then at any FA-VA representing the SUL, there exist either one error state q_e such that $\{val_1, val_2, ..., val_n\} = \rho^{-1}(q_e)[i]$ or more than one error states, in essence a set of error states Q_e, such that $\{val_1, val_2, ..., val_n\} = \bigcup_{q_e \in Q_e} \rho^{-1}(q_e)[i]$ (by the Lemma 2.2.8 and the Proposition 2.3.1). However, if for another legal complete assignment c', any two of those values, say val_1 and val_2, yield $LEG(\alpha(c', i, val_1)) = True$ and $LEG(\alpha(c', i, val_2)) = False$, then at any FA-VA representing the SUL there exists two states $q \neq q'_e$ such that $\alpha(c', i, val_1)) \in \rho^{-1}(q)[i]$ and $\alpha(c', i, val_2)) \in \rho^{-1}(q'_e)[i]$. Thus, by the Proposition 2.3.1 and Proposition 2.2.6, val_1 and val_2 cannot be in the same subdomain of v_i. Furthermore, since each P_i produced by the algorithm is a disjoint union of subdomains, the algorithm guarantees that each subdomain contained by each P_i is equivalent to $\rho^{-1}(q)[i]$ for some state q of the FA-VA being inferred (by Proposition 2.3.1).

Once the process of partitioning the domains is completed, the procedure **generateSemiFAVA** constructs and returns the semi-FA-VA (analogous to semiautomaton [18]) underlying the FA-VA representing the SUL (i.e. it constructs Q, δ, ρ, but does

not determine s and E), according to the following construction:

- $Q = \{q_p | p \in P\}$ where $P = \prod_{v_i \in V} partitions[i]$.
- $\rho = \{(c, q_p) | c \in C(p) \; and \; p \in P\}$ where $C(p) = \prod_{v_j \in V} p[j]$ is the set of all complete assignments over (V, p).
  (Follows from the definition of ρ -i.e ρ is a function and covers its domain which is the set of all complete assignments over (V, D)- and Proposition 2.3.1)
- $\delta = \delta_1 \cup \delta_2$ where δ_1 and δ_2 are defined as such:
  $\delta_1 = \{(q_p, v_j \leftarrow val, q_{p'}) \mid q_p, q_{p'} \in Q \; \wedge \; q_p \neq q_{p'} \; \wedge \; val \in p'[j]$
  $\wedge \; \forall k \neq j . p[k] = p'[k] \; \wedge \; k, j \in [1, |V|]\}$
  $\delta_2 = \{(q_p, v_j \leftarrow val, q_p) \mid val \in p[j] \; \wedge \; j \in [1, |V|] \; \wedge \; q_p \in Q\}$
  (δ_1 follows from Proposition 2.2.3 and δ_2 follows from Proposition 2.2.5)

Lastly, the procedure **completeFAVA** takes c_{in}, LEG and the semi-FA-VA constructed by the procedure generateSemiFAVA and determines the starting state and the error states by such a simple operation:

$s \leftarrow \rho(c_{in})$.
Initialize an empty set E.
For each state $q \in Q$:
  Randomly select 1 element $c \in \rho^{-1}(q)$,
  if $LEG(c) == no$, *add* q *to* E.
return $(Q, s, \Sigma, \delta, E, V, D, \rho)$.

Notice that there is an obvious one-to-one correspondence with the FA-VA $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$ returned by the algorithm LearnFAVA (i.e. the FA-VA constructed by completeFAVA and generateSemiFAVA) and the maximal FA-VA $M' = (Q, s, \Sigma', \delta', E, V, D', \rho')$ such that $D'(v_i) = partitions[i]$ or equivalently, $D'(v_i) = \{d \mid d \in partitions[i]\}$, and ρ' and δ' are constructed according to that definition of $D'(v_i)$, by following the construction given in the proof of the Proposition 2.2.1.

In simpler terms, in the FA-VA M constructed by the algorithm LearnFAVA, all values that are elements of the same element of a partition (i.e. all elements of a d where $d \in partitions[i]$) "show the same behavior". That is, for any such d, if $val_1 \in \rho^{-1}(q)[i]$

for some $val_1 \in d$, then $val_2 \in \rho^{-1}(q)[i]$ also holds for every other $val_2 \in d$ (which is, in fact, necessary due to the Proposition 2.3.1); and if $(q, v_i \leftarrow val_1, q') \in \delta$ for some $val_1 \in d$, then $(q, v_i \leftarrow val_2, q') \in \delta$ also holds for every other $val_2 \in d$ (which is also necessary due to the propositions 2.3.1, 2.2.6 and 2.2.3.).

Importantly, in M' and M, no subset Q' of Q can be merged into a single state. Assume there is such a Q' that can be merged into a single state q_m. That is, in fact, equivalent to creating a state q_m such that $\rho^{-1}(q_m) = \bigcup_{q' \in Q'} \rho^{-1}(q')$. Then, for each i, $\rho^{-1}(q_m)[i] = \bigcup_{q' \in Q'} \rho^{-1}(q')[i]$. This implies, at least for one i, $\rho^{-1}(q_m)[i]$ equals the union of more than one $d_i \in partitions[i]$ since $\rho^{-1}(q) \neq \rho^{-1}(q')$ by the definition of ρ. However, it is previously shown that for no error state q_e of M (and thus, of M') $\rho^{-1}(q_e)[i]$ can be equal to the union of more than one $d_i \in partitions[i]$. Therefore, by the Proposition 2.3.1, no subset Q' of Q can be merged into a single state, which concludes that M is minimal.

Lastly, we can easily conclude that the algorithm LearnFAVA eventually halts (in case the SUL to be inferred is representable by a FA-VA) as such:

The first outer loop between the lines 4-22 iterates exactly $|C'|$ times (C' is the set of all legal assignment over the given (V, D)) whereas the innermost loop between the lines 8-19 iterates exactly $|\Sigma|$ times for each iteration of the outer loop. $|C'| \times |\Sigma|$ is finite since C' and Σ are finite, therefore, this part of the algorithm eventually terminates. Additionally and importantly, in this part (i.e. while constructing the observation table), the algorithm makes $|C'| \times |\Sigma|$ legality queries (i.e. one legality query for each cell of the table).

The loop between the lines 24-42 also eventually terminates, which can be shown as such: Its inner loop (lines 27-40) iterates $(|D(v_i)|^2 - |D(v_i)|)/2$ times at the worst case, which is the case for no pair of values $val_1, val_2 \in D(v_i)$ $X(v_i, val_1) = X(v_i, val_2)$ holds (i.e. $P_i = \{\{val\} \mid val \in D(v_i)\}$). Therefore, the loop between the lines 24-42 terminates after at most $\sum_{v_i \in V}(|D(v_i)|^2 - |D(v_i)|)/2$ iterations. This is bounded from above by $|V| \times (|D(v_b)|^2 - |D(v_b)|)/2$ where $D(v_b)$ is the domain with the greatest cardinality. Since each of the domains and V are finite, $|V| \times (|D(v_b)|^2 - |D(v_b)|)/2$ is finite.

Moreover, the function generateSemiFAVA obviously requires finite time since it constructs finite number of finite sets from finite input. In more detail, that function can be implemented by first constructing a table which maps each $c \in C$ (where $C = \prod_{v_i \in V} D(v_i)$) to the corresponding element of P where $P = \prod_{v_i \in V} partitions[i]$, following the order dictated by the Cartesian product $\prod_{v_i \in V} partitions[i]$ (Thus, Q and ρ is constructed). Thereafter, δ can be constructed by checking the state reached by each $(q, v_i \leftarrow val)$ pair (where $q \in Q$ and $v_i \leftarrow val \in \Sigma$) from the table (i.e. by finding $c' = \alpha(c, i, val)$ and $\rho(c')$ directly looking to the position at which it resides where c is a random element of $\rho^{-1}(q)$). Such an implementation of the function generateSemiFAVA, constructs the correct semi-FA-VA by doing at most $|C| + |C| \times |\Sigma|$ operations ($|C|$ for constructing the table, $|C| \times |\Sigma|$ for constructing the δ).

Finally, the function completeFAVA traverses the (finite) set of states of the inferred minimal semi-FA-VA and makes one legality query for each state; therefore, it also terminates in finite time ($|C|$ iterations and $|C|$ legality queries at the worst case).

In parallel, it can be concluded that the algorithm LearnFAVA terminates after at most $(|C'| \times |\Sigma|) + (|C'| \times \sum_{v_i \in V}(|D(v_i)|^2 - |D(v_i)|)/2) + (|C| \times (|\Sigma| + 1)) + |C|$ steps; moreover, it makes at most $(|C'| \times |\Sigma|) + |C|$ legality queries.

The content presented in this subsection up to this point can be summarized in two theorems:

Theorem 3.2.2. *For each variable assignment language L, there exist a minimal FA-VA M, such that $L(M) = L$.*

Theorem 3.2.3. *For any SUL that can be represented by a FA-VA, the algorithm LearnFAVA infers the minimal FA-VA $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$ such that $L(M) = L(SUL)$ in polynomial time if it is provided with V, D, one complete assignment c_{in} such that $\rho(c_{in}) = s$, and access to the function LEG of the SUL.*

There are two further remarks about the algorithm LearnFAVA which are worth mentioning:

First, there is a simple bijection between the observation table constructed by the algorithm and the subset δ'' of the transition relation δ_{max} of the maximal FA-VA representing the SUL, where δ'' is the set of all transitions $(q, \sigma, q') \in \delta_{max}$ such that

q is not an error state (i.e. $\rho^{-1}(q) \not\subseteq A'$). Therefore, the algorithm LearnFAVA can easily be adapted to be used for state minimization.

Second, the observation table generated by the algorithm, in fact, is not required to be wholly stored in the memory. The algorithm anyway follows the logic of constructing an observation table. In other words, it implicitly constructs an observation table that is either stored in the memory or not. However, after the table is filled, the algorithm does only check the values of X (i.e. T is not checked). That is, once an iteration of the loop which fills rows of the table is completed, the whole row filled in that iteration does not need to be stored; storing explored values of X is sufficient. While filling each row, the data required to continue to explore other rows of the table are simultaneously appended to the list $newRows$.

Finally, let us conclude this section with an example that illustrates an operation of LearnFAVA:

Example 3.2.1. The observation table generated by LearnFAVA while modelling a SUL such that $L(SUL) = (V, D, \Sigma, p, A')$ where

$V = \{v_1, v_2, v_3\}$,
$D = \{v_1 \to \{11, 12, 21, 22\}, v_2 \to \{2, 3, 4\}, v_3 \to \{1, 2\}\}$,
$p = (11, 3, 1)$,
$A' = \{\{21, 22\} \times \{2, 3\} \times \{1, 2\}\ , \{11, 12, 21, 22\} \times \{3, 4\} \times \{2\}\}$;

and the FA-VA M_{SUL} representing the SUL inferred by the algorithm LearnFAVA given $c_{in} = (11, 3, 1)$ are given respectively at Table 3.2 and Figure 3.1. At the graphical representation of the M_{SUL}, self transitions and alphabet symbols read in transitions are not shown. Furthermore, between some rows of the table, empty rows with the text "next iteration:" are placed, which indicates the outer loop constructing the table (i.e. the loop between the lines 5-29 of the pseudocode) begins a new iteration at that point.

In tuple notation, the FA-VA is $M = (Q, s, \Sigma, \delta, E, V, D, \rho)$ where

$Q = \{q_{121}, q_{131}, q_{141}, q_{122}, q_{132}, q_{142}, q_{221}, q_{231}, q_{241}, q_{222}, q_{232}, q_{242}\}$

$$\rho^{-1} = \{(q_{121}, \{11, 12\} \times \{2\} \times \{1\}), (q_{131}, \{11, 12\} \times \{3\} \times \{1\}),$$
$$(q_{141}, \{11, 12\} \times \{4\} \times \{1\}), (q_{122}, \{11, 12\} \times \{2\} \times \{2\}),$$
$$(q_{132}, \{11, 12\} \times \{3\} \times \{2\}), (q_{142}, \{11, 12\} \times \{4\} \times \{2\}),$$
$$(q_{221}, \{21, 22\} \times \{2\} \times \{1\}), (q_{231}, \{21, 22\} \times \{3\} \times \{1\}),$$
$$(q_{241}, \{21, 22\} \times \{4\} \times \{1\}), (q_{222}, \{21, 22\} \times \{2\} \times \{2\}),$$
$$(q_{232}, \{21, 22\} \times \{3\} \times \{2\}), (q_{242}, \{21, 22\} \times \{4\} \times \{2\}),$$
$$\}$$

$s = \{q_{131}\}$

$E = \{q_{132}, q_{142}, q_{221}, q_{231}, q_{222}, q_{232}, q_{242}\}$

δ is not explicitly provided; it be inferred from ρ^{-1} and the arrows at the graphical representation.

Table 3.2: Observation Table for $L(SUL)$

	v_1				v_2			v_3	
	11	12	21	22	2	3	4	1	2
$(11,3,1)$	$(11,3,1)$	$(12,3,1)$	$(21,3,1)$	$(22,3,1)$	$(11,2,1)$	$(11,3,1)$	$(11,4,1)$	$(11,3,1)$	$(11,3,2)$
next iteration:									
$(12,3,1)$	$(11,3,1)$	$(12,3,1)$	$(21,3,1)$	$(22,3,1)$	$(12,2,1)$	$(12,3,1)$	$(12,4,1)$	$(12,3,1)$	$(12,3,2)$
$(11,2,1)$	$(11,2,1)$	$(12,2,1)$	$(21,2,1)$	$(22,2,1)$	$(11,2,1)$	$(11,3,1)$	$(11,4,1)$	$(11,2,1)$	$(11,2,2)$
$(11,4,1)$	$(11,4,1)$	$(12,4,1)$	$(21,4,1)$	$(22,4,1)$	$(11,2,1)$	$(11,3,1)$	$(11,4,1)$	$(11,4,1)$	$(11,4,2)$
next iteration:									
$(12,2,1)$	$(11,2,1)$	$(12,2,1)$	$(21,2,1)$	$(22,2,1)$	$(12,2,1)$	$(12,3,1)$	$(12,4,1)$	$(12,2,1)$	$(12,2,2)$
$(12,4,1)$	$(11,4,1)$	$(12,4,1)$	$(21,4,1)$	$(22,4,1)$	$(12,2,1)$	$(12,3,1)$	$(12,4,1)$	$(12,4,1)$	$(12,4,2)$
next iteration:									
$(11,2,2)$	$(11,2,2)$	$(12,2,2)$	$(21,2,2)$	$(22,2,2)$	$(11,2,2)$	$(11,3,2)$	$(11,4,2)$	$(11,2,1)$	$(11,2,2)$
next iteration:									
$(21,4,1)$	$(11,4,1)$	$(12,4,1)$	$(21,4,1)$	$(22,4,1)$	$(21,2,1)$	$(21,3,1)$	$(21,4,1)$	$(21,4,1)$	$(21,4,2)$
$(22,4,1)$	$(11,4,1)$	$(12,4,1)$	$(21,4,1)$	$(22,4,1)$	$(22,2,1)$	$(22,3,1)$	$(22,4,1)$	$(22,4,1)$	$(22,4,2)$
next iteration:									
$(12,2,2)$	$(11,2,2)$	$(12,2,2)$	$(21,2,2)$	$(22,2,2)$	$(12,2,2)$	$(12,3,2)$	$(12,4,2)$	$(12,2,1)$	$(12,2,2)$

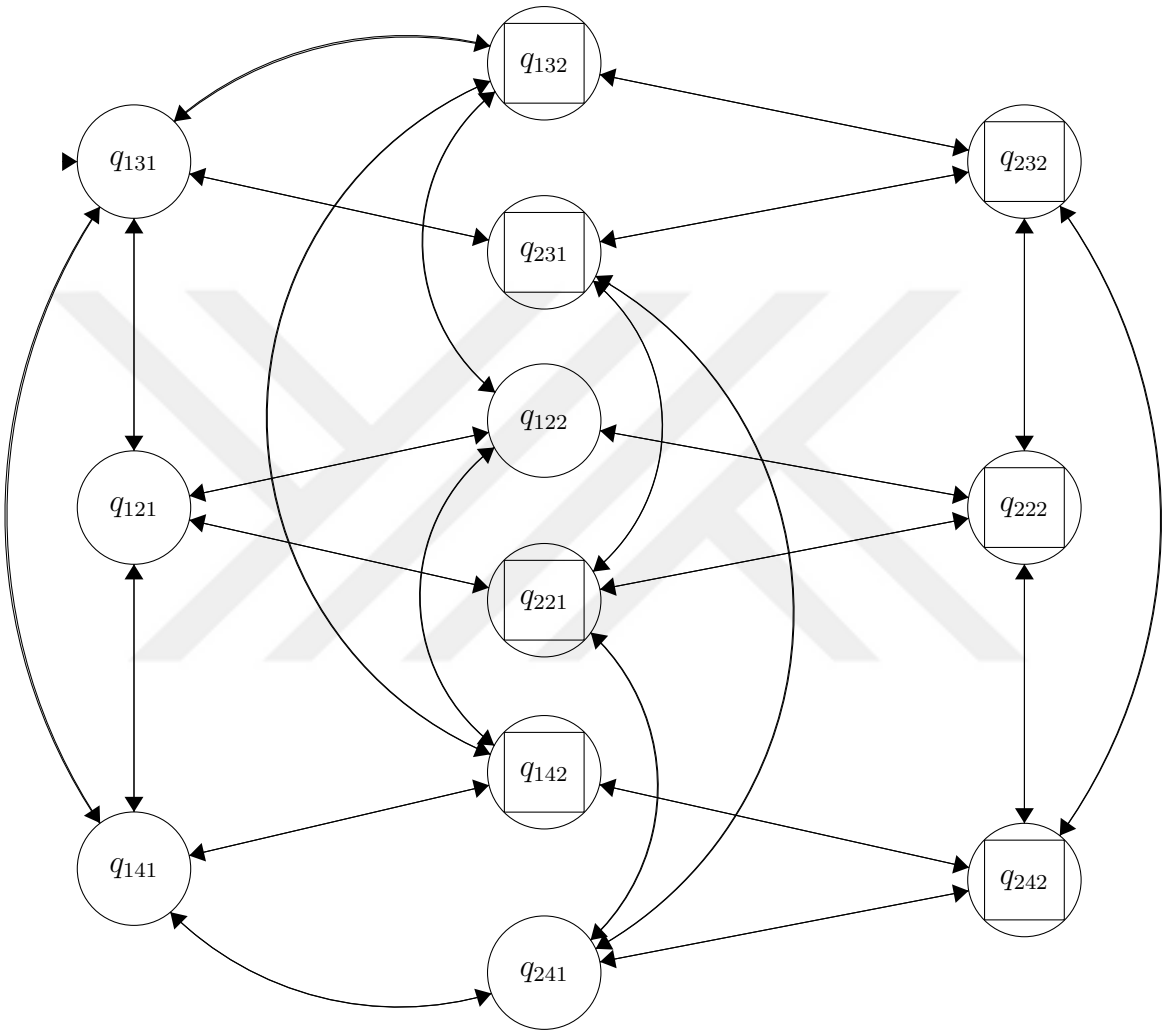


Figure 3.1: M_{SUL}

CHAPTER 4

DISCUSSION

An automata theoretic basis for representing variable assignment constraint systems is presented. Moreover, an active learning algorithm that infers the minimal automata (i.e. FA-VAs) representing those systems is provided.

On the ground of the following properties of FA-VAs, it can be argued that FA-VA-based representations of interactive variable assignment constraint systems are close to be intuitive and provide a basis for easier and broader reasoning about those systems:

- They keep track of both the input sequence and the complete assignments formed after application of each variable assignment in the input sequence. At each state, the effect of each variable assignment defined on the system (i.e. each assignment in the alphabet) is directly observable both in terms of the formed complete assignment and in terms of state change (thus, in terms of getting closer to or farther from one or more error states).
- They employ a compositional and concrete concept of state by means of the ρ^{-1}. This facilitates direct (not sequence-based or diversity-based) conception of each state.
- They form clusters/groups of uniformly behaving complete assignments. For each state, they map the state either to a cluster/group of illegal complete assignments (which are grouped according to the overlap between their effective variables, and to their mergeability into a Cartesian product), or to a cluster/-group of the legal complete assignments (which are grouped according to their difference from the groups of illegal assignments).
- They cluster/categorize values of each variable into uniformly behaving subdo-

mains, which results in the reduction of the number of atomic entities that must be processed while conceiving the system and reasoning about the system. This can also be interpreted as they create new "child" variables from the existing variables under a IS-A hierarchy where each instance (i.e. value) of the newly created variables is also an instance of their parent variable.

Regarding the algorithm LearnFAVA, one of the first questions that comes to mind is about how legality queries are replied by the SUL. In practice, the SUL may not have a direct function that evaluates legality of a given complete assignment. However, each legality query can be replied by simply setting the SUL to c_{in} by sequentially inputting the variable assignments at its each index (thus resetting the system to the initial state), then setting the system to the desired complete assignment again by means of a sequential input. This brings a cost of $2 \times |V|$ to each legality query.

Another important question about both the representation and the algorithm is how they can be used in case domains of variables are not finite. This problem is naturally divided into two cases: First, the upper bound or the lower bound (or both) of the values may be unknown. Second, the domain may be a continuous domain. For the first case, there seems no solution except finding/assuming an effective interval of values based on observations made on the system. The second case requires discretization of the domains. In this case, the FA-VA representing the system can be approximated by an approach based on iteratively refining an underdetailed initial model: By relying on the knowledge of the experts (of the domain to which the SUL belongs), divide the continuous domain (or domains) to assumedly uniform intervals (i.e. subdomains) and select one value from each interval, then construct the FA-VA with those values. Thereafter, that initial FA-VA can be refined by checking whether the initial intervals are really uniform or not on the basis of the Proposition 2.2.5. as such:

$test_values$ is the list of lists of test values for each interval
for each error state $q_e \in E$:
  for each interval $interval_j$ of the variable v_i:
    if the value representing the $interval_j \in \rho^{-1}(q_e)$:
      for each $c \in \rho^{-1}(q_e)$:
        for each $val \in test_values[j]$
          check if $\alpha(c, i, val)$ is also illegal

If $\alpha(c, i, val)$'s are always illegal, then the FA-VA is correct assuming the test values are sufficient to check the uniformity of the intervals. If not, intervals should be further partitioned considering which test values caused some $\alpha(c, i, val)$ to be legal for which error states. The same method can also be used to improve the efficiency of the algorithm when one or more of the variables have huge domains consisting of numeric values. Additionally, an algorithm implementing this approach (at least a human-in-the loop algorithm) seems to be devisable, which can be proposed as a future work.

Another critical question about the limitations and potential extensions of the algorithm (also of the representation in the current case), which should be mentioned, is how they learn/represent systems with linear, quadratic or polynomial constraints. In case the constraints are known, a FA-VA-based representation of a system with such constraints can be designed as such:

- If only one such constraint exists and the variables taking part at this constraint does not take part in any other constraint, then accept the defining equation of the constraint as a variable (i.e. add to the set of variables) and delete the variables that take part in that constraint from the set of variables. To illustrate, we have the constraint $x + y \leq 5$ and neither x nor y is an effective variable of any other constraint (i.e. of any other error state). Then, delete x and y from the set of variables and add the variable $x + y$ to the set of variables.

- If there exist more than one such constraint and those constraints share some variables with each other or with other constraints, representing the system with a single FA-VA is still possible but highly probably that FA-VA will be huge and hard-to-trace for humans. So as to achieve an easy-to-trace representation, discarding all variables which take part in at least one of such constraint, and designing a FA-VA for other variables, then designing other FA-VAs for each of the linear/quadratic/polynomial constraints may be a good solution for this case. At the end, the collection of those FA-VAs will represent the system such that a sequence is accepted by the system if and only is it is accepted by all of the FA-VAs in the collection. Notice that such a representation will be meaningful if and only if there are multiple constraints with the same defining

equation (e.g. "$x + y > 10$ and $x + y \leq 5$", or "$x - 2y \leq -5$ where $y \in [5, 10]$ and $x - 2y \leq 0$ where $y \in [0, 5)$"). Therefore, the following method which does not directly consider equations may be more useful (however, it is currently incomplete):

- Each element of $\prod_{v_i \in V} D(v_i)$ can be seen as a point at the n-dimensional Euclidean space (where $n = |V|$). Then, each constraint equation with k variables, together with the subdomains on which the equation is defined, define a subspace (of illegal points) of the k-dimensional Euclidean space. The main idea is adding such subspaces as new variables. If the subspace defined by an equation does not intersect with other subspaces defined by other equations, domain of the new variable will be determined by the subdomains on which the equation is defined. Then, if there is an equation defined on k variables, where $V' \subseteq V$ is the set of those k variables and d'_i is the subset of v'_i on which the equation is defined, each k-tuple in the set $\prod_{v'_i \in V'} d'_i$ will be a value of that new variable. When it comes to the actual variables that take part at the equation, if for such a variable v'_i, $D(v'_i) \backslash d'_i \neq \emptyset$, then the variable continues to exist with a smaller domain d_i such that $d_i = D(v'_i) \backslash d'_i$. What critical is, multiple equations may together determine a subspace if their domains intersect (i.e. subspaces defined by different equations intersect each other). This is the case in which the method is partially incomplete, however, the main idea that is expected to work at this case is as such: First generate the subspaces according to each constraint equation and domains of its variables as explained, then partition the those subspaces until none of them intersect each other. At the end, each partition (i.e. each final subspace) will correspond to (the domain of) a new variable. Notably, this is analogous to the LearnFAVA's domain partitioning method.

When it comes to algorithmic inference, if (at the maximal FA-VA representing the SUL) each variable v_i has a subdomain d_i such that $\rho^{-1}(q_e)[i] \cap d_i = \emptyset$ for each error state q_e, and if one complete assignment c_{clear} such that $c_{clear} \in \prod_{v_i \in V} d_i$ is known; further, if it is known that the constraints defined on the system are linear constraints, such a human in the loop method may be followed: Construct the observation table. Observe the variables domains of which are partitioned "too much". Create the set V', which is the set of such variables. Place each element of (i.e. point

from) $\prod_{v'_i \in V'} D(v'_i)$ to the n-dimensional Euclidean space (where $n = |V'|$). Set the SUL to c_{clear}. Make a legality query for each point (notice that the values of all other variables are determined by c_{clear}, which ensures no collision with constraints defined on another set of variables will occur). Mark erroneous points. Then, determine the hyperplanes separating erroneous and non-erroneous points. Compute slopes of the hyperplanes for each dimension. Also determine the points where each hyperplane starts and end. Each hyperplane will correspond to a constraint equation and each of the computed slopes will correspond to the coefficient of one of the variables in the equation.

If a well-defined way of determining whether a domain is partitioned "too much" and a well-defined way of determining the slopes of border-lines(/planes/hyperplanes) between the erroneous and non-erroneous regions of the constructed Euclidean space can be found, an algorithm implementing this method can be devised. Further, extending the same method to the quadratic or higher order polynomials seems also possible on the basis of "checking the slope of the slope", however, to be able to clearly see how this can be achieved, first the algorithm that works on the systems of linear constraints must be completely formulated. Nevertheless, that proposed method (even when the constraints are linear) may require an excessive amount of computational resources. Even further, there may not be a general feasible solution for the specified limitation. Even so, the problem may be partially solved by sacrificing the exactness and following a method similar to the constraint acquisition system Model Seeker [14] employs: Take a catalog of possible linear (or even quadratic) constraints from the user. After marking erroneous points at the Euclidean space, find the constraint (from the catalog) that explains most of the points. Continue finding the most-explanatory constraints until no points are left.

Lastly, there is another arguably-weak side of the FA-VAs: They cannot overcome the problem of combinatorial explosion. More specifically, even a fundamental FA-VA has 2^n states where n is the number of effective variables. However, this is an inherent problem of constraint-based systems. Even so, at a non-fundamental FA-VA, the number of states is $\prod_{1 \leq i \leq |V'|} partitions[i]$ where V' is the subset of variables each element of which is an effective variable of at least one error state. Considering the existence of multiple error states sharing the same effective variable v_i increase

the number of elements of $partitions[i]$ at least by the number of such states, it can be easily seen how rapid the state space explodes with the increasing number of error states. As a solution, representing the systems with multiple fundamental FA-VAs, as shown by the Theorem 2.3.4, can be proposed. Therefore, an algorithm that infers interactive variable assignment systems in terms of multiple fundamental FA-VAs is urgently needed, which will be an immediate next step of this work.

REFERENCES

[1] H. R. Lewis and C. H. Papadimitriou, *Elements of the theory of computation, 2nd Edition*. Prentice Hall, 1998.

[2] F. Vaandrager, "Model learning," *Communications of the ACM*, vol. 60, p. 86–95, Jan 2017.

[3] S. J. Russell, P. Norvig, and M.-W. Chang, *Artificial Intelligence: A modern approach, Fourth edition*. Pearson Education, 2022.

[4] D. R. Kuhn, R. Bryce, F. Duan, L. S. Ghandehari, Y. Lei, and R. N. Kacker, "Chapter one - combinatorial testing: Theory and practice," vol. 99 of *Advances in Computers*, pp. 1–66, Elsevier, 2015.

[5] A. Garipler, E. Sorguç, E. Özel, S. E. Uslu, M. Saeedi Nikoo, and H. Oğuztüzün, "Model learning for the programming interface of a measurement device," in *Proceedings of the 13th Turkish National Software Engineering Symposium*, p. 249–259, 2019.

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

[7] S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen, "A succinct canonical register automaton model," *Journal of Logical and Algebraic Methods in Programming*, vol. 84, no. 1, pp. 54–66, 2015.

[8] P.-N. Tan, M. Steinbach, A. Karpatne, and V. Kumar, *Introduction to data mining*. Pearson Education, 2020.

[9] S. Cassel, F. Howar, B. Jonsson, and B. Steffen, "Active learning for extended finite state machines," *Formal Aspects of Computing*, vol. 28, pp. 233–263, 2016.

[10] F. Aarts, P. Fiterau-Brostean, H. Kuppens, and F. Vaandrager, "Learning register automata with fresh value generation," in *Theoretical Aspects of Computing-*

ICTAC 2015: 12th International Colloquium, Cali, Colombia, October 29-31, 2015, Proceedings 12, pp. 165–183, Springer, 2015.

[11] V. N. Rao, "Solving constraint satisfaction problems using finite state automata," *National Conference on Artificial Intelligence*, p. 453–458, Jul 1992.

[12] C. Bessiere, A. Daoudi, E. Hébrard, G. Katsirelos, N. Lazaar, Y. Mechqrane, N. Narodytska, C.-G. Quimper, and T. Walsh, "New Approaches to Constraint Acquisition," in *Data Mining and Constraint Programming*, vol. 10101 of *Lecture Notes in Computer Science*, pp. 51–76, Springer International Publishing AG, 2016.

[13] C. Bessiere, F. Koriche, N. Lazaar, and B. O'Sullivan, "Constraint Acquisition," *Artificial Intelligence*, vol. 244, pp. 315–342, Mar. 2017.

[14] N. Beldiceanu and H. Simonis, "A model seeker: Extracting global constraint models from positive examples," in *Principles and Practice of Constraint Programming* (M. Milano, ed.), (Berlin, Heidelberg), pp. 141–157, Springer Berlin Heidelberg, 2012.

[15] R. Coletta, C. Bessière, B. O'Sullivan, E. C. Freuder, S. O'Connell, and J. Quinqueton, "Semi-automatic modeling by constraint acquisition," in *Principles and Practice of Constraint Programming – CP 2003* (F. Rossi, ed.), (Berlin, Heidelberg), Springer Berlin Heidelberg, 2003.

[16] A. Nerode, "Linear automaton transformations," *Proceedings of the American Mathematical Society*, vol. 9, no. 4, pp. 541–544, 1958.

[17] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to automata theory, languages, and computation," *Acm Sigact News*, vol. 32, no. 1, pp. 60–65, 2001.

[18] A. Ginzburg, *Algebraic theory of Automata*. Academic Press, 1968.