Suffix-reading Automata: Putting Practice into Theory (and back)

By

R Keerthan

A thesis submitted in partial fulfilment of the requirements

for the degree of Doctor of Philosophy

to

Chennai Mathematical Institute

April 2026

Plot-H1, SIPCOT IT Park,

Siruseri, Kelambakkam,

Tamilnadu - 603103

India

CHENNAI
MATHEMATICAL
INSTITUTE

R Keerthan

Plot-H1, SIPCOT IT Park

Padur Post, Siruseri

Tamil Nadu, India-603103

E-mail:keerthan@cmi.ac.in

DECLARATION

I declare that the thesis entitled **"Suffix-reading Automata: Putting Practice into Theory (and back)"** submitted by me for the degree of **Doctor of Philosophy in Computer Science** is the record of academic work carried out by me under the guidance of Professor B Srivathsan and this work has not formed the basis for the award of any degree, diploma, associateship, fellowship or other titles in this University or any other University or Institution of Higher Learning.

R Keerthan

Chennai Mathematical Institute

April 2026.

CHENNAI
MATHEMATICAL
INSTITUTE

B Srivathsan

Plot-H1, SIPCOT IT Park

Padur Post, Siruseri

Tamil Nadu, India-603103

E-mail:sri@cmi.ac.in

CERTIFICATE

I certify that the thesis entitled **"Suffix-reading Automata: Putting Practice into Theory (and back)"** submitted for the degree of **Doctor of Philosophy in Computer Science** by "R Keerthan" is the record of research work carried out by him under my guidance and supervision, and that this work has not formed the basis for the award of any degree, diploma, associateship, fellowship or other titles in this University or any other University or Institution of Higher Learning. I further certify that the new results presented in this thesis represent his independent work in a very substantial measure.

Chennai Mathematical Institute — *B Srivathsan*

Date: April 2026. — *Supervisor.*

ACKNOWLEDGEMENT

This thesis is the culmination of a long journey, made possible only through the support and patience of a large group of well-wishers. Although it might not do justice to their contributions, here is my attempt to thank everyone.

First, I thank my advisor, Prof. B Srivathsan. for being as supportive and patient as humanly possible through a long-winded, and at times, 'no-end-in-sight' path. There have been multiple occasions where I simply wanted to give up and move on with life, and I would have if not for him. Today I can move on without regret, having learned so much from his approach to our work together. I gained not only from his careful and sure-footed guidance to any technical work, which I will hold close as inspiration and try to emulate, but also his sheer tenacity to keep attacking a problem that gave us so little initially. If I can imitate a fraction of the persistence I saw, I will surely do well in any future challenges I may face.

Next, I thank Venky for being a mentor, and practically my second advisor during the latter half of my PhD. Venky's contribution began early in my journey, during our attempts to collaborate with TCS Innovation Labs on an industry problem. After an initial exploration of some theoretical problems in automata learning, the shape of the work changed and took on a more practical motivation thanks to him. I will also be forever grateful to Venky for allowing me to join his team at TCS and still continue my PhD, during a crucial juncture when the pandemic made it unclear whether work could even proceed as before.

As a a collaborator, I could not have asked for a better partner than Sagar Verma at TCS. My time there was so fruitful to this thesis only because Sagar helped me tackle every problem I encountered, while managing his own work effortlessly. With the benefit of hindsight, our shared commiserations after failed attempts towards this work are now memories I will cherish. I also thank Arjun Arul for his contribution to a technical tool in our work, but more importantly for nudging me to join CMI (and hence being solely responsible for any frustrations this thesis may create).

I thank the doctoral committee members who oversaw the thesis, Prof. Madhavan Mukund and Prof. C. Aiswarya, for timely discussions and feedback that helped shape the journey of

this work. I also thank the other faculty members at CMI, especially in Computer Science, for their influence on my path whether direct or indirect. From my time at TCS, I thank Supriya Agarwal and Ulka Shrotri for their collaboration and mentorship. I also thank Prof. Supratik Chakraborty for all the discussions and insights that helped shape this work. I am grateful to Prof. Kumar Madhukar and Dr. Ocan Sankur for their thoughtful comments and questions while reviewing the thesis.

My time at CMI was eventful and thoroughly enjoyable thanks to all the friends, colleagues and flatmates who helped while sharing their own journeys with me. The list is too long but I must name at least a few; Sonakshi, Arjun, Himalay, Govind, Sayan, Adwitee, Manu, Ramadas, Vishnu, Archit and so many others! I also thank my friends and colleagues in the Foundations of Computing group at TCS, Sagar, Diganta, Nakshatra, Pranav, Afzal and many others. Some helped directly in discussions about my work, some shared their time beyond what one can reasonably expect, some played football with me to let off steam, but everyone helped shape who I am and what I can do today.

A special thanks to all the staff at CMI, spearheaded by the indefatigable efforts of Rajeshwari, Sripathy and Ranjini, for all their help given to every student who asked (or as in my case sometimes, did not even know to ask). I tried repeatedly to defeat them with my problems and failed every single time.

I thank the Infosys foundation, Chennai Mathematical Institute and Tata Consultancy Services for funding. CMI and TCS especially played a crucial role in supporting my travel and participation in several workshops and conferences. These include MOVEP 2018 at ENS Cachan, Marktoberdorf Summer School 2019, GandALF 2024, NETYS 2025, and almost every possible event in India relevant to our work (FSTTCS, FM Update, SAT-SMT school, etc.). I also thank TCS for funding travel and enabling academic visits to Srivathsan at CMI periodically.

Outside of research, I thank my friends from school (Arjun), undergrad, and grad years (Himalay, Sonakshi, Kela, Rajit) for the role they played in my life. Since I cannot describe their contributions in full, I will simply mention the time they planned for us to stay in Gangtok for a month, after the pandemic when we could still work from ‘home’. It was the most enjoyable month of ‘work’ I have ever done.

Finally, my family – my mother, Mangala, my father, Ravi, my sisters, Shruthi and Swathi, and my wife, Sonakshi. Thank you. I could repeat the words a thousand times and it would not be enough. Thank you for everything. I dedicate this thesis to you.

P.S. A special mention to Snowy and Ilish. They were good cats, and they still are! :)

Contents

1 Introduction **1**

1.1 The need for Verifiable and Succinct System Models 1

1.2 Automata models in Practical applications 6

1.3 Bridging the Gap to Concurrent Systems 11

1.4 Thesis Contributions and Outline 13

2 Preliminaries **17**

3 A new automaton model – Deterministic Suffix-Reading Automata **21**

3.1 DSA: formal syntax and semantics 22

3.2 Comparison with DFA and DGA 25

4 Suffix-tracking sets – deriving DSA from DFA **31**

4.1 Building an induced DSA 34

4.2 Removing some redundant transitions. 40

5 Minimality – some observations and challenges **45**

5.1 Some properties of minimal DSAs 46

5.2 Complexity of minimization 51

5.3 Strongly Deterministic Suffix-reading Automata (sDSA) 54

5.3.1 Syntax of strong DSAs 55

5.3.2 Minimality for strong DSAs 57

6 Multi-port Deterministic Suffix-Reading Automata **63**

6.1 Some challenges in extending to multiport 68

6.2 Multi-port DSAs: formal syntax and semantics 69

7 Multiport DSA with outputs **73**

7.1 mDSA-with-outputs: formal syntax and semantics 75

7.2 Complexity of state reachability 77

7.3 Expressive decision tables 79

7.3.1 Translating EDT to mDSA-with-outputs 80

7.3.2 Experiments 81

8 Conclusion **85**

A Bounded test generation **97**

A.1 Mealy machines 97

A.2 Basic EDT 98

A.2.1 Mealy machines for basic EDTs 101

A.3 Test case search using SAT 102

A.3.1 Example 102

A.3.2 General case 104

Chapter 1

Introduction

1.1 The need for Verifiable and Succinct System Models

Role of Formal Methods in Engineering Reliable Systems

The design and implementation of contemporary software and hardware systems are endeavours of immense complexity. As these systems become increasingly integrated into safety-critical and security-critical domains, from avionics and medical devices to financial networks and autonomous vehicles, ensuring their correctness and reliability has grown paramount. In response to this challenge, the field of computer science has developed **Formal Methods**, a collection of mathematically rigorous techniques and tools for the specification, design, and verification of such systems. The foundational principle of formal methods is that by modelling systems as mathematical entities, one can leverage the power of logic and proof to reason about their behaviour with a level of precision and completeness that empirical testing alone cannot achieve.

The value of formal methods lies in their capacity to symbolically examine the entire state space of a design, thereby establishing correctness or safety properties that hold true for all possible inputs and execution scenarios. This rigorous approach can be applied at various points throughout the development lifecycle. In the initial stages, formal specification languages help disambiguate informal requirements, providing a precise and unambiguous

blueprint that can guide subsequent development. During design and implementation, formal verification techniques such as model checking and automated theorem proving can be used to prove that a system implementation correctly adheres to its specification, uncovering subtle flaws that might otherwise go undetected until late in the process, or even after deployment.

Despite their power and a growing number of industrial success stories, the widespread adoption of formal methods has been historically hindered by a well-documented "practicality gap". These techniques are often perceived as being too complex, costly in terms of time and resources, and requiring a level of specialized mathematical expertise that is not always available in typical engineering teams. This perception has created a persistent challenge: how to bridge the divide between the theoretical power of formalisms and the practical needs of system engineers. A central theme of this thesis is the direct engagement with this challenge. *The research presented here is driven by the goal of developing formal models that are not only theoretically sound but also more intuitive, concise, and "readable"*. By striving for models that are arguably more readable, this work aims to lower the barrier and make formal reasoning more accessible to a broader range of engineering contexts, putting formal methods into wider practice.

Automata Theory - A Foundation for System Modelling

Central to the discipline of formal methods is **Automata Theory**. Automata are abstract machines that serve as formal models of computation. Among various classes of automata, the **Deterministic Finite Automaton (DFA)** is the canonical model for the class of regular languages. A DFA is formally a tuple $(Q, \Sigma, q_{\text{init}}, \delta, F)$ where Q is a finite set of states, Σ is a finite alphabet of input symbols, q_{init} is the initial state, δ is a transition function, and F is a set of accepting states. DFAs are fundamental to the theory of computation and have been successfully applied in numerous practical domains.

A cornerstone of automata theory is the Myhill-Nerode theorem, which establishes a connection between a regular language and its unique minimal-state DFA. The theorem employs the Nerode equivalence relation, which deems two words equivalent if they are indistinguish-

able by any suffix. The number of equivalence classes of this relation is precisely the number of states in the smallest possible DFA for the language. The minimal DFA, also called the **canonical DFA**, is unique up to isomorphism. This result provides a point of reference, as we will see a departure from this notion when considering alternative automaton models.

The Pervasive Challenge: State-Space Explosion

While DFAs and other finite-state models provide a basis for system verification, their practical application is often constrained by the **state-space explosion problem**. This refers to the exponential growth in the number of states of a model as the number of system components, variables, or concurrent processes increases linearly. For even moderately complex systems, the resulting state space can become astronomically large, quickly overwhelming the memory and computational resources of verification tools and rendering exhaustive analysis infeasible.

This state-space explosion is recognized as the primary bottleneck limiting the scalability of automata-based verification techniques, particularly model checking. Consequently, a significant portion of research in formal verification over the past several decades has been dedicated to developing techniques to mitigate its effects. These techniques include, among others, abstraction methods that hide irrelevant detail, partial order reduction for concurrent systems, and symbolic model checking, which uses compact data structures like Ordered Binary Decision Diagrams (OBDDs) or formulas in a boolean logic to represent vast sets of states and transitions implicitly rather than explicitly.

Another line of attack against this problem, and the one central to this dissertation, is the development of *more succinct modelling formalisms*. The goal is to devise new types of automata or specification languages that can represent complex behaviours using fewer descriptive resources (e.g., states, transitions, or symbols) than, say, a DFA. This thesis tackles two facets of this combinatorial challenge: The first part addresses the state explosion that arises in modelling complex sequential behaviour, proposing a new automaton model [KSVV24] designed for greater conciseness. The second part tackles the "interleaving explosion" [KSV26] that occurs when modelling concurrent systems, where the need to account for all possible

orderings of asynchronous events leads to a combinatorial blow-up in the model.

Model-Based Testing: A Motivation for Succinct Models

The search for more succinct and effective formal models is not just a theoretical pursuit; it is strongly driven by practical engineering needs, particularly in the domain of **Model-Based Testing (MBT)** [DNSVT07]. MBT is a software testing paradigm where test cases are automatically generated from an abstract, formal model of the System-Under-Test (SUT) [Bro24]. This approach offers the promise of more systematic and thorough testing with reduced manual effort, enhancing software quality and reliability [LLGS18].

The typical MBT workflow involves several key steps [LLGS18]:

1. **Model Creation:** An engineer develops a formal model (e.g. a state machine, a decision table, or a UML diagram) that captures the desired behaviour of the SUT.

2. **Test Generation:** A tool automatically derives abstract test cases from the model by applying test selection criteria. These criteria can be based on structural coverage of the model (e.g., covering all states or transitions) or on requirements coverage.

3. **Test Concretization and Execution:** The abstract test cases are then translated into concrete, executable test scripts that can be run against the SUT, thus providing the oracle to determine pass/fail verdicts.

The effectiveness of the entire MBT process is fundamentally dependent on the quality and tractability of the underlying model. A model that is excessively large, difficult to construct, or hard to understand becomes a significant bottleneck, undermining the potential benefits of automation [DNSVT07]. This creates a powerful, industry-relevant motivation for the research presented in this thesis. The development of formalisms that are more concise and intuitive directly translates into more efficient and effective MBT. A key contribution of this thesis is the application of its novel automata models to provide a formal semantics for Expressive Decision Tables (EDTs) [VSKA14], an industrial specification notation used for test generation.

Expressive Decision Tables (EDTs). Introduced by Venkatesh et al., at the verification research group of Tata Consultancy Services Innovation Labs, EDTs are a notation for specifying software requirements for reactive systems. Due to the ease of their usage, EDTs have been successfully applied in various industry projects to generate test cases for software requirements. The notation however lacks rigorous formal semantics and existing test case generation procedures are based on an intuitive understanding of EDT behaviour. In particular, the current procedures are incomplete: they employ search heuristics which try to cover as many requirements as possible.

In the EDT syntax, systems are assumed to consist of input variables, output variables and local variables (or states). The software requirements specify the behaviour of the outputs and local variables based on patterns in the inputs and local variables. Table 1.1 gives an example inspired from [VSZA15] which describes selected requirements of a car alarm module.

Table 1.1: EDT for an Alarm module

sno	in PanicSw	in Alarm	out Alarm	out Flash
1	(Press; Press){< 3s}	Off	On	
2	Press{> 3s}	On	Off	False

An EDT can be seen as being divided into two sides - the input side and the output side. Column headers on the input side describe the input and local variables, and are prefixed with *in*. Similarly, column headers on the output side describe the local and output variables, and are prefixed by *out*. Local variables occur on both sides. In Table 1.1, note that the variable *Alarm* appears on both the sides, making it a local variable. Each row gives a system requirement. The EDT of Table 1.1 gives a concise representation of the following requirements:

1. If *Alarm* is *Off*, and *PanicSw* is pressed twice within 3 seconds, then *Alarm* should be turned *On*.

2. If it is more than 3 seconds since the *PanicSw* is pressed, and the *Alarm* is *On*, then *Flash* should be *False* and *Alarm* should be turned *Off*.

A test case for a requirement is a sequence of inputs that triggers the particular require-

ment. EDTs have been successfully applied to generate test cases for software requirements — thanks to the simple syntax, system engineers are comfortable in translating a large set of requirements into EDTs and algorithms for generating test cases have been studied and shown to be more effective than manual testing [VSZA15]. However, the notation lacks a rigorous formal semantics, except for a brief description of key elements of formal semantics in [VSKA14]. Current test generation algorithms are based on this intuitive understanding of the language, and apply heuristics to cover as many requirements as possible. When certain requirements do not get covered, a manual analysis is performed. The mechanics of EDTs is intricate due to various interactions between the rows. An automated tool to generate test cases or *guarantee* non-coverability is indispensable. To achieve this, the first step is to have a formal operational semantics for EDT.

In this thesis, we provide formal (operational) semantics for a significant (untimed) fragment of EDTs, via new automata models. In addition to providing a framework for reasoning about EDTs, the formal semantics gives us a test generation procedure.

1.2 Automata models in Practical applications

Deterministic Finite Automata (DFAs) have been widely applied in areas as diverse as text processing [MMW09], formal specification languages [Har87], and as the underlying engine for verification techniques like model-checking [CGK+18] and runtime verification [BJNT00, BHV04, GH01]. The broad utility of automata is further evidenced by the development of numerous software libraries dedicated to their creation and manipulation [LMS22].

However, the state-space explosion problem consistently hinders their practical application. Although the state-and-transition paradigm is intuitive, its descriptions operate at a low level of abstraction. This granular, character-by-character processing often results in automata of unmanageable size for non-trivial applications, creating a need for more concise formalisms. While non-determinism is a well-known path to exponential succinctness, a deterministic model is often essential for unambiguous formal specifications and efficient implementations. The following survey discusses several deterministic approaches from the

literature aimed at tackling the problem of large automaton size.

One major strategy for achieving succinctness is to enhance transition labels, moving from single letters to more complex patterns. *Generalized automata* (GA), first defined by Eilenberg [Eil74], extend non-deterministic automata with string-labeled transitions; a word is accepted if it can be partitioned into segments that match a sequence of these transition labels. Subsequent work established bounds on the label lengths required for minimal GAs [Has91]. The deterministic variant, *Deterministic Generalized Automata* (DGA), requires that for any state, the set of outgoing string labels must be prefix-free [GM99]. The key method for constructing smaller DGAs involves starting with a DFA and "suppressing" states to create longer labels. Giammarresi et al. [GM99] showed that minimal DGAs (in terms of state count) can be derived from the canonical DFA using this technique. However, the more natural question of minimizing the *total size* of a DGA—a metric that includes the number of states, edges, and the sum of label lengths—was left as an open problem. A natural extension from strings to more complex objects is to consider regular expressions. The resulting model, known as *Expression Automata* [HW04], was first considered in the context of converting automata to regular expressions [BM63]. To achieve determinism, however, the *Deterministic Expression Automata* (DEA) model imposes two strict requirements: from any state, the languages of any two outgoing expressions must be disjoint, and each of those languages must be prefix-free. It is stated in [HW04] that DEA accept prefix-free regular languages, and hence DEAs are less expressive than standard DFAs.

Another strategy for succinctness targets the complexity that arises from the size of the alphabet itself. For instance, an alphabet consisting of all ASCII characters can cause the number of transitions to blow up. *Symbolic automata* [VHL+12, DV17] have been proposed to handle large or infinite alphabets. They replace letters on edges with logical formulas or predicates, which allows them to club together numerous transitions between a pair of states into a single symbolic transition. For example, if the domain is the set of natural numbers, a transition labeled with a predicate like $\mathrm{odd(x)}$ can act as a placeholder for all transitions labeled with an odd number. This approach has been widely applied and implemented in many tools [D'A].

Despite these advances, a gap remains for a formalism that can succinctly model behaviors defined by high-level patterns that are not necessarily contiguous prefixes and may be separated by long, irrelevant sequences of events. For instance, EDTs talk about sequences that contain a certain pattern in order to fire certain outputs. This was our inspiration: what is a corresponding notion in an automaton setting, a model that functions based on patterns?

Our model. In this work, we introduce *Deterministic Suffix-reading Automata (DSAs)* to fill this gap. We continue to work with strings on transitions, as in a DGA. However, the meaning of transitions is different. A transition $q \xrightarrow{abba} q'$ is enabled if at q, a word w *ending* with $abba$ is seen, and moreover no other transition out of q is enabled at any proper prefix of w. Intuitively, the automaton tracks a finite set of pattern strings at each state. It stays in a state, passively consuming input, until one of them appears as the *suffix* of the word read so far, and then makes the appropriate transition.

We start with a motivating example. Consider a model for out-of-context `else` statements, in relation to `if` and `endif` statements in a programming language. Assume a suitable alphabet Σ of characters. Let $L_{\texttt{else}}$ be the set of all strings over the alphabet where (1) there are no nested `if` statements, and (2) there is an `else` which is not between an `if` and an `endif`. A standard DFA for this language would need to perform detailed string matching to detect the keywords. The DSA, shown in Figure 1.1, offers a more direct abstraction. At state s_0, it passively reads letters until it first sees a suffix matching either `if` or `else`. If the first match is `if`, it transitions to s_1. For instance, on a word like `abf4fgif`, the automaton moves to s_1 because the word ends with `if` and no trigger pattern appeared in any of its proper prefixes. Similarly, from s_1, it waits for the next pattern, `if` or `endif`, to decide its next move.

Suffix-reading automata have the ability to wait at a state, reading long words until a matching pattern is seen. This results in an arguably more readable specification for languages which are "pattern-intensive". This representation is orthogonal to the approaches considered so far. While symbolic automata club together transitions between a specific pair of states, DSAs can consolidate entire paths of states and transitions. And while DGAs can also club

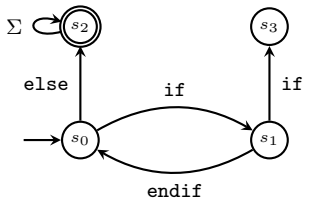


FIGURE 1.1: DSA for out-of-context `else`

paths, they cannot ignore intermediate letters, which can result in extra states and transitions that DSAs can avoid.

Overview of results. We formally present deterministic suffix-reading automata and its semantics, quantify its size in comparison to an equivalent DFA, and study an algorithm to construct DSAs starting from a DFA. This is in the same spirit as in DGAs, where smaller DGAs are obtained by suppressing states. For automata models with strings on transitions, the number of states is not a faithful measure of the size of a DSA. As described in [GM99] for DGAs, we consider the total-size of a DSA which includes the number of states, edges, and the sum of label lengths. The key contributions of this work are:

1. Presentation of a definition of a new kind of automaton - DSA (Chapter 3).

2. Proof that DSAs accept regular languages, and nothing more.

   (a) Every complete DFA can be seen as a DSA.

   (b) For the converse, we prove that for every DSA of size k, there is a DFA with size at most $2k \cdot (1 + 2|\Sigma|)$, where Σ is the alphabet (Lemma 3.5, Theorem 3.7).

   We describe a family of languages L_n, with alphabet size n, for which the minimal DFA has size quadratic in n, whereas size of DSAs is a linear function of n (Lemma 3.6).

3. We present a method to derive DSAs out of DFAs, a DFA-to-DSA conversion (Chapter 4).

   The derivation procedure selects subsets of DFA-states, and adds transitions labeled with (some of) the acyclic paths between them. We identify sufficient conditions on the selected subset of states, so that the derivation procedure preserves the language (Theorem 4.9).

4. We remark that minimal DSAs need not be unique, and make an observation: the smallest DSA that we derive from the canonical DFA of a language L need not be a minimal DSA. (Section 5).

5. Motivated by the above observation, we present a restricted definition of DSA, called *strong DSA* (Section 5.3). We show that minimal strong DSAs can in fact be derived from the canonical DFA, through our derivation procedure.

6. Finally, we show that given a DFA and a number k, deciding if there exists a DSA of size $\leq k$ is NP-complete (Section 5.2).

Related work. The closest to our work is [GM99] which introduces DGAs, and gives a procedure to derive DGAs from DFAs. The focus however is on getting DGAs with as few states as possible. The observations presented in Chapter 5 of our work, also apply for state-minimality: the same example shows that in order to get a DSA with fewer states, one may have to start with a bigger DFA. This is in sharp contrast to the DGA setting, where the derivation procedure of [GM99] yields a minimal DGA (in the number of states) when applied on the canonical DFA. The problem of deriving DGAs with minimal total-size was left open in [GM99], and continues to remain so, to the best of our knowledge. Expression automata [HW04] allow regular expressions as transition labels. This model was already considered in [BM63] to convert automata to regular expressions. Every DFA can be converted to a two state expression automaton with a regular expression connecting them. A model of deterministic Expression automata (DEA) was proposed in [HW04] with restrictions that limit the expressive power. An algorithm to convert a DFA to a DEA, by repeated state elimination, is proposed in [HW04]. The resulting DEA is minimal in the number of states. Minimization of NFAs was studied in [JR93] and shown to be hard. Succinctness of models with different features, like alternation, two-wayness, pebbles, and a notion of concurrency, has been studied in [GH96].

Here are some works that are related to the spirit of finding more readable specifications. [Fer09] has used the model of deterministic GA to develop a learning algorithm for some

simple forms of regular expressions, with applications in learning DTD specifications for XML code. In this paper, the author talks about readability of specifications. They claim that regular expressions (REs) are arguably the best way to specify regular languages. They also claim that the translation algorithms from DFAs to REs give unreadable REs. In the paper, they consider specific language classes and generate algorithms to learn simple looking REs.

Another work that talks about readability of specifications is [ZLL02]. They consider automata based specification languages and perform extensive experiments to determine which choice of syntax gives better readability. They remark that hierarchical specifications are easier, since not every transition needs to be specified. Once again, we observe that there is an effort to remove transitions.

1.3 Bridging the Gap to Concurrent Systems

DSA provide a new toolkit for the succinct specification of sequential, pattern-based systems. However, modern software systems are rarely monolithic and sequential. They are typically composed of multiple interacting components that operate concurrently. This section motivates the extension of the suffix-reading paradigm from the sequential to the concurrent domain [KSV26].

The Limits of Sequential Models for Concurrency

Applying a sequential model like a DSA directly to a concurrent system runs into an obstacle: the "interleaving explosion". A concurrent system consists of multiple processes or components that execute asynchronously. To model such a system with a sequential automaton, one must account for all possible orderings, or interleavings, of the events generated by the different components. The number of such interleavings grows exponentially with the number of events.

This problem is made concrete by a Car Security System example, illustrated in Figure 1.2. A practical requirement for starting a car might be that the key is in the ignition (K), the brake pedal is pressed (B), and the transmission is in park (P), with these three actions

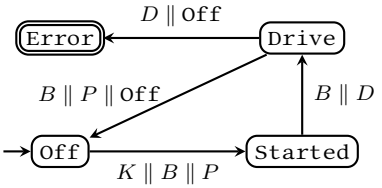


FIGURE 1.2: Car security system example

allowed to occur in any order. A standard DSA, in order to capture this behaviour, would be forced to have a transition whose label explicitly enumerates all $3! = 6$ permutations: KBP, KPB, BKP, BPK, PKB, and PBK. This approach completely undermines the goal of conciseness and readability that motivated the DSA model in the first place. As the number of concurrent components grows, this becomes utterly impractical.

Specification of Concurrent Suffix-Based Rules

The failure of the sequential model in the setting of concurrency gives rise to the second central problem of this thesis: *How can we model specifications that involve patterns across multiple concurrent "ports"?* A natural syntactic approach is to introduce a parallel composition operator $\|$, into the transition labels. Using this operator, the car security requirement could be expressed with a single, elegant transition labeled 'K $\|$ B $\|$ P'. This syntax would abstract away the interleavings, capturing the logical intent of the specification directly. However, introducing such an operator should not be merely a syntactic convenience. The core challenge lies in defining a formal operational semantics for it that is both theoretically sound and intuitively useful for a wide range of real-world specification scenarios. The development of such a semantics is the primary focus of this next part.

Our model. To address the challenge of specifying concurrent systems, this thesis introduces the **Multi-port Suffix-reading Automaton (mDSA)**, an extension of the DSA model designed to handle concurrency and outputs.

The mDSA model operates over a multi-port alphabet $\Sigma = \langle \Sigma_1, ..., \Sigma_k \rangle$, where the global alphabet is partitioned among k distinct ports. Transitions are labeled with concurrent patterns like $(u_1 \parallel u_2 \parallel ... \parallel u_k)$, where each u_i is a word (a pattern) from the alphabet of port i,

Σ_i^* [KSV26].

The development of the mDSA model is not just theoretical, but driven by the practical need to formalize existing industrial specification methods. This is demonstrated through its application to Expressive Decision Tables (EDTs). As we mentioned earlier, EDTs allow engineers to specify complex, suffix-based rules in a convenient format.

To model the full behaviour of EDTs, which includes not just input conditions but also the generation of output actions, the mDSA model requires an additional extension. This practical requirement directly motivates our development of **mDSAs-with-outputs**. In this model, a transition can now specify that upon matching an input condition, a set of output symbols should be produced. It creates a feedback loop where the automaton can modify input conditions of subsequent transitions via previous outputs. The main theoretical result here is that state reachability for mDSAs-with-outputs is PSPACE-complete [KSV26].

This entire development arc embodies the (sub)title of this dissertation, "Putting Practice into Theory (and Back)." The practical need to provide a formal semantics for an industrial notation (EDT) led to theoretical model extensions, from DSA to mDSA and mDSAs-with-outputs). The formal analysis of these new theoretical models, in turn, revealed a fundamental complexity result. This theoretical discovery then feeds back to practice, providing a formal explanation for the inherent difficulty of test generation for EDT-like specifications and informing the design of such verification tools.

1.4 Thesis Contributions and Outline

This dissertation introduces and analyzes a new family of automata, Suffix-Reading Automata, designed to provide succinct and readable formal models for sequential and concurrent systems. The research bridges the gap between automata theory and engineering practice by developing formalisms that are both theoretically sound and practically motivated.

Summary of Contributions

The primary contributions of this dissertation can be summarized as follows:

- **A Model for Sequential Specifications:** The introduction of the Deterministic Suffix-reading Automaton (DSA) as a novel, fully expressive, and deterministic formalism for regular languages. This model provides a more succinct and readable representation for pattern-intensive specifications compared to traditional DFAs. A theoretical analysis of its properties is provided.
  - **Automaton Minimization Analysis:** The discovery of a fundamental bottleneck in DSA minimization, demonstrating that the canonical (Myhill-Nerode) DFA is insufficient for deriving minimal DSAs. This is complemented by a formal proof that the DSA minimization problem is NP-complete. As a constructive solution, the Strong DSA (sDSA) subclass is introduced, which is shown to be a tractable and well-behaved class for which minimal automata are derivable from the canonical DFA.
- **A Model for Concurrent Specifications:** The extension of the suffix-reading paradigm to concurrent systems with the Multi-port DSA (mDSA). This model features a tape-based semantics with a history marker that formalizes the management of event history and the notion of "freshness" required for triggering concurrent transitions.
  - **Bridging Formalism and Industrial Practice:** The development of the first formal operational semantics for the industrial Expressive Decision Table (EDT) notation, via a translation to mDSAs-with-outputs. The proof that the associated test generation problem (state reachability) is PSPACE-complete reveals the complexity inherent in such practical specification languages.

Thesis Structure

The remainder of this dissertation is organized as follows:

- **Chapter 2: Preliminaries (with a Survey of Automata Models):** This chapter will formally define the foundational concepts from automata theory and formal methods that are used throughout the thesis. It will also present a technical introduction of related automata models discussed in this introduction.

- **Chapter 3: Deterministic Suffix-Reading Automata (DSA):** This chapter will present the formal definition, semantics and theoretical analysis of the DSA model. This includes proofs of expressiveness, and succinctness results.
- **Chapter 4: Derivation of DSAs:** This chapter will detail the DFA-to-DSA derivation procedure via suffix-tracking sets.
- **Chapter 4: DSA Minimality:** This chapter will demonstrate the canonical DFA bottleneck for minimization, the full NP-completeness proof, and the theory of Strong DSAs (sDSAs).
- **Chapter 6: Multi-Port DSA (mDSA):** This chapter will introduce the mDSA model for specifying concurrent systems. It will detail the multi-port alphabet, the $\|$ operator, and the tape-based semantics.
- **Chapter 7: Application to Expressive Decision Tables (mDSA-with-outputs):** This chapter will present the translation from the EDT notation to mDSAs-with-outputs. It will then present the full proof of PSPACE-completeness for the state reachability problem.
- **Chapter 8: Conclusion and Future Work:** This chapter will summarize the key findings of the dissertation and discuss several promising directions for future research.

Chapter 2

Preliminaries

Before introducing the formal details of the new automaton model at the heart of this thesis, this chapter establishes the necessary theoretical foundation. We will review the standard definitions and notations from formal language theory, focusing on Deterministic Finite Automata (DFAs) as the canonical model for regular languages. Furthermore, we will define Deterministic Generalized Automata (DGAs), a key model for comparison that also utilizes strings on transitions but with a different matching rule. This groundwork will provide the vocabulary and context required to situate and analyze the suffix-reading paradigm presented in the chapters that follow.

Basic notations. We fix a finite alphabet Σ. Following standard convention, we write Σ^* for the set of all words (including ε) over Σ, and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. For $w \in \Sigma^*$, we write $|w|$ for the length of w, with $|\varepsilon|$ considered to be 0. A word u is a *prefix* of word w if $w = uv$ for some $v \in \Sigma^*$; it is a *proper-prefix* if $v \in \Sigma^+$. Observe that ε is a prefix of every word. A set of words W is said to be a *prefix-free set* if no word in W is a prefix of another word in W. A word u is a *suffix* (resp. *proper-suffix*) of w if $w = vu$ for some $v \in \Sigma^*$ (resp. $v \in \Sigma^+$). We state below some notations which will come in handy for the formal definition and analysis of our suffix-reading automaton model.

Definition 2.1 (Notations for suffix)**.** For two words w_1, w_2, we write $w_1 \sqsubseteq_{\mathsf{sf}} w_2$ if w_1 is a suffix of w_2. For a set of words $U \subseteq \Sigma^*$ and $\alpha, \beta \in \Sigma^*$, $\alpha \sqsubseteq^U_{\mathsf{lsf}} \beta$ if α is the longest word in U

which is a suffix of β, i.e. for all $\alpha' \in U$ if $\alpha' \sqsubseteq_{\mathsf{sf}} \beta$ then $|\alpha'| \leq |\alpha|$.

Definition 2.2 (Deterministic Finite Automata (DFA))**.** A *Deterministic Finite Automaton (DFA)* M is a tuple $(Q, \Sigma, q^{init}, \delta, F)$ where Q is a finite set of states, $q^{init} \in Q$ is the initial state, $F \subseteq Q$ is a set of accepting states, and $\delta : Q \times \Sigma \to Q$ is a partial function describing the transitions. If δ is complete, the automaton is said to be a complete DFA. Else, it is called a trim DFA.

The run of DFA M on a word $w = a_1 a_2 \dots a_n$ (where $a_i \in \Sigma$) is a sequence of transitions $(q_0, a_1, q_1)(q_1, a_2, q_2) \dots (q_{n-1}, a_n, q_n)$ where $\delta(q_i, a_{i+1}) = q_{i+1}$ for each $0 \leq i < n$, and $q_0 = q^{init}$, the initial state of M. The run is accepting if $q_n \in F$. If the DFA is complete, every word has a unique run. On a trim DFA, each word either a has unique run, or it has no run. The language $\mathcal{L}(M)$ of DFA M, is the set of words for which M has an accepting run.

Nerode equivalence and minimality of DFAs. We will now recall some useful facts about minimality of DFAs. Here, by minimality, we mean DFAs with the least number of states. Every complete DFA M induces an equivalence $\sim_M$ over words: $u \sim_M v$ if M reaches the same state on reading both u and v from the initial state. In the case of trim DFAs, this equivalence can be restricted to set of prefixes of words in $\mathcal{L}(M)$. For a regular language L, we have the Nerode equivalence: $u \approx_L v$ if for all $w \in \Sigma^*$, we have $uw \in L$ iff $vw \in L$. By the well-known Myhill-Nerode theorem (see [HMU07] for more details), there is a canonical DFA M_L with the least number of states for L, and $\sim_{M_L}$ equals the Nerode equivalence $\approx_L$. Furthermore, every DFA M for L is a *refinement* of M_L: $u \sim_M v$ implies $u \sim_{M_L} v$. If two words reach the same state in M, they reach the same state in M_L.

Definition 2.3 (Deterministic Generalized Automata (DGA))**.** A *Deterministic Generalized Automaton (DGA)* [GM99] H is given by $(Q, \Sigma, q^{init}, E, F)$ where Q, q^{init}, F mean the same as in DFA, and $E \subseteq Q \times \Sigma^+ \times Q$ is a finite set of edges labeled with words from Σ^+. For every state q, the set $\{\alpha \mid (q, \alpha, q') \in E\}$ is a prefix-free set.

A run of DGA H on a word w is a sequence of edges $(q_0, \alpha_1, q_1)(q_1, \alpha_2, q_2) \dots (q_{n-1}, \alpha_n, q_n)$ such that $w = \alpha_1 \alpha_2 \dots \alpha_n$, with q_0 being the initial state. As usual, the run is accepting if

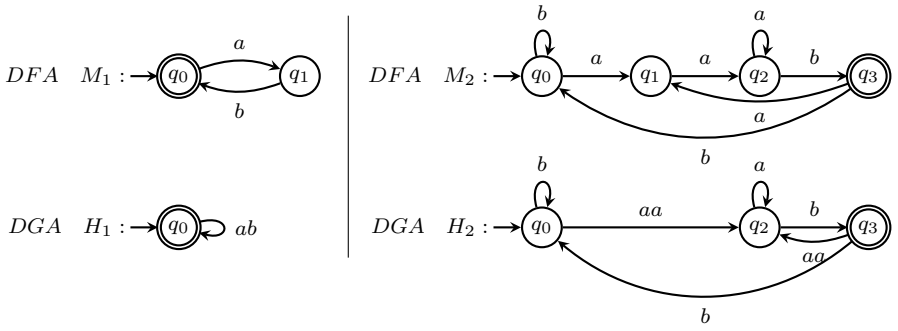


FIGURE 2.1: Examples of DFAs and corresponding DGAs, over alphabet $\{a, b\}$.

$q_n \in F$. Due to the property of the set of outgoing labels being a prefix-free set, there is a atmost one run on every word. The language $\mathcal{L}(H)$ is the set of words with an accepting run. Figure 2.1 gives examples of DFAs and corresponding DGAs.

It was shown in [GM99] that there is no unique smallest DGA. The paper defines an operation to suppress states and create longer labels. A state of a DGA is called *superfluous* if it is neither the initial nor final state, and it has no self-loop. For example, in Figure 2.1, in M_1 and M_2, state q_1 is superfluous. Such states can be removed, and every pair $p \xrightarrow{\alpha} q$ and $q \xrightarrow{\beta} r$ can be replaced with $p \xrightarrow{\alpha\beta} r$ when q is superfluous. This operation is extended to a set of states: given a DGA H, a set of states S, a DGA $\mathcal{S}(H, S)$ is obtained by suppressing states of S, one after the other, in any arbitrary order. For correctness (i.e. language equivalence with the original DGA), there should be no cycle in the induced subgraph of H restricted to S. The paper proves that minimal DGAs (in number of states) can be derived by suppressing states, starting from the canonical DFA.

This chapter has provided the necessary formal preliminaries, defining the classical DFA model and the closely related DGA. With this foundational framework in place, we have established a common language and a basis for comparison. We are now prepared to move from these traditional prefix-matching models to the central contribution of this work. The following chapter will introduce the Deterministic Suffix-reading Automaton (DSA), presenting its formal syntax, its unique suffix-based operational semantics, and an initial analysis of its expressive power and succinctness.

Chapter 3

A new automaton model – Deterministic Suffix-Reading Automata

The preceding chapters established that while Deterministic Finite Automata (DFAs) are the canonical model for regular languages, their practical application is often limited by the state-space explosion problem. This has motivated a long-standing search in automata theory for more succinct, yet still deterministic, models of computation.

One prominent approach to achieving conciseness is to allow transitions to be labeled with strings or more complex patterns instead of single symbols, as seen in models like Deterministic Generalized Automata (DGA). As we saw in the previous chapter, in a DGA, a transition $q \xrightarrow{\alpha} q'$ is taken from state q only if the remaining input word has the string α as a prefix. The word α is then consumed, and computation continues from state q'. This prefix-matching semantic is effective for collapsing linear chains of DFA states into a single transition. However, it is inherently rigid; a DGA cannot ignore or "skip" over intermediate segments of the input stream. This makes it ill-suited for specifying behaviors where an action should be triggered by a specific pattern appearing anywhere in the input, irrespective of what precedes it. DEAs, which use regular expressions, offer more power but come with significant restrictions on the expressions to maintain determinism, ultimately limiting their general applicability.

This chapter introduces a fundamentally different paradigm for string-based automata: ‘suffix-reading’. Instead of actively consuming prefixes from the remaining input, our model

operates on a "wait-and-jump" principle based on the suffixes of the word read so far. The core conceptual shift is from asking, "Does the start of the remaining input match my transition label?" (the DGA approach) to asking, "Does the input processed thus far end with one of my trigger patterns?".

In a Deterministic Suffix-reading Automaton (DSA), a state does not just represent the endpoint of processing a given set of strings, but rather enables a mode of "waiting" for a specific set of patterns to appear. This allows the automaton to passively consume long sequences of irrelevant inputs, only becoming active when a meaningful, pre-defined pattern is detected as a suffix of the history. This approach is designed to more naturally capture the logic of specifications that are "pattern-intensive."

Structure of this chapter. In the sections that follow, we will formalize this intuition.

- In Section 3.1 we will provide the complete syntax for DSAs, followed by a rigorous definition of their operational semantics, which resolves ambiguity through rules for earliest and longest matching patterns.
- In Section 3.2 we will then establish the model's theoretical properties, proving that it retains the full expressive power of DFAs for regular languages and analyzing its potential for providing significantly more succinct specifications.

3.1 DSA: formal syntax and semantics

We have seen an example of a deterministic suffix automaton in Figure 1.1. A DSA consists of a set of states, and a finite set of outgoing labels at each state. On an input word w, the DSA finds the earliest prefix which ends with an outgoing label of the initial state, erases this prefix and goes to the target state of the transition with the matching label. Now, the DSA processes the rest of the word from this new state in the same manner. In this section, we will formally describe the syntax and semantics of DSA.

We start with some more examples. Figure 3.1 shows a DSA for $L_2 = \Sigma^* aab$, the same language as the automata M_2 and H_2 of Figure 2.1. At q_0, DSA $\mathcal{A}_2$ waits for the first occurrence

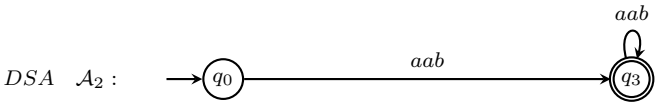


FIGURE 3.1: DSA $\mathcal{A}_2$ accepts $L_2 = \Sigma^* aab$, with $\Sigma = \{a, b\}$.

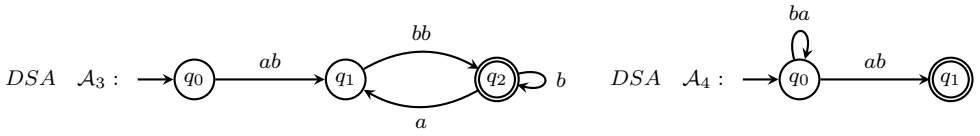


FIGURE 3.2: $\mathcal{A}_3$ accepts $L_3 = \Sigma^* ab \Sigma^* bb$ and $\mathcal{A}_4$ accepts $L_4 = (b^* ba)^* a^* ab$.

of aab and as soon as it sees one, it transitions to q_3. Here, it waits for further occurrences of aab. For instance, on the word $abbaabbbaab$, it starts from q_0 and reads until $abbaab$ to move to q_3. Then, it reads the remaining $bbaab$ to loop back to q_3 and accepts. On a word $baabaa$, the automaton moves to q_3 on $baab$, and continues reading aa, but having nowhere to move, it makes no transition and rejects the word. Consider another language $L_3 = \Sigma^* ab \Sigma^* bb$ on the same alphabet Σ. A similar machine (as $\mathcal{A}_2$) to accept L_3 would look like $\mathcal{A}_3$ depicted in Fig. 3.2. For example, on the word $abbbb$, it would read until ab and move from q_0 to q_1, read further until bb and move to q_2, then read b and move back to q_2 to accept. We can formally define such machines as automata that transition on suffixes, or suffix-reading automata.

Definition 3.1 (DSA)**.** A *deterministic suffix-reading automaton (DSA)* $\mathcal{A}$ is given by a tuple $(Q, \Sigma, q^{init}, \Delta, F)$ where Q is a finite set of states, Σ is a finite alphabet, $q^{init} \in Q$ is the initial state, $\Delta \subseteq Q \times \Sigma^+ \times Q$ is a finite set of transitions, $F \subseteq Q$ is a set of accepting states. For a state $q \in Q$, we define $\mathrm{Out}(q) := \{\alpha \mid (q, \alpha, q') \in \Delta \text{ for some } q' \in Q\}$ for the set of labels present in transitions out of q. No state has two outgoing transitions with the same label: if $(q, \alpha, q') \in \Delta$ and $(q, \alpha, q'') \in \Delta$, then $q' = q''$.

The (total) size $|\mathcal{A}|$ of DSA $\mathcal{A}$ is defined as the sum of the number of states, the number of transitions, and the size $|\mathrm{Out}(q)|$ for each $q \in Q$, where $|\mathrm{Out}(q)| := \sum_{\alpha \in \mathrm{Out}(q)} |\alpha|$.

As mentioned earlier, at a state q the automaton waits for a word that ends with one of its outgoing labels. If more than one label matches, then the transition with the longest label is taken. For example, consider the DSA in Figure 1.1. At state s_1 on reading $fghendif$, both the `if` and `endif` transitions match. The longest match is `endif` and therefore the DSA

moves to s_0. This gives a deterministic behaviour to the DSA. More precisely: at a state q, it reads w to fire (q, α, q') if α is the longest word in $\mathrm{Out}(q)$ which is a suffix of w, and no proper prefix of w has any label in $\mathrm{Out}(q)$ as suffix. We call this a 'move' of the DSA. For example, consider $\mathcal{A}_4$ of Figure 3.2 as a DSA. Let us denote $t := (q_0, ab, q_1)$ and $t' := (q_0, ba, q_1)$. We have moves (t, ab), (t, aab), $(t, aaab)$, and (t', ba), (t', bba), etc. In order to make a move on t, the word should end with ab and should have neither ab nor ba in any of its proper prefixes.

Definition 3.2. A *move* of DSA $\mathcal{A}$ is a pair (t, w) where $t = (q, \alpha, q') \in \Delta$ is a transition of $\mathcal{A}$ and $w \in \Sigma^+$ such that

- α is the longest word in $\mathrm{Out}(q)$ which is a suffix of w, and
- no proper prefix of w has a label in $\mathrm{Out}(q)$ as suffix (no label is a factor of w).

A move (t, w) denotes that at state q, transition t gets triggered on reading word w. We will also write $q \xrightarrow[\alpha]{w} q'$ for the move (t, w).

Whether a word is accepted or rejected is determined by a 'run' of the DSA on it.

Definition 3.3. A run of $\mathcal{A}$ on word w, starting from a state q, is a sequence of moves that consume the word w, until a (possibly empty) suffix of w remains for which there is no move possible.

Formally, a run is a sequence of moves $q_i \xrightarrow[\alpha_i]{w_i} q_{i+1}$ for $0 \leq i \leq m$ and $q_m \xrightarrow{w_m}$ where $q = q_0$ and m is the index of the last state in the sequence i.e. $q = q_0 \xrightarrow[\alpha_0]{w_0} q_1 \xrightarrow[\alpha_1]{w_1} \cdots \xrightarrow[\alpha_{m-1}]{w_{m-1}} q_m \xrightarrow{w_m}$ such that $w = w_0 w_1 \dots w_{m-1} w_m$, and $q_m \xrightarrow{w_m}$ denotes that there is no move using any outgoing transition from q_m on w_m or any of its prefixes.

The run is accepting if $q_m \in F$ and $w_m = \varepsilon$ (no dangling letters in the end). The language $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of all words that have an accepting run starting from the initial state q^{init}.

Naturally, the set of words with accepting runs gives the language of the DSA. Moreover, due to our "move" semantics, there is a unique run for every word.

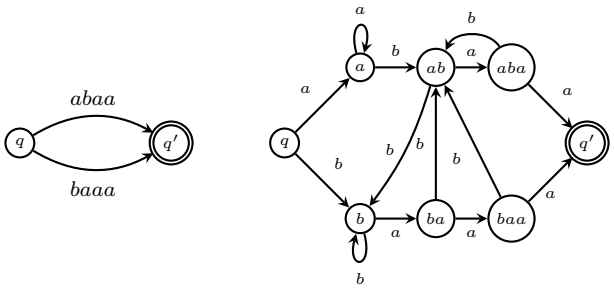


FIGURE 3.3: A DSA on the left, and the corresponding DFA for matching the strings $abaa$ and $baaa$.

3.2 Comparison with DFA and DGA

Every complete DFA can be seen as an equivalent DSA — since $\mathrm{Out}(q) = \Sigma$ for every state, the equivalent DSA is forced to move on each letter, behaving like the DFA that we started off with. For the DSA-to-DFA direction, we associate a specific DFA to every DSA, as follows. The idea is to replace transitions of a DSA with a string-matching-DFA for $\mathrm{Out}(q)$ at each state. Figure 3.3 gives an example. The intermediate states correspond to proper prefixes of words in $\mathrm{Out}(q)$.

Definition 3.4 (Tracking DFA for a DSA.)**.** For a DSA $\mathcal{A} = (Q^{\mathcal{A}}, \Sigma, q_{in}^{\mathcal{A}}, \Delta^{\mathcal{A}}, F^{\mathcal{A}})$, we give a DFA $M_{\mathcal{A}}$, called its *tracking DFA*. For $q \in Q^{\mathcal{A}}$, let $\overline{\mathrm{Out}}(q)$ be the set of all prefixes of words in $\mathrm{Out}(q)$. States of $M_{\mathcal{A}}$ are given by: $Q^M = \bigcup_{q \in Q^{\mathcal{A}}} \{\{(q, \beta) \mid \beta \in \overline{\mathrm{Out}}(q)\} \cup \{(q, \overline{\varepsilon})\}\}$, where $\overline{\varepsilon}$ is used as a special symbol to create a copy of each (q, ε).

The initial state is $(q_{in}^{\mathcal{A}}, \varepsilon)$ and final states are $\{(q, \varepsilon) \mid q \in F^{\mathcal{A}}\}$. Transitions are as below: for every $q \in Q^{\mathcal{A}}, \beta \in \overline{\mathrm{Out}}(q), a \in \Sigma$, let β' be the longest word in $\overline{\mathrm{Out}}(q)$ s.t β' is a suffix of βa.

- $(q, \beta) \xrightarrow{a} (q', \varepsilon)$ if $\beta' \in \mathrm{Out}(q)$ and $(q, \beta', q') \in \Delta^{\mathcal{A}}$,
- $(q, \beta) \xrightarrow{a} (q, \beta')$ if $\beta' \notin \mathrm{Out}(q)$ and $\beta' \neq \varepsilon$,
- $(q, \beta) \xrightarrow{a} (q, \overline{\varepsilon})$ if $\beta' = \varepsilon$,
- $(q, \overline{\varepsilon}) \xrightarrow{a} s$, if $(q, \varepsilon) \xrightarrow{a} s$ according to the above (same outgoing transitions).

Intuitively, the tracking DFA implements the transition semantics of DSAs. Starting at (q, ε), the tracking DFA moves along states marked with q as long as no label of $\mathrm{Out}(q)$ is

seen as a suffix. For all such words, the tracking DFA maintains the longest word among $\overline{\mathrm{Out}}(q)$ seen as a suffix so far. For instance, in Figure 3.3, at q on reading word aab, the DFA on the right is in state ab (which is the equivalent of (q, ab) in the tracking DFA definition). To distinguish between reaching q after seeing a full pattern (in which case we may need to accept the word) and "resetting" the word seen so far since there was no matching suffix, we consider the copy $(q, \overline{\varepsilon})$. For example, in Figure 3.3, if we assume there is an additional letter c appears in transition out of q, on reading the word $aabc$, the tracking DFA goes to state $(q, \overline{\varepsilon})$.

Lemma 3.5. *For every DSA $\mathcal{A}$, the language $\mathcal{L}(\mathcal{A})$ equals the language $\mathcal{L}(M_{\mathcal{A}})$ of its tracking DFA.*

Proof. The tracking DFA satisfies the following invariants on every state (q, ε) (we make use of notations from Definition 2.1) :

- Let w be a word that has no $\alpha \in \mathrm{Out}(q)$ as its factor. Then, the run of w starting at (q, ε), remains in states of the form (q, β) and ends in state (q, β_w) where $\beta_w \sqsubseteq_{\mathsf{lsf}}^{\overline{\mathrm{Out}}(q)} w$, when $\beta_w \neq \varepsilon$; if $\beta_w = \varepsilon$, then it ends in state $(q, \overline{\varepsilon})$.
- Let w be a word such that $q \xrightarrow[\alpha]{w} q'$ is a move of $\mathcal{A}$. Then the run of $M_{\mathcal{A}}$ on w starting at (q, ε) remains in states of the form (q, β) and ends in (q', ε).
- Let w be a word such that the run of $M_{\mathcal{A}}$ starting at (q, ε) remains in states of the form (q, β) and ends in a state (q, β_w). If $\beta_w \neq \overline{\varepsilon}$, then $\beta_w \sqsubseteq_{\mathsf{lsf}}^{\overline{\mathrm{Out}}(q)} w$. If $\beta_w = \varepsilon$, then there is no suffix of w in $\overline{\mathrm{Out}}(q)$.
- Let w be a word such that the run of $M_{\mathcal{A}}$ starting at (q, ε) ends in (q', ε), and all intermediate states are of the form (q, β). Then there is a move $q \xrightarrow[\alpha]{w} q'$ in $\mathcal{A}$ where $\alpha \sqsubseteq_{\mathsf{lsf}}^{\mathrm{Out}(q)} w$.

All these invariants can be proved by induction on the length of the word w and making use of the way the transitions have been defined in the tracking DFA. The first two invariants show that every accepting run of $\mathcal{A}$ has a corresponding accepting run in $M_{\mathcal{A}}$, thereby proving $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(M_{\mathcal{A}})$. The next two invariants show that every accepting run of $M_{\mathcal{A}}$ corresponds to an accepting run of $\mathcal{A}$, thereby showing $\mathcal{L}(M_{\mathcal{A}}) \subseteq \mathcal{L}(\mathcal{A})$. □

Lemma 3.5 and the fact that every complete DFA is also a DSA, prove that DSAs recognize regular languages. We will now compare succinctness of DSA wrt DFA and DGA, starting with a family of languages for which DSAs are concise.

Lemma 3.6. *Let $\Sigma = \{a_1, a_2, \ldots, a_n\}$ be an alphabet for some $n \geq 1$. Consider the language $L_n = \Sigma^* a_1 a_2 \ldots a_n$. There is a DSA for this language with size $(4 + 2|\Sigma|)$. Any trim DFA for L_n has size at least $|\Sigma|^2$ (Note: size is counted as the sum of the number of states, edges and length of edge labels, in all the automata) .*

Proof. Consider the DSA with states q_0, q_1 and transitions $q_0 \xrightarrow{a_1 \ldots a_n} q_1$, $q_1 \xrightarrow{a_1 \ldots a_n} q_1$, with q_0 the initial state and q_1 the accepting state. This DSA accepts L_n.

For any pair $a_1 a_2 \ldots a_i$ and $a_1 a_2 \ldots a_j$ with $i \neq j$, there is a distinguishing suffix: $a_1 a_2 \ldots a_i \cdot a_{i+1} \ldots a_n \in L_n$, but $a_1 a_2 \ldots a_j \cdot a_{i+1} \ldots a_n \notin L_n$. Therefore, the strings a_1, $a_1 a_2$, $\ldots$, $a_1 \ldots a_n$ go to different states in the canonical DFA. This shows there are at least n states in the minimal DFA. Now, from $a_1 a_2 \ldots a_j$, there is a transition of every letter: clearly, there needs to be a transition on a_{j+1}; if there is no transition on, say $a_\ell \neq a_{j+1}$, then the word $a_1 a_2 \ldots a_j a_\ell a_1 a_2 \ldots a_n$ will be rejected. A contradiction. Hence, there are n transitions from each state. □

We make a remark about DGAs for the language family in Lemma 3.6. Notice that suppressing superfluous states of the minimal DFA leads to a strict increase in the label length, which only strictly increases the (total) size. For instance, suppose $q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} q_3$ is a sequence of transitions in the minimal DFA for L_n. There are additional transitions $q_2 \xrightarrow{a_1} q_1$ and $q_2 \xrightarrow{\Sigma \setminus \{a_1, a_3\}} q_0$ (where q_0 is the initial state). Suppressing q_2 leads to edges with labels $a_2 a_3$, $a_2 a_1$, $a_2 c$ for each $c \in \Sigma \setminus \{a_1, a_3\}$. Notice that a_2 gets copied $|\Sigma|$ many times. Therefore suppressing states from the minimal DFA is not helpful in getting a smaller DGA. However, using this argument, we cannot conclude that the minimal DGA has size at least n^2. Since a characterization of minimality of DGAs in terms of total size is not known, we leave it at this remark.

We now state the final result of this section, which summarizes the size comparison between DSAs, DFAs, DGAs. For the comparison to DFAs, we use the fact that every DSA of

size k can be converted to its tracking DFA, which has atmost $2k$ states. Therefore, size of the tracking DFA is bounded by $2k$ (states) $+2k \cdot |\Sigma|$ (edges) $+2k \cdot |\Sigma|$ (label length), which comes to $2k(1 + 2|\Sigma|)$.

For a regular language L, let $n_F^{cmp}(L), n_F^{trim}(L), n_G^{trim}(L), n_S(L)$ denote the size of the minimal complete DFA, minimal trim DFA, minimal trim DGA and minimal DSA respectively, where size is counted as the sum of the number of states, edges and length of edge labels, in all the automata.

Theorem 3.7. *We have:*

1. *For all L, we have* $\dfrac{n_F^{cmp}(L)}{2(1+2|\Sigma|)} \leq n_S(L) \leq n_F^{cmp}(L)$
2. *There is a language for which $n_S(L)$ is the smallest, and another language for which $n_S(L)$ is the largest of the three.*

Proof. A complete DFA can be seen as a DSA accepting the same language. This gives us $n_S(L) \leq n_F^{cmp}(L)$. For the inequality $\dfrac{n_F^{cmp}(L)}{2(1+2|\Sigma|)} \leq n_S(L)$, we show that the tracking DFA (Definition 3.4) of a DSA has size at most $n_S(L) \cdot 2(1 + 2|\Sigma|)$. The number of states of the tracking DFA is atmost $2n_S(L)$ (recall that $n_S(L)$ denotes the total size of the DSA, which includes the label lengths). For each state there are Σ transitions. Therefore, total size is $2n_S(L)$ (states) + $2n_S(L) \cdot |\Sigma|$ (edges) + $2n_S(L) \cdot |\Sigma|$ (label lengths), which equals $2n_S(L)(1 + 2|\Sigma|)$.

For the second part of the proof, Lemma 3.6 gives an example where $n_S(L)$ is the smallest. For the other direction, consider the language $L_1 = (ab)^*$. A trim DFA for this language has two states q_0, q_1 (with q_0 accepting), and two transitions $q_0 \xrightarrow{a} q_1$, and $q_1 \xrightarrow{b} q_0$. Therefore $n_F^{trim}(L_1) \leq 2 + 2 + 2 = 6$. A trim DGA for this language is $q_0 \xrightarrow{ab} q_0$. Therefore $n_G^{trim}(L_1) \leq 2 + 1 + 2 = 5$. We first claim that any DSA for this language needs to maintain the following information: (1) the initial state which is accepting, as ε is in the language; (2) there is a path from the initial state on ab to an accepting state (otherwise ab will not be accepted); (3) on reading aa or b from the initial state, the DSA has to make some transitions and go to a sink state, which is a non-accepting state — otherwise, the words aab or bab will get accepted. This

shows that any DSA has at least two states, at least three transitions, and ab, b and aa split across some transition labels. This gives either: $n_S(L_1) \geq 2+2+4 = 8$ for the DSA with two states and $\xrightarrow{ab}$, $\xrightarrow{b}$ and $\xrightarrow{aa}$ as transitions, which is bigger than the corresponding trim DFA and DGA, or the min DFA viewed as DSA, with 3 states, and transitions on each of a and b from each of the states giving $n_S \geq 3 + 6 + 6 = 15$ □

Concluding remarks for Chapter 3. This chapter has laid the complete theoretical groundwork for Deterministic Suffix-reading Automata. We began with a formal syntax in **Definition 3.1**, introducing not only the structure of the automaton but also the **total-size** metric, a more faithful measure of descriptive complexity for models with string-based transition labels. The core idea of the model was detailed through its unique operational semantics, where we established how a DSA defines a deterministic run by adhering to two principles: acting on the **earliest** occurrence of a recognizable pattern, and in cases of ambiguity, choosing the transition corresponding to the **longest matching suffix**.

One main observation of this chapter was the proof that DSAs recognize exactly the class of regular languages. This was established constructively through the **tracking DFA** construction (Definition 3.4), a method for converting any DSA into an equivalent, standard DFA. This result ensures that our model is a new representation for a well-understood class of languages, rather than an extension into non-regular territory. Finally, we quantified the primary motivation for this new model: **succinctness**. By analyzing a family of languages ($L_n = \Sigma^* a_1 a_2 \dots a_n$), we formally demonstrated that a DSA can be exponentially more concise in total size than the minimal DFA for the same language, highlighting its potential for modeling pattern-intensive specifications.

With the model's theoretical properties and potential benefits now established, the focus naturally shifts from the abstract to the constructive. A formal model is only as useful as our ability to create instances of it for practical problems. This leads to the central question that will drive the next chapter: Given a regular language, typically represented by a DFA, how can we derive a language-equivalent and concise DSA? We will answer this by presenting a systematic DFA-to-DSA conversion procedure. This investigation will, in turn, unveil

some counter-intuitive challenges of DSA minimization, revealing fundamental ways in which suffix-reading automata differ from their classical counterparts.

Chapter 4

Suffix-tracking sets – deriving DSA from DFA

The previous chapter formally defined Deterministic Suffix-reading Automata and demonstrated their potential for providing succinct representations of regular languages. However, a model's practical utility hinges on our ability to systematically construct it for a given language. This chapter addresses this question:

Problem of interest. How does one derive a correct and concise DSA from a standard representation like a DFA?

Our approach is centered on a form of abstraction, or state suppression. The core intuition is to "downsample" a given DFA by selecting a subset of its states to serve as the primary "macro-states" of the new DSA, while the paths between them become the new, string-based transitions. This general strategy is not new; a similar state-suppression method exists for generating Deterministic Generalized Automata [GM99]. However, the challenge is significantly greater for DSAs. The sophisticated, non-local semantics of suffix-reading mean that the suppressed states and the paths through them must collectively behave as a single, consistent suffix-matcher. Naively collapsing paths without respecting this underlying semantic requirement can alter the accepted language.

For the derivation to be sound, the collection of paths that will become the new DSA

transitions must satisfy certain consistency requirements. An analogy can be drawn to ensuring a complex electronic circuit is functional: it is not enough for the individual wires to be well-made; they must also be connected correctly at their terminals to avoid ambiguity and short-circuits. Our derivation procedure rests on two such conditions.

First, we must ensure that the paths are **internally consistent**. This is the role of what we call a 'suffix-compatibility' condition. It guarantees that as we trace a path through the suppressed part of the DFA, the logic of suffix-matching is maintained correctly at every single step. This is akin to checking that each wire in the circuit functions correctly along its entire length, faithfully transmitting signals.

Second, we must ensure the paths are consistent with each other at their **endpoints**—the selected macro-states. This is the purpose of what we call a 'well-formedness' condition. It prevents ambiguity by forbidding situations where a path leading to a chosen macro-state could be mistaken for part of a different path leading to a suppressed state. This is analogous to checking that all wires are connected to the correct terminals in the circuit, ensuring that signals arrive at their intended destinations without interference.

This chapter will formalize these ideas, culminating in the definition of a 'suffix-tracking set': a set of states that satisfies both of these conditions. The main theorem will then prove that any DSA derived from such a set is guaranteed to be sound and language-equivalent to the source DFA.

For DGAs, a method to derive smaller DGAs by suppressing states was recalled in Section 2. Our goal is to investigate a similar procedure for DSAs. The DSA model creates new challenges. Suppressing states may not always lead to smaller automata (in total size). Figure 4.1 illustrates an example where suppressing states leads to an exponentially larger automaton, due to the exponentially many paths created. But, suppressing states may sometimes indeed be useful: in Figure 4.2, the DFA on the left is performing a string matching to deduce the pattern ab. On seeing ab, it accepts. Any extension is rejected. This is succinctly captured by the DSA on the right. Notice that the DSA is obtained by suppressing states q_1 and q_3. So, suppressing states may sometimes be useful and sometimes not. In [GM99], the focus was

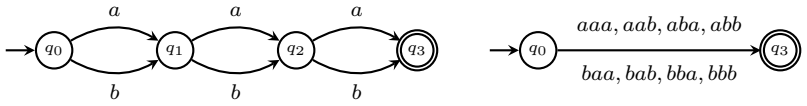


FIGURE 4.1: Suppressing states can add exponentially many labels and increase total size.

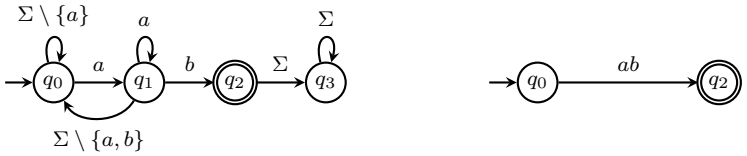


FIGURE 4.2: Suppressing states can sometimes reduce total size

on getting a DGA with minimal number of states, and hence suppressing states was always useful.

More importantly, when can we suppress states? DGAs cannot "ignore" parts of the word. This in particular leads to the requirement that a state with a self-loop cannot be suppressed. DSAs have a more sophisticated transition semantics. Therefore, the procedure to suppress states is not as simple. This is the subject of this section. We deviate from the DGA setting in two ways: we will select a subset of good states from which we can construct a DSA (essentially, this means the rest of the states are suppressed); secondly, our starting point will be complete DFA, on which we make the choice of states (in DGAs, one could start with any DGA and suppress states).

Structure of this chapter. Our procedure can be broken down into two steps, which we will describe in the two sections of this chapter.

- In Section 4.1, we start from a complete DFA, select a subset of states and build an induced DSA by connecting states using acyclic paths between them.

- In Section 4.2, we show how to remove some redundant transitions. In the end we obtain a potentially more concise DSA from the initial DFA.

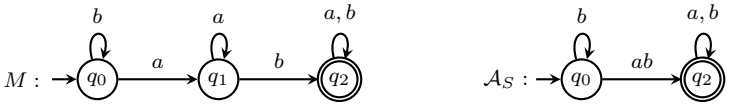


FIGURE 4.3: DFA M and an equivalent DSA $\mathcal{A}_S$ 'induced' with $S = \{q_0, q_2\}$.

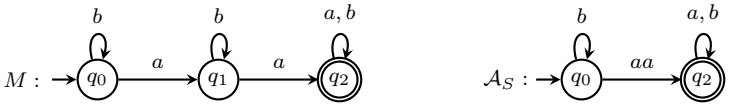


FIGURE 4.4: DFA M and DSA $\mathcal{A}_S$ 'induced' with $S = \{q_0, q_2\}$. Not equivalent.

4.1 Building an induced DSA

We start with an illustrative example. Consider DFA M in Figure 4.3. The DSA on the right of the figure shows such an induced DSA obtained by marking states $\{q_0, q_2\}$ and connecting them using simple paths. Notice that the language of the induced DSA and the original DFA are same in this case. Intuitively, all words that end with an a land in q_1. Hence, q_1 can be seen to "track" the suffix a. Now, consider Figure 4.4. We do the same trick, by marking states $\{q_0, q_2\}$ and inducing a DSA. Observe that the DSA does not accept aba, and hence is not language equivalent. When does a subset of states induce a language equivalent DSA? Roughly, this is true when the states that are suppressed track "suitable suffixes" (a reverse engineering of the tracking DFA construction of Definition 3.4). As we will see, the suitable suffixes will be the simple paths from the selected states to the suppressed states. We begin by formalizing these ideas and then present sufficient conditions that ensure language equivalence of the resulting DSA.

Definition 4.1 (Simple words)**.** Consider a complete DFA $M = (Q, \Sigma, q^{init}, \Delta, F)$. Let $S \subseteq Q$ be a subset of states, and $p, q \in Q$. We define $\mathsf{SP}(p \rightsquigarrow q, S)$, the *simple words from* p *to* q *modulo* S, as the set of all words $a_1 a_2 \ldots a_n \in \Sigma^+$ such that there is a path: $p = p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \cdots p_{n-1} \xrightarrow{a_n} p_n = q$ in M where

- no intermediate state belongs to S: $\{p_1, \ldots, p_{n-1}\} \subseteq Q \setminus S$, and
- there is no intermediate cycle: if $p_i = p_j$ for some $0 \leq i < j \leq n$, then $p_i = p_0$ and $p_j = p_n$.

We write $\mathsf{SP}(p, S)$ for $\bigcup_{q \in Q} \mathsf{SP}(p \rightsquigarrow q, S)$, the set of all simple words modulo S, emanating

from p.

For example, in Figure 4.3, with $S = \{q_0, q_2\}$, we have $\mathsf{SP}(q_0 \rightsquigarrow q_1, S) = \{a\}$, $\mathsf{SP}(q_0 \rightsquigarrow q_0, S) = \{b\}$ and $\mathsf{SP}(q_0 \rightsquigarrow q_2, S) = \{ab\}$. These are the same in Figure 4.4, except $\mathsf{SP}(q_0 \rightsquigarrow q_2, S) = aa$.

Here are some preliminary technical lemmas. Due to determinism of M, we get the following property, which underlies several arguments that come later.

Lemma 4.2. *Let $S \subseteq Q$, and $p, q, r \in Q$ s.t. $q \neq r$. We have $\mathsf{SP}(p \rightsquigarrow q, S) \cap \mathsf{SP}(p \rightsquigarrow r, S) = \emptyset$.*

The next lemma says that every transition either extends a simple path or is a "back-edge" leading to an ancestor in the path.

Lemma 4.3. *Let $S \subseteq Q$ and $p, q, u \in Q$ $(q \notin S, p \neq q)$ such that $q \xrightarrow{a} u$ is a transition. For every $\sigma \in \mathsf{SP}(p \rightsquigarrow q, S)$, either σa belongs to $\mathsf{SP}(p \rightsquigarrow u, S)$, or some proper prefix of σa belongs to $\mathsf{SP}(p \rightsquigarrow u, S)$.*

Proof. Since $\sigma \in \mathsf{SP}(p \rightsquigarrow q, S)$, there is a path $p = p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \cdots p_{n-1} \xrightarrow{a_n} p_n = q$, with $\sigma = a_1 a_2 \ldots a_n$, satisfying the conditions of Definition 4.1. If $u \neq p_i$ for $0 \leq i \leq n$, then $\sigma a \in \mathsf{SP}(p \rightsquigarrow u, S)$ since $q \notin S$. If $u = p_i$ for some $0 < i \leq n$, then the prefix $a_1 a_2 \ldots a_{i-1} \in \mathsf{SP}(p \rightsquigarrow u, S)$. If $u = p_0$, then we have $\sigma a \in \mathsf{SP}(p \rightsquigarrow u, S)$. □

Fix a complete DFA M for this section. A DSA can be 'induced' from M using S, by fixing states to be S (initial and final states retained) and transitions to be the simple words modulo S connecting them i.e. $p \xrightarrow{\sigma} q$ if $\sigma \in \mathsf{SP}(p \rightsquigarrow q, S)$ (Figure 4.3).

Definition 4.4 (Induced DSA)**.** Given a DFA M and a set S of states in M that contains the initial and final states, we define the induced DSA of M (using S). The states of the induced DSA are given by S. The initial and final states are the same as in M. The transitions are given by the simple words modulo S i.e. $p \xrightarrow{\sigma} q$ if $\sigma \in \mathsf{SP}(p \rightsquigarrow q, S)$, for every pair of states $p, q \in S$.

The induced DSA may not be language-equivalent (Figure 4.4); to ensure that, we need to check some conditions. Here is a central definition.

Definition 4.5 (Suffix-compatible transitions)**.** Fix a subset $S \subseteq Q$. A transition $q \xrightarrow{a} u$ is suffix-compatible w.r.t. S if either $q \in S$ or $u \in S$ **OR** for all $p \in S$, and for every $\sigma \in \mathsf{SP}(p \rightsquigarrow q, S)$, there is an $\alpha \in \mathsf{SP}(p \rightsquigarrow u, S)$ s.t.:

- α is a suffix of σa, and
- moreover, α is the longest suffix of σa among words in $\mathsf{SP}(p, S)$.

Note that a transition $q \xrightarrow{a} u$ is trivially suffix-compatible if $q \in S$ or $u \in S$. The rest of the condition only needs to be checked when both of $q, u \notin S$. In Figure 4.4, we find the self-loop at q_1 to not be suffix-compatible: we have $S = \{q_0, q_2\}$, and $\mathsf{SP}(q_0 \rightsquigarrow q_1, S) = \{a\}$, $\mathsf{SP}(q_0, S) = \{b, a, ab\}$; the transition $q_1 \xrightarrow{b} q_1$ is not suffix-compatible since there is no suffix of ab in $\mathsf{SP}(q_0 \rightsquigarrow q_1, S)$. Whereas in Figure 4.3, the loop is labeled a instead of b. The transition $q_1 \xrightarrow{a} q_1$ is suffix-compatible, since the longest suffix of aa among $\mathsf{SP}(q_0, S)$ is a and it is present in $\mathsf{SP}(q_0 \rightsquigarrow q_1, S)$. Let us take the DFA in the right of Figure 3.3, and let $S = \{q, q'\}$. Here are some of the simple path sets: $\mathsf{SP}(q \rightsquigarrow ab, S) = \{ab, bab, baab\}$, $\mathsf{SP}(q \rightsquigarrow aba, S) = \{aba, baba, baaba\}$. Consider the transition $aba \xrightarrow{b} ab$. It can be verified that for every $\sigma \in \mathsf{SP}(q \rightsquigarrow aba, S)$, the longest suffix of the extension σb, among simple paths out of q, indeed lies in the state ab. In fact, all transitions satisfy suffix-compatibility w.r.t. the chosen set S.

The suffix-compatibility condition is described using simple paths to states. It requires that every transition take each simple word reaching its source to the state tracking the longest suffix of its one-letter extension. This condition on simple words, transfers to all words, that circle around the suppressed states. In Figure 3.3, this property can be verified by considering the word $bbabab$ and its run: $q \xrightarrow{b} b \xrightarrow{b} b \xrightarrow{a} ba \xrightarrow{b} ab \xrightarrow{a} aba \xrightarrow{b} ab$. At each step, the state reached corresponds to the longest suffix among the simple words out of q. In the next two lemmas, we prove this claim.

We will use a special notation: for a state $p \in S$, we write $\mathrm{Out}(p, S)$ for $\bigcup_{r \in S} \mathsf{SP}(p \rightsquigarrow r, S)$; these are the simple words that start at p and end in some state r of S. Notice that these are the words that appear as transitions in the induced DSA. In particular, $\mathrm{Out}(p)$ in the induced DSA equals $\mathrm{Out}(p, S)$. We also remark that $\mathrm{Out}(p, S)$ is different from $\mathsf{SP}(p, S)$:

the latter considers simple words from p to all states in Q, whereas the former only considers words from p to S.

Lemma 4.6. *Let S be a set of states such that every transition of M is suffix-compatible w.r.t. S. Pick $p \in S$, and let $w \in \Sigma^+$ be a word with a run $p = p_0 \xrightarrow{w_1} p_1 \xrightarrow{w_2} p_2 \dots p_{n-1} \xrightarrow{w_n} p_n$ such that the intermediate states $p_1, \dots, p_{n-1}$ belong to $Q \setminus S$. The state p_n may or may not be in S. Then:*

- *no proper prefix of w has any word from $\mathrm{Out}(p, S)$ as suffix (no factor of w is in $\mathrm{Out}(p, S)$), and*
- *there is $\alpha \in \mathsf{SP}(p \rightsquigarrow p_n, S)$ such that α is the longest suffix of w among words in $\mathsf{SP}(p, S)$.*

Proof. We prove this by induction on the length of the word w. When $w = a$ for $a \in \Sigma$, we have the run $p \xrightarrow{a} q$. The first conclusion of the lemma is vacuously true, since there is no non-empty proper prefix of a. For the second conclusion, note that, by definition, we have $a \in \mathsf{SP}(p \rightsquigarrow q, S)$ (in both cases when $q \neq p$ and $q = p$). Moreover, clearly a is the longest suffix of a.

Now, let $w = w'a$ with a run $p \xrightarrow{w'} p' \xrightarrow{a} p_n$ such that p' and the intermediate states while reading w' belong to $Q \setminus S$. Assume the lemma holds for w'. Therefore, the longest suffix σ' of w', among $\mathsf{SP}(p, S)$, belongs to $\mathsf{SP}(p \rightsquigarrow p', S)$, that is: $\sigma' \sqsubseteq_{\mathsf{lsf}}^{\mathsf{SP}(p,S)} w'$ and $\sigma' \in \mathsf{SP}(p \rightsquigarrow p', S)$ (notation as in Definition 2.1). We claim that the longest suffix of $w'a$ is in fact the longest suffix of $\sigma'a$. If not: there is $\sigma''a$ with $|\sigma''| > |\sigma'|$ such that $\sigma''a \sqsubseteq_{\mathsf{lsf}}^{\mathsf{SP}(p,S)} w'a$. Therefore $\sigma'' \sqsubseteq_{\mathsf{lsf}}^{\mathsf{SP}(p,S)} w'$. This contradicts $\sigma' \sqsubseteq_{\mathsf{lsf}}^{\mathsf{SP}(p,S)} w'$. If $p_n \in Q \setminus S$, the longest suffix of $\sigma'a$ lies in p_n, by suffix-compatibility (Definition 4.5). If $p_n \in S$, we will have the exact word $\sigma'a \in \mathsf{SP}(p \rightsquigarrow p_n, S)$.

□

Lemma 4.7. *Let S be a set of states such that every transition of M is suffix-compatible w.r.t. S. Let $p \in S$, and $w \in \Sigma^+$ be a word such that no proper prefix of w has a word from $\mathrm{Out}(p, S)$ as suffix (no factor of w is in $\mathrm{Out}(p, S)$). Then:*

- *The run of M starting from p, is of the form $p \xrightarrow{w_1} p_1 \xrightarrow{w_2} p_2 \dots p_{n-1} \xrightarrow{w_n} p_n$ where $\{p_1, \dots, p_{n-1}\} \subseteq Q \setminus S$ (notice that we have not included p_n, which may or may not be in S).*

- *the longest suffix of w, among $\mathsf{SP}(p, S)$ lies in $\mathsf{SP}(p \rightsquigarrow p_n, S)$.*

Proof. We prove this by induction on the length of the word w. Suppose $w = a$, for $a \in \Sigma$. Since M is complete, there is a transition $p \xrightarrow{a} p_1$. The first item is vacuously true. Moreover, if there is $q \in S$ and $\alpha \in \mathsf{SP}(p \rightsquigarrow q, S)$ such that α is a suffix of a, then clearly $\alpha = a$. Hence $q = p_1$, due to the determinism of the underlying automaton. If there is no such q, then it means $p_1 \notin S$.

Suppose $w = w'a$ satisfies the assumptions of the lemma. Assuming the lemma holds for w', we have a run $p \xrightarrow{w'} p'$ such that p' and all intermediate states lie in $Q \setminus S$. Let $p' \xrightarrow{a} p_n$ be the outgoing transition on a from p'. This gives a path $p \xrightarrow{w'} p \xrightarrow{a} p_n$ satisfying the first part of the lemma. By the second item of Lemma 4.6, the longest suffix $\alpha'a$ of $w'a$ among $\mathsf{SP}(p, S)$ belongs to $\mathsf{SP}(p \rightsquigarrow p_n, S)$. □

Suffix-compatibility alone does not suffice to preserve the language. In Figure 4.5, consider $S = \{0, 2, 4\}$. Every transition is suffix-compatible w.r.t. S. The DSA induced using S is shown in the middle. Notice that it is not language equivalent, due to the word aba for instance. The run of aba looks as follows: $0 \xrightarrow{ab} 4 \xrightarrow{b} 4$. The expected run was $0 \xrightarrow{aba} 2$, but that does not happen since there is a shorter prefix with a matching transition. Even though, we have suffix-compatibility, we need to ensure that there are no "conflicts" between outgoing patterns. This leads to the next definition.

Definition 4.8 (Well-formed set)**.** A set of states $S \subseteq Q$ is well-formed if there is no $p \in S, q \in S$ and $q' \notin S$, with a pair of words $\alpha \in \mathsf{SP}(p \rightsquigarrow q, S)$ (simple word to a state in S) and $\beta \in \mathsf{SP}(p \rightsquigarrow q', S)$ (simple word to a state not in S) such that α is a suffix of β.

We observe that the set $S = \{0, 2, 4\}$ is not well-formed since $b \in \mathsf{SP}(0 \rightsquigarrow 4, S), ab \in \mathsf{SP}(0 \rightsquigarrow 3, S)$ and b is a suffix of ab. Whereas $S' = \{0, 2, 3, 4\}$ is both suffix-tracking, and well-formed, and induces an equivalent DSA. On the word aba, the run on the DSA would

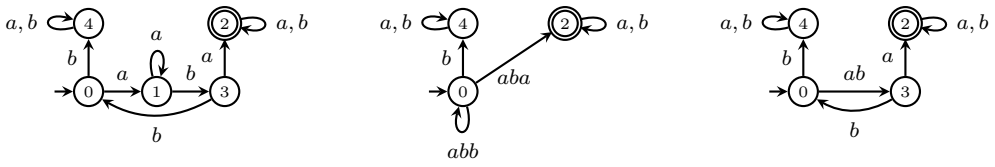


FIGURE 4.5: A DFA, a non-equivalent DSA and an equivalent induced DSA.

be $0 \xrightarrow{ab} 3 \xrightarrow{a} 2$. The first move $0 \xrightarrow{ab} 3$ applies the longest match criterion, and fires the ab transition since ab is a longer suffix than b. This was not possible before since $3 \notin S$. It turns out that the two conditions — suffix-compatibility and well-formedness — are sufficient to induce a language equivalent DSA.

Definition 4.9 (Suffix-tracking sets)**.** A set of states $S \subseteq Q$ is suffix-tracking if it contains the initial and accepting states, and

1. every transition of M is suffix-compatible w.r.t. S,
2. and S is well-formed.

All these notions lead to the main theorem of this section.

Theorem 4.10. *Let S be a suffix-tracking set of complete DFA M, and let $\mathcal{A}_S$ be the DSA induced using S. Then: $\mathcal{L}(\mathcal{A}_S) = \mathcal{L}(M)$*

Proof. Pick $w \in \mathcal{L}(M)$. There is an accepting run $q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} \dots \xrightarrow{w_n} q_n$ of M on w. By Definition 4.9, we have $q_0, q_n \in S$. Let $1 \leq i \leq n$ be the smallest index greater than 0, such that $q_i \in S$. Consider the run segment $q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} \dots \xrightarrow{w_i} q_i$. By Lemma 4.6, and by the definition of induced DSA 4.4, no transition of $\mathcal{A}_S$ out of q_0 is triggered until $w_1 \dots w_{i-1}$, and then on reading w_i, the transition $q_0 \xrightarrow{\alpha} q_i$ is triggered, where $\alpha \in \mathsf{SP}(p \rightsquigarrow q, S)$, and α is also the longest suffix of $w_1 \dots w_i$ among $\mathsf{SP}(p, S)$. In particular, it is the longest suffix among outgoing labels from q_0 in $\mathcal{A}_S$. This shows there is a move $q_0 \xrightarrow[\alpha]{w_1 \dots w_i} q_i$ in $\mathcal{A}_S$. Repeat this argument on rest of the run $q_i \xrightarrow{w_{i+1}} q_{i+1} \xrightarrow{w_{i+1}} \dots \xrightarrow{w_n} q_n$ to extend the run of $\mathcal{A}_S$ on the rest of the word. This shows $w \in \mathcal{L}(\mathcal{A}_S)$.

Pick $w \in \mathcal{L}(\mathcal{A}_S)$. There is an accepting run ρ of $\mathcal{A}_S$ starting at the initial state q_0. Consider the first move $q_0 \xrightarrow[\alpha]{w_1 \dots w_i} q_i$ of $\mathcal{A}_S$ on the word. By the semantics of a move (Definition 3.2)

and Lemma 4.7, we obtain a run $q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} \dots q_{i-1} \xrightarrow{w_i} q_i$ of M where the intermediate states $q_1, \dots, q_{i-1}$ lie in $Q \setminus S$. We apply this argument for each move ρ in the accepting run of $\mathcal{A}_S$ to get an accepting run of M. □

We extend the idea of well-formed set of states to a corresponding notion on DSAs. We will employ this notion when we remove some unnecessary transitions from the induced DSAs that are obtained.

Definition 4.11 (Well-formed DSA)**.** A DSA $\mathcal{A}$ is *well-formed* if for every state q, no outgoing label $\alpha \in \mathrm{Out}(q)$ is a suffix of some proper prefix β' of another outgoing label $\beta \in \mathrm{Out}(q)$.

4.2 Removing some redundant transitions.

Let us now get back to Figure 3.3 to see if we can derive the DSA on the left from the DFA on the right (assuming q is the initial state). As seen earlier, the set $S = \{q, q'\}$ is suffix tracking. It is also well formed since $baaa$ is not a suffix of any prefix of $abaa$ and vice-versa. The DSA $\mathcal{A}_S$ induced using q and q' will have the set of words in $\mathsf{SP}(q \rightsquigarrow q', S)$ as transitions between q and q'. Both $abaa$ and $baaa$ belong to $\mathsf{SP}(q \rightsquigarrow q', S)$. However, there are some additional simple words: for instance, $abbaaa$. Notice that $baaa$ is a suffix of $abbaaa$, and therefore even if we remove the transition on $abbaaa$, there will be a move to q' via $q \xrightarrow{baaa} q'$. This tempts us to use only the suffix-minimal words in the transitions of the induced DSA. This is not always safe, as we explain below. We show how to carefully remove "bigger-suffix-transitions".

Consider the DSA on the left in Figure 4.6. If $caba$ is removed, the moves which were using $caba$ can now be replaced by ba and we still have the same pair of source and target states. Consider the picture on the right of the same figure. There is an outgoing edge to a different state on aba. Suppose we remove $caba$. The word $caba$ would then be matched by the longer suffix aba and move to a different state. Another kind of redundant transitions are some of the self-loops on DSAs. In Figure 4.3, the self-loop on b at q_0 can be removed, without changing the language. This can be generalized to loops over longer words, under some conditions.

Definition 4.12. Let $\mathcal{A}$ be a DSA, q, q' be states of $\mathcal{A}$ and $t := q \xrightarrow{\alpha} q'$ be a transition.

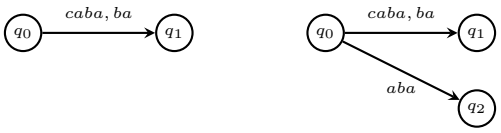


FIGURE 4.6: Illustrating bigger-suffix transitions and when they are redundant

We call t a *bigger-suffix-transition* if there exists another transition (q, β, q') with β a suffix of α.

If there is a transition $t' := q \xrightarrow{\gamma} q''$ $(q'' \neq q')$, such that β is a suffix of γ, and γ is a suffix of α, we call t *useful*. A bigger-suffix-transition is called *redundant* if it is not useful.

We will say that t is a *redundant self-loop* if $q = q'$, q is not an accepting state, and no suffix of α is a prefix of some outgoing label in $\mathrm{Out}(q)$.

In Figure 4.6, for the automaton on the left, the transition on $caba$ is redundant. Whereas for the DSA on the right, $caba$ is a bigger-suffix-transition, but it is useful. The self-loop on q_0 in Figure 4.3 is redundant, but the loop on q_0 in Figure 3.2 is useful. Lemmas 4.13 and 4.14 prove correctness of removing redundant transitions.

Lemma 4.13. *Let $\mathcal{A}$ be a DSA, and let $t := q \xrightarrow{\alpha} q'$ be a redundant bigger-suffix-transition. Let $\mathcal{A}'$ be the DSA obtained by removing t from $\mathcal{A}$. Then, $L(\mathcal{A}) = L(\mathcal{A}')$.*

Proof. To show $L(\mathcal{A}) \subseteq L(\mathcal{A}')$. Let $w \in L(\mathcal{A})$ and let $q_0 \xrightarrow[\alpha_0]{w_0} q_1 \xrightarrow[\alpha_1]{w_1} \cdots \xrightarrow[\alpha_{m-1}]{w_{m-1}} q_m$ be an accepting run. If no (q_i, α_i, q_{i+1}) equals (q, α, q'), then the same run is present in S', and hence $w \in L(S')$. Suppose $(q_j, \alpha_j, q_{j+1}) = (q, \alpha, q')$ for some j. So, the word w_j ends with α. As (q, α, q') is a bigger-suffix-transition, there is another (q, β, q') such that $\beta \sqsubseteq_{\mathsf{sf}} \alpha$. Therefore, the word w_j also ends with β. Since there was no transition matching a proper prefix of w_j, the same will be true at $\mathcal{A}'$ as well, since it has fewer transitions. It remains to show that $q_j \xrightarrow[\beta]{w_j} q_{j+1}$ is a move. The only way this cannot happen is if there is a $q \xrightarrow{\gamma} q''$ with $\beta \sqsubseteq_{\mathsf{sf}} \gamma \sqsubseteq_{\mathsf{sf}} \alpha$. But this is not possible since $q \xrightarrow{\alpha} q'$ is a redundant bigger-suffix transition. Therefore, every move using (q, α, q') in $\mathcal{A}$ will now be replaced by (q, β, q') in $\mathcal{A}'$. Hence we get an accepting run in $\mathcal{A}'$, implying $w \in L(\mathcal{A}')$.

To show $L(\mathcal{A}') \subseteq L(\mathcal{A})$. Consider $w \in L(\mathcal{A}')$ and an accepting run $q_0 \xrightarrow[\alpha_0]{w_0} q_1 \xrightarrow[\alpha_1]{w_1} \cdots \xrightarrow[\alpha_{m-1}]{w_{m-1}} w_m$ in $\mathcal{A}'$. Notice that if $q \xrightarrow[\beta]{w_j} q'$ is a move in $\mathcal{A}'$, the same is a move in $\mathcal{A}$ when $\alpha \not\sqsubseteq_{\mathsf{sf}} w_j$.

When $\alpha \sqsubseteq_{\mathsf{sf}} w_j$, then the bigger-suffix-transition $q \xrightarrow{\alpha} q'$ will match and the move $q \xrightarrow[\beta]{w_j} q'$ gets replaced by $q \xrightarrow[\alpha]{w_j} q'$. Hence we will get the same run, except that some of the moves using $q \xrightarrow{\beta} q'$ may get replaced with $q \xrightarrow{\alpha} q'$. □

For the correctness of removing redundant self-loops, we assume that the DFA that we obtain is well-formed (Definition 4.11) and has no redundant bigger-suffix-transitions. The induced DSA that we obtain from suffix-tracking sets is indeed well-formed. Starting from this induced DSA, we can first remove all redundant bigger-suffix-transitions, and then remove the redundant self-loops.

Lemma 4.14. *Let $\mathcal{A}$ be a well-formed DSA that has no removable bigger-suffix-transitions. Let $t := (q, \alpha, q)$ be a removable self-loop. Then the DSA $\mathcal{A}'$ obtained by removing t from $\mathcal{A}$ satisfies $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.*

Proof. To show $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$. Let $w \in \mathcal{L}(\mathcal{A})$ and let $\rho := q_0 \xrightarrow{w_0} q_1 \xrightarrow{w_1} \cdots \xrightarrow{w_{m-1}} q_m$ be an accepting run. Suppose t matches the segment $q_j \xrightarrow{w_j} q_{j+1}$. Hence $q_j = q_{j+1} = q$. Observe that as q is not accepting, we have $j + 1 \neq m$. Therefore there is a segment $q_{j+1} \xrightarrow{w_{j+1}} q_{j+2}$ in the run. We claim that if t is removed, then no transition out of q can match any prefix of $w_j w_{j+1}$.

First we see that no prefix of w_j can be matched, including w_j itself: if at all there is a match, it should be at w_j, and a β that is smaller than α. By assumption, α is not a removable bigger-suffix-transition. Therefore, there is a transition $q \xrightarrow{\gamma} q'$, with $\beta \sqsubseteq_{\mathsf{sf}} \gamma \sqsubseteq_{\mathsf{sf}} \alpha$. This contradicts the assumption that α is a removable self-loop. Therefore there is no match upto w_j.

Suppose some (q, β, q') matches a prefix $w_j u$ such that $\beta = vu$, that is, β overlaps both w_j and w_{j+1}. If $\alpha \sqsubseteq_{\mathsf{sf}} v$, then it violates well-formedness of S since it would be a suffix of a proper prefix (v) of β. This shows $v \sqsubseteq_{\mathsf{sf}} \alpha$ (since both are suffixes of w_j) and $v \sqsubseteq_{\mathsf{pr}} \beta$, contradicting the assumption that t is removable. Therefore, β does not overlap w_j. But then, if β is a suffix of a proper prefix of w_{j+1}, we would not have the segment $q_{j+1} \xrightarrow{w_{j+1}} q_{j+2}$ in the run ρ. Therefore, the only possibility is that we have a segment $q_j \xrightarrow{w_j w_{j+1}} q_{j+2}$. We

have fewer occurrences of the removable loop (q, α, q) in the modified run. Repeating this argument for every match of (q, α, q) gives an accepting run of $\mathcal{A}'$. Hence $w \in L(\mathcal{A}')$.

To show $L(\mathcal{A}') \subseteq L(\mathcal{A})$. Let $w \in L(\mathcal{A}')$ and $\rho' := q_0 \xrightarrow{w_0} q_1 \xrightarrow{w_1} \cdots \xrightarrow{w_{m-1}} q_m$ be an accepting run in $\mathcal{A}'$. Suppose $q_j \xrightarrow{w_j} q_{j+1}$ is matched by (q, β, q'). Let $w_j = vu$ with $\alpha \sqsubseteq_{\mathsf{sf}} v$. Then the removable-self-loop (q, α, q) will match the prefix v. Suppose β overlaps with both v and u, that is $\beta = \beta' u$. We cannot have $\alpha \sqsubseteq_{\mathsf{sf}} \beta'$ due to well-formedness of $\mathcal{A}$. We cannot have $\beta' \sqsubseteq_{\mathsf{sf}} \alpha$ since this would mean there is a suffix of α which is a prefix of β, violating the removable-self-loop condition. Therefore, β is entirely inside u, that is, $\beta \sqsubseteq_{\mathsf{sf}} u$. Hence in $\mathcal{A}$ the run will first start with $q \xrightarrow{v} q$. Applying the same argument, prefixes of the remaining word where t matches will be matched until there is a part of the word where (q, β, q') matches. This applies to every segment, thereby giving us a run in $\mathcal{A}$. □

We now get to the core definition of this section, which tells how to derive a DSA from a DFA, using the methods developed so far.

Definition 4.15 (DFA-to-DSA derivation)**.** A DSA is said to be *derived from* DFA M using $S \subseteq Q$, if it is identical to an induced DSA of M (using S) with all redundant transitions removed.

By Theorem 4.10 and Lemma 4.13, we get the following result.

Theorem 4.16. *Every DSA that is derived from a complete DFA using a suffix-tracking set, is language equivalent to it.*

Concluding remarks for Chapter 4. This chapter has provided a concrete answer to the constructive question that concluded the previous one: how to build a DSA for a given regular language. We have developed a complete and sound derivation procedure that transforms a given complete DFA into a language-equivalent DSA.

The core of this method is the concept of the *suffix-tracking set*. By identifying the key technical conditions of *suffix-compatibility*, which ensures the internal consistency of the suppressed automaton paths, and *well-formedness*, which ensures non-ambiguous connections at the path endpoints, we established a rigorous framework for abstracting away DFA

states while correctly preserving the original language's semantics within the suffix-reading paradigm. The procedure is fully constructive, providing a clear algorithm for generating correct DSAs, complete with post-processing steps to prune redundant transitions for improved conciseness.

Having established a method for constructing correct DSAs, the focus naturally shifts to optimization. Our procedure allows for many possible suffix-tracking sets within a single DFA, each yielding a different DSA of potentially different size. This raises the fundamental question of **minimization**: how can we find a DSA with the smallest possible total size for a given language? One might intuitively expect that applying our derivation procedure to the canonical (minimal-state) DFA would be the optimal strategy, mirroring the approach for models like DGAs [GM99]. The next chapter will investigate this question and reveal a surprising departure from this classical principle. We will see that the canonical DFA is not necessarily the best starting point.

Chapter 5

Minimality – some observations and challenges

The previous chapter provided a constructive answer to the question of how to build a Deterministic Suffix-reading Automaton from a DFA. The derivation procedure, based on suffix-tracking sets, can generate a language-equivalent DSA. However, the choice of suffix-tracking set is not unique, meaning our procedure can produce many different DSAs for the same language, each with a different total size. This naturally gives rise to the optimization problem: how do we find a DSA that is minimal, that is, one with the smallest possible total size?

In classical automata theory, the path to a minimal DFA is anchored by the Myhill-Nerode theorem and the existence of a unique canonical DFA for every regular language. One might reasonably expect a similar principle to apply here: that a minimal DSA could be found by applying our derivation procedure to this canonical DFA. This chapter will demonstrate that this is not the case. DSA minimization is far more complex.

We begin by establishing that unlike DFAs, minimal DSAs are not necessarily unique. We then present the central, surprising observation of this chapter: we will show via a concrete counterexample that the smallest DSA derivable from the canonical DFA of a language is not always a minimal DSA for that language. In some cases, a smaller DSA can only be derived from a larger, non-minimal DFA. This finding suggests that the standard Nerode equivalence does not capture the right notion of state equivalence for DSA minimization.

Structure of this chapter. This investigation will proceed in three parts.

- In Section 5.1, we will explore some of the properties of minimal DSAs and the surprising observation that minimal DSAs need not be derived from the canonical DFA.
- In Section 5.2, we will formalize the inherent difficulty of this problem by proving that DSA minimization is NP-complete.
- Finally, in response to this complexity, in Section 5.3 we will introduce and analyze a well-behaved subclass, 'Strongly Deterministic Suffix-reading Automata (sDSAs)', for which the desirable link to the canonical DFA is restored.

5.1 Some properties of minimal DSAs

Theorem 3.7 shows that we cannot expect DSAs to be smaller than (trim) DFAs or DGAs in general. However, Lemma 3.6 and Figure 1.1 show that there are cases where DSAs are smaller and more readable. This motivates us to ask the question of how we can find a minimal DSA, that is, a DSA of the smallest (total) size. The first observation is that minimal DSAs need not be unique — see Figure 5.1.

Lemma 5.1. *The minimal DSA need not be unique.*

Proof. Figure 5.1 illustrates two DSAs, both of size 8 (3 states, 2 edges, total label length 3), accepting the same language. From DSA $\mathcal{A}_1$, we can see that the language accepted is $b^*a^*abb^*a$. From DSA $\mathcal{A}_2$, we can derive the expression $b^*aa^*b^*ba$. It is easy to see that both the expressions are equivalent.

It remains to show that there is no DSA of smaller size for this language. Let $L = b^*a^*abb^*a = b^*aa^*b^*ba$. Let $\mathcal{A}$ be a minimal automaton for L. Any DSA for L has an initial state q_0 which is non-accepting (since $\varepsilon \notin L$, and an accepting state $q_1 \neq q_0$). So, in particular $\mathcal{A}$ has at least two states q_0, q_1. The smallest word in L is aba. Suppose the accepting run of $\mathcal{A}$ on aba is due to a transition $t := q_0 \xrightarrow{aba} q_1$. Consider the word $abba \in L$.

FIGURE 5.1: Minimal DSA is not unique

Transition t does not match $abba$. Therefore, the first transition in the accepting run of $\mathcal{A}$ on $abba$ is some transition $t' \neq t$. This transition t' will have a label of size at least 1. Hence, $\mathcal{A}$ has at least two states, and at least two transitions: t which contributes to size 4 (includes 1 edge and label of length 3) and a transition t' of size at least 2. Therefore $|\mathcal{A}| \geq 2+4+2 = 8$, which is at least the size of $|\mathcal{A}_1|$ and $|\mathcal{A}_2|$, and hence any DSA that accepts aba through a single transition has total-size at least 8. Now, suppose the accepting run of $\mathcal{A}$ on aba has a first transition on a prefix of aba, either a or ab, corresponding to an intermediate transition $q_0 \xrightarrow{a} q'$ or $q_0 \xrightarrow{ab} q'$. In the former case, there can be a transition $q' \xrightarrow{ba} q_1$ to accept aba (if the transition is on just b, it only makes the automaton larger since a remains to be read), and in the latter case, a transition $q' \xrightarrow{a} q_1$. This discussion shows that the two automata $\mathcal{A}_2$ and $\mathcal{A}_1$, are indeed minimal. □

The next observation is that minimal DSAs will be well-formed (Definition 4.11): if there are two transitions $q \xrightarrow{\alpha} q_1$ and $q \xrightarrow{\beta_1\alpha\beta_2} q_2$, then we can remove the second transition since it will never get fired.

Due to the "well-formedness" property in suffix-tracking sets, the DSAs induced by suffix-tracking sets are naturally well-formed. Furthermore, since removing redundant transitions preserves this property, the DSAs that are derived using our DFA-to-DSA procedure (Definition 4.15) are well-formed. The next proposition shows that every DSA that is well-formed and has no redundant transitions (and in particular, the minimal DSAs) can be derived from the corresponding tracking DFAs (see Definition 4.15 for what it means to be "derived").

Proposition 5.2. *Every well-formed DSA with no redundant transitions can be derived from its tracking DFA.*

Proof. We use notation as in Definition 2.1. Consider a DSA $\mathcal{A}$ that is well-formed and has no redundant bigger-suffix-transitions. Let $M_{\mathcal{A}}$ be its tracking DFA as in Definition 3.4. Let

$S = \bigcup_{q \in \mathcal{A}} (q, \varepsilon)$. Here is the schema of the proof:

$$M_{\mathcal{A}} \xrightarrow{S} \text{ induced DSA } \mathcal{A}' \xrightarrow{\text{remove redundant transitions}} \mathcal{A}$$

1. We first show that S is suffix-tracking and well-formed.
2. For each state q of $\mathcal{A}$, and for each $\alpha \in \mathsf{SP}((q, \varepsilon), S)$, either $\alpha \in \mathrm{Out}(q)$ (a transition out of q in $\mathcal{A}$) or α is a redundant bigger-suffix-transition that will be removed in the second step.

All together, this shows that $\mathcal{A}$ is derived from $M_{\mathcal{A}}$ using our procedure.

S is suffix-tracking. Pick a state (q, β) with $\beta \neq \varepsilon$. What is the set $\mathsf{SP}((q, \varepsilon) \rightsquigarrow (q, \beta), S)$? Clearly, $\beta \in \mathsf{SP}((q, \varepsilon) \rightsquigarrow (q, \beta), S)$. Let $\alpha \in \mathsf{SP}((q, \varepsilon) \rightsquigarrow (q, \beta), S)$, with $\alpha \neq \beta$. Then there is a path $(q, \varepsilon) \xrightarrow{\alpha_1} (q, \beta_1) \xrightarrow{\alpha_2} (q, \beta_2) \cdots (q, \beta_{k-1}) \xrightarrow{\alpha_k} (q, \beta_k) = (q, \beta)$. Let $j \in \{1, \ldots, k\}$ be the largest index such that $\beta_j, \beta_{j+1}, \ldots, \beta_k$ are all prefixes of β. Therefore, α starts by visiting a branch different from the prefixes of β, moves to potentially different "branches" (prefixes of other words in $\mathrm{Out}(q)$, and on $\alpha_1 \ldots \alpha_j$ hits the β branch, after which it stays in the same branch. By definition, β_j is the longest prefix among $\overline{\mathrm{Out}}(q)$ such that $\beta_j \sqsubseteq_{\mathsf{sf}} \alpha_1 \ldots \alpha_j$. We can infer the following: (1) $\beta \sqsubseteq_{\mathsf{sf}} \alpha$, and (2) among $\overline{\mathrm{Out}}(q)$, β is the longest suffix of α: otherwise, for $\alpha_1 \ldots \alpha_j$, there would be a $\beta' \neq \beta_j$ which is a longer suffix, contradicting the definition of the transition $(q, \beta_{j-1}) \xrightarrow{\alpha_j} (q, \beta_j)$.

These observations are helpful to show that S is suffix-tracking. Consider a transition $(q, \beta) \xrightarrow{a} (q, \beta')$ with $\beta' \neq \varepsilon$. We require to show that for every $\alpha \in \mathsf{SP}((q, \varepsilon) \rightsquigarrow (q, \beta), S)$, the longest suffix of αa among $\mathsf{SP}((q, \varepsilon), S)$ lies in $\mathsf{SP}((q, \varepsilon) \rightsquigarrow (q, \beta'), S)$. Pick $\alpha \in \mathsf{SP}((q, \varepsilon) \rightsquigarrow (q, \beta), S)$, and consider the extension αa. By (1) above, we have $\beta \sqsubseteq_{\mathsf{sf}} \alpha$. Since $\beta' \sqsubseteq_{\mathsf{sf}} \beta a$, we have $\beta' \sqsubseteq_{\mathsf{sf}} \alpha a$. Suppose there is an $\alpha'' \in \mathsf{SP}((q, \varepsilon) \rightsquigarrow (q, \beta''), S)$ such that $|\alpha''| > |\beta'|$ and $\alpha'' \sqsubseteq_{\mathsf{sf}} \alpha a$. By definition of the transition $(q, \beta) \xrightarrow{a} (q, \beta')$, β' is the longest suffix of βa, and hence $|\beta'| > |\beta''|$. From $|\alpha''| > |\beta'|$, $\alpha'' \sqsubseteq_{\mathsf{sf}} \alpha a$ and $\beta' \sqsubseteq_{\mathsf{sf}} \beta a$, we have $\beta' \sqsubseteq_{\mathsf{sf}} \alpha''$. By point (2) of the above paragraph, we have $|\beta''| > |\beta'|$. A contradiction. Therefore, there is no such α''. Hence β' is indeed the longest suffix of αa for all $\alpha \in \mathsf{SP}((q, \varepsilon) \rightsquigarrow (q, \beta), S)$.

Now, consider a transition $(q, \beta) \xrightarrow{a} (q, \overline{\varepsilon})$. This happens when no non-empty word in $\overline{\mathrm{Out}}(q)$ is a suffix of βa. In that case, there is a transition $(q, \varepsilon) \xrightarrow{a} (q, \overline{\varepsilon})$, and a is indeed the longest suffix of βa among $\mathsf{SP}((q, \varepsilon), S)$. This shows that every transition is suffix-compatible.

S is well-formed. S is well-formed naturally since $\mathcal{A}$ is well-formed. Suppose it was not. That means there exist states $p \in S, q' \notin S, q$, and words $\alpha \in \mathsf{SP}(p \rightsquigarrow q, S), \beta' \in \mathsf{SP}(p \rightsquigarrow q', S)$, such that $\alpha \sqsubseteq_{\mathsf{sf}} \beta'$. Then there is a state $q'' \in S$, and a word $\beta \in \mathsf{SP}(p \rightsquigarrow q'', S)$ such that $\beta' \sqsubset_{\mathsf{pr}} \beta$, and we have $\alpha \sqsubseteq_{\mathsf{sf}} \beta'$. Since $\alpha, \beta \in \mathrm{Out}(p)$, this means $\mathcal{A}$ is not well-formed, which is a contradiction.

Extra strings in the induced DSA are redundant. In the induced DSA, we will have $\mathsf{SP}((q, \varepsilon), S)$ to have more words than $\mathrm{Out}(q)$. We need to show that all the other words are redundant bigger-suffix-transitions, and hence will be removed by the derivation procedure. Consider a transition of the form $(q, \beta) \xrightarrow{a} (q', \varepsilon)$. For every word $\alpha \in \mathsf{SP}((q, \varepsilon) \rightsquigarrow (q, \beta), S)$, with $|\alpha| > |\beta|$, we have β as the longest suffix of α, among $\overline{\mathrm{Out}}(q)$. Therefore, there is no β' with $|\alpha| \geq |\beta'| > |\beta|$ with $\beta' \sqsubseteq_{\mathsf{sf}} \alpha$. This is sufficient to see that there is no $\beta' a \in \mathrm{Out}(q)$ such that $\beta a \sqsubseteq_{\mathsf{sf}} \beta' a \sqsubseteq_{\mathsf{sf}} \alpha a$. Hence αa is a redundant bigger-suffix-transition in the induced DSA.

By assumption, we have no redundant bigger-suffix-transitions in $\mathcal{A}$. Hence, in the derivation procedure, we do not remove any transition already present in $\mathcal{A}$. This shows that the finally derived DSA is exactly $\mathcal{A}$. □

Proposition 5.2 says that if we somehow had access to the tracking DFA of a minimal DSA, we will be able to derive it using our procedure. The challenge however is that this tracking DFA may not necessarily be the canonical DFA for the language. In fact, we now show that a smallest DSA that can be derived from the canonical DFA need not be a minimal DSA.

Figure 5.2 shows a DFA M^*. Observe that M^* is minimal: every pair of states has a distinguishing suffix. Let us now look at DSAs that can be derived from M^*. Firstly, any suffix-tracking set on M^* would contain q_0, q_4 (since they are initial and accepting states). If p is not picked, the transition $p \xrightarrow{a} p$ is not suffix-compatible. Therefore, p should belong to the

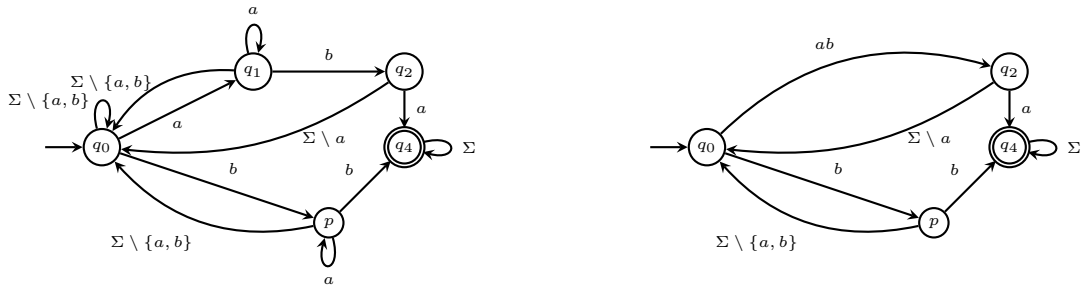


FIGURE 5.2: DFA M^* on the left and a derived DSA $\mathcal{A}^*_S$ with $S = \{q_0, q_2, q_4, p\}$ on the right.

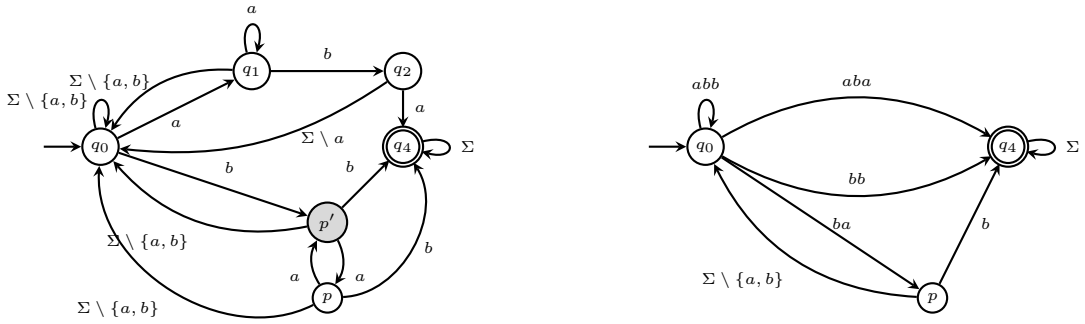


FIGURE 5.3: DFA M^{**} on the left and a derived DSA $\mathcal{A}^{**}_S$ with $S = \{q_0, q_2, q_4, p\}$ on the right.

selected set. If p is picked, and q_2 not picked, then the set is not well-formed (see Definition 4.8): the simple word b from q_0 to p is a suffix of the simple word ab to q_2. Therefore, any suffix-tracking set should contain the 4 states q_0, p, q_2, q_4. This set $S = \{q_0, p, q_2, q_4\}$ is indeed suffix-tracking, and the DSA derived using S is shown in the right of Figure 5.2. The only other suffix-tracking set is the set S' of all states. The DSA derived using S' will have state q_1 in addition, and the transitions $\Sigma \setminus \{a, b\}$. If Σ is sufficiently large, this DSA would have total size bigger than $\mathcal{A}^*_S$ (Note: Σ can be any alphabet containing a and b, so we can increase its size arbitrarily to blow up the DSA). We deduce $\mathcal{A}^*_S$ to be the smallest DSA that can be derived from M^*.

Figure 5.3 shows DFA M^{**} which is obtained from M^* by duplicating state p to create a new state p', which is equivalent to p. So M^{**} is language equivalent to M^*, but it is not minimal. Here, if we choose p in a suffix-tracking set, the simple word to p is ba, which is not a suffix of ab (the simple word to q_2). Hence, we are not required to add q_2 into the set. Notice that $S = \{q_0, p, q_4\}$ is indeed a suffix-tracking set in M^{**}. The derived DSA $\mathcal{A}^{**}_S$ is shown in the right of the figure. The "heavy" transition on $\Sigma \setminus a$ disappears. There are some extra

transitions, like $q_0 \xrightarrow{bb} q_4$, but if Σ is large enough, the size of $\mathcal{A}_S^{**}$ will be smaller than $\mathcal{A}_S^{*}$. This shows that starting from a big DFA helps deriving a smaller DSA, and in particular, the canonical DFA of a regular language may not derive a minimal DSA for the language.

5.2 Complexity of minimization

The goal of this section is to prove the following theorem.

Theorem 5.3. *Given a DFA M and positive integer k, deciding whether there exists a DSA of total size $\leq k$ language equivalent to M is NP-complete.*

If k is bigger than the size of the DFA M, then the answer is trivial. Therefore, let us assume that k is smaller than the DFA size. For the NP upper bound, we guess a DSA of total size k, compute its tracking DFA in time $\mathcal{O}(k \cdot |\Sigma|)$ and check for its language equivalence with the given DFA M. This can be done in polynomial-time by minimizing both the DFA and checking for isomorphism.

The rest of the section is devoted to proving the lower bound. We provide a reduction from the minimum vertex cover problem which is a well-known NP-complete problem [Kar72]. A vertex cover of an undirected graph $G = (V, E)$ is a subset $S \subseteq V$ of vertices, such that for every edge $e \in E$, at least one of its end points is in S. The decision problem takes a graph G and a number $k' \geq 1$ as input and asks whether there is a vertex cover of G with size at most k'. Using the graph G, we will construct a DFA M_G over an alphabet Σ_G. We then show that G has a vertex cover of size $\leq k'$ iff M_G has an equivalent DSA with total size $\leq k$ where $k = (k' + 2) \times 2\theta + (2\theta - 1)$. Here, θ is a sufficiently large polynomial in $|V|, |E|$ which we will explain later.

The alphabet Σ_G is given by $V \cup E \cup \{\$\} \cup D$ where $D = \{1, 2, \ldots, \theta\}$. States of M_G are $V \cup \{q^{init}, q^{sink}, q^{acc}\}$. For simplicity, we use the same notation for v as a vertex in G, v as a letter in Σ_G and v as a state of M_G. The actual role of v will be clear from the context. For every edge $e = (u, v)$, there are two transitions in the automaton: $u \xrightarrow{e} v$ and $v \xrightarrow{e} u$. For every $v \in V$, there are transitions $q^{init} \xrightarrow{v} v$ and $v \xrightarrow{\$} q^{acc}$. This automaton can be

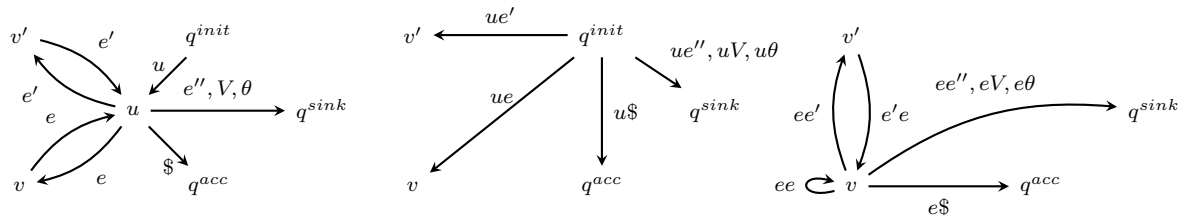


FIGURE 5.4: Left: Illustration of the neighbourhood of state u in the DFA M_G. Middle, Right: Transitions induced from q^{init} and v, on removing u.

completed by adding all missing transitions to the sink state q^{sink}. Figure 5.4 (left) illustrates the neighbourhood of a state u. The notation e'' stands for any edge that is not incident on u; there is one transition for every such e''. Initial and accepting states are respectively q^{init} and q^{acc}. Let $L_G(u)$ be the set of words that have an accepting run in M_G starting from u as the initial state. If $u \neq v$, $L_G(u) = L_G(v)$ implies (u, v) is an edge and there are no other edges outgoing either from u or v. To avoid this corner case, we restrict the vertex cover problem to connected graphs of 3 or more vertices. Then we have M_G to be a minimal DFA, with no two states equivalent. Here are two main ideas.

Suppressing a state. Suppose state u of M_G is suppressed (i.e. u is not in a suffix-tracking set). In Figure 5.4, we show the induced transitions from q^{init} and a vertex v. However, some of them will be useless transitions: most importantly, the set of transitions $q^{init} \xrightarrow{u1,u2,\ldots,u\theta} q^{sink}$ will be useless bigger-suffix-transitions due to $q^{init} \xrightarrow{1,2,\ldots,\theta} q^{sink}$. Similarly, $v \xrightarrow{e1,e2,\ldots,e\theta} q^{sink}$ will be removed. There are some more useless bigger-suffix-transitions, like $v \xrightarrow{ee''} q^{sink}$ for some e'' that is not incident on v and u. So from each v, at most $2|E|$ transitions are added. But crucially, after removing useless transitions, the θ transitions from u no longer appear. If we choose θ large enough to compensate for the other transitions, we get an overall reduction in size by suppressing states.

Two states connected by an edge cannot both be suppressed. Suppose $e = (u, v)$ is an edge. If S is a set where $u, v \notin S$, then the transition $v \xrightarrow{e} u$ is not suffix-compatible: the simple word ue from q^{init} to v, when extended with e gives the word uee; no suffix of uee is a simple word from q^{init} to u. We deduce that suffix-tracking sets in M_G correspond to a vertex cover in G, and vice-versa.

These two observations lead to a translation from minimum vertex cover to suffix-tracking

sets with least number of states. Due to our choice of θ, DSAs with smallest (total) size are indeed obtained from suffix-tracking sets with the least number of states. Let $k = (k' + 2) \times 2\theta + (2\theta - 1)$. Below, we elaborate these ideas in more detail and present the proof of the reduction.

Vertex cover $\leq k'$ implies DSA $\leq k$. Assume there is a vertex cover $\{v_1, \ldots, v_p\}$ in G with $p \leq k'$. Let S be the set of states in M_G corresponding to $\{v_1, \ldots, v_p\}$. Observe that $S \cup \{q^{init}, q^{sink}, q^{acc}\}$ is a suffix-tracking set; every transition is trivially suffix-compatible ($\forall q \xrightarrow{a} u, q \in S$ or $u \in S$). Well-formedness holds because $\forall p, q \in S, \alpha \in \mathsf{SP}(p \rightsquigarrow q, S)$ we have $|\alpha| \leq 2$; this means $\forall q' \notin S, \beta \in \mathsf{SP}(p \rightsquigarrow q', S)$, we have $\alpha \not\sqsubseteq_{\mathsf{sf}} \beta$ (since $|\beta| = 1$). Hence the derived DSA will be equivalent to M.

The derived DSA has $p + 3$ states, and transitions $q \xrightarrow{1,2,\ldots,\theta} q^{sink}$ from each except for the q^{sink} state. The transitions on q^{sink} are removable, and hence will be absent. All of this adds $(p + 2) \times 2\theta$ to the total size (edges + label lengths). Apart from these, there are transitions with labels of length at most 2, over the alphabet $V \cup E \cup \$$. From each vertex, v, there are $|V|$ transitions to q^{sink}, one transition to q^{acc} and at most $2|E|$ transitions to other states or q^{sink}. We can choose a large enough θ (say $(|V| + |E|)^4$), so that the size of these extra transitions is at most $2\theta - 1$. Hence, total size is $\leq (p + 2) \times 2\theta + (2\theta - 1)$.

By assumption, we have $p \leq k'$. Therefore, the size of the DSA is $\leq (k'+2)\times 2\theta + (2\theta - 1) = k$.

DSA $\leq k$ implies vertex cover $\leq k'$. Let $\mathcal{A}$ be a DSA with size $\leq k$. It may not be derived from M_G. However, by Proposition 5.2 we know $\mathcal{A}$ is derived from a DFA M, the tracking DFA for $\mathcal{A}$. Moreover since M_G is the minimal DFA, we know that M will be a *refinement* of M_G (see Section 2 for definition).

Let us consider a pair of states u and v from M_G, such that the vertices $u, v \in G$ have an edge between them labeled e. The DFA M will have two sets of states $u_1, u_2, \ldots, u_i$ and $v_1, v_2, \ldots, v_j$ that are language-equivalent to u and v respectively. Its initial state must have a transition on v to one of $v_1, v_2, \ldots, v_j$. Without loss of generality, let it be to v_1. Each of

$v_1, v_2, \ldots, v_j$ must have a transition on e to one of $u_1, u_2, \ldots, u_i$ (for equivalence with M_G) and vice-versa. Consider the run from the initial state on ve^{i+j+1}. At least one of the states among $u_1, u_2, \ldots, u_i, v_1, v_2, \ldots, v_j$ must be visited twice; consider the first such instance. The transition on e that re-visits a state cannot be suffix-compatible w.r.t a set S, if none of these states are in S. For it to be suffix-compatible, the string $ve^k.e$ (from initial state to the first repeated state) must have its longest simple-word suffix go the same state. Since $ve^k.e$ is not simple by itself, its longest suffix must consist entirely of e's. But on any string of e's, the initial state moves only to the sink state(s) and not to any of $u_1, u_2, \ldots, u_i, v_1, v_2, \ldots, v_j$. Hence any suffix-tracking set must contain at least one of these states, which maps to at least one of u or v in G. Every suffix-tracking set of M therefore maps to a vertex cover $\{v_1, v_2, \ldots, v_p\}$.

Now we show that the size of this vertex cover is $\leq k'$. Each of the states picked in the suffix-tracking set will contribute to atleast 2θ in the total size, due to the θ transitions. We will also have these θ transitions from the initial and accepting states. Therefore, the total size is $(p+2) \times 2\theta + y$ for some $y > 0$. Hence $(p+2) \times 2\theta \leq k$. This implies $p \leq k'$: otherwise we will have $p \geq k' + 1$, and hence $(p+2) \times 2\theta \geq (k'+1+2) \times 2\theta = (k'+2) \times 2\theta + 2\theta > k$, a contradiction.

5.3 Strongly Deterministic Suffix-reading Automata (sDSA)

As we saw in Section 5, the minimal DFA of a regular language may not derive a minimal DSA for the language (Figures 5.2 and Figure 5.3). While this is true for the general case, under a restricted definition of DSA we can show the minimal DFA to derive a minimal (restricted) DSA. The plan for this section is as follows:

Section 5.3.1 We first define a class of DSAs with a restricted syntax and call them *strongly deterministic* suffix-reading automata or *strong DSAs* (Definition 5.4) and give examples of DSAs that fall under this stronger class.

Section 5.3.2 In the second part, we prove that every minimal strong DSA can be derived from the canonical DFA using the derivation procedure of Section 4.

This section is inspired by a question that was left open in [KSVV24]: when does the smallest DSA derived from the canonical DFA correspond to a minimal DSA? Although we do not provide an answer to this question, we now understand that when we suitably restrict the DSA syntax, the derivation procedure indeed is able to generate a minimal automaton in the restricted class, starting from the canonical DFA. This provides new insights about the derivation procedure. Moreover, as we will see, many DSAs discussed in this paper are already strong. We also provide a real-life-inspired example of a strong DSA. So, overall, this section sends the message that there are specifications that can be encoded naturally as strong DSAs, and we have a method to generate minimal strong DSAs.

5.3.1 Syntax of strong DSAs

We start with a formal description of the syntax of strong DSAs. The examples that follow aim to give an explanation of this definition. In this definition, we say α' is a non-empty prefix of a word α when $\alpha' \neq \varepsilon$ and $\alpha' \sqsubseteq_{\mathsf{pr}} \alpha$.

Definition 5.4 (sDSA)**.** (Strongly Deterministic Suffix-reading Automata). A DSA $\mathcal{A}$ given by $(Q, \Sigma, q_{init}, \Delta, F)$ is said to be strongly deterministic if for every state $q \in Q$, for every $\alpha, \beta \in \mathrm{Out}(q)$, and for all non-empty prefixes $\alpha' \sqsubseteq_{\mathsf{pr}} \alpha$ and all non-empty proper prefixes $\beta' \sqsubset_{\mathsf{pr}} \beta$, we have $\alpha' \not\sqsubseteq_{\mathsf{sf}} \beta'$: no prefix of α is a factor of β.

For instance, the DSA in Figure 1.1 is not an sDSA: at state s_1 we have the outgoing labels $\mathrm{Out}(s_1) = \{\texttt{if}, \texttt{endif}\}$, and the letter `i` (which is a prefix of `if`) appears in `endif` as a suffix of `endi`. The DSAs in Figures 3.1 and 3.2 are strong DSAs: it is easy to verify this for $\mathcal{A}_2$ and $\mathcal{A}_3$; for $\mathcal{A}_4$, we provide an explanation. We have $\mathrm{Out}(q_0) = \{ab, ba\}$; let $\alpha = ab$ and $\beta = ba$. The non-empty prefixes of α are a and ab. The only non-empty proper prefix of β is b. Notice that neither a nor ab is a suffix of b. The same exercise can be repeated with $\alpha = ba$ and $\beta = ab$. We conclude that $\mathcal{A}_4$ is an sDSA. Note that any DFA is an sDSA, as the labels of each outgoing transition at a state are distinct letters (and a DFA is a valid DSA). A slightly modified version of the DSA in Figure 1.1 gives us an example of an sDSA. In Figure 5.5 we have a DSA for out-of-context `else` statements, where we allow nested `if`, but a single

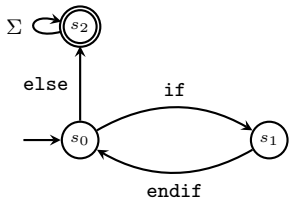


FIGURE 5.5: sDSA for out-of-context `else`, without nested if statements

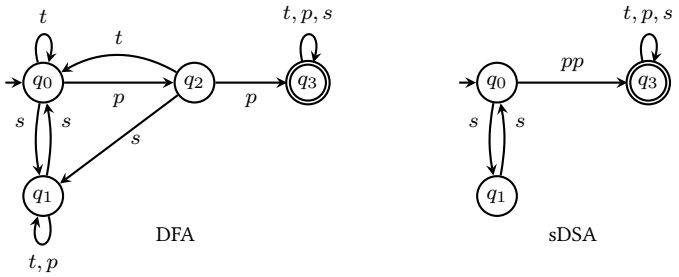


FIGURE 5.6: A simplified specification of an automotive application

`endif` appearing later corresponds to all the open `if` so far. A context would therefore be the part between the first `if` and the last `endif`. If the specification additionally requires to disallow nested `if`, then it can be modeled by the intersection of the sDSA of Figure 5.5 and another automaton that rejects words with two `if` with no `endif` in between.

A motivating example. We describe another example of a strong DSA, motivated by a real-life situation. It is a simplified specification of an automotive application: "when the alarm is off, and the panic switch is pressed twice within one clock cycle, go to an error state". The DFA in Figure 5.6 models this. States q_0 and q_1 represent the alarm being off and on respectively. State q_3 is the error state. The DFA toggles between off and on states on receiving Signal s. Signal t denotes a "tick" marking the separation of clock cycles, and p denotes the pressing of the panic switch. The sDSA for this specification is given on the right of Figure 5.6. At state q_0, the automaton keeps receiving signals until either an s or a sequence pp is seen. If s is seen first, the automaton moves to q_1, otherwise it moves to q_3. The move on $q_0 \xrightarrow{pp} q_3$ happens on words over $\{t, p, s\}$ that end with pp, and contain neither an s nor a previous occurrence of pp. This expresses that the panic switch was pressed twice, within one clock cycle (as no t occurred), while the alarm was still off. Here, the sDSA provides a smaller, and arguably, a more readable representation than the DFA.

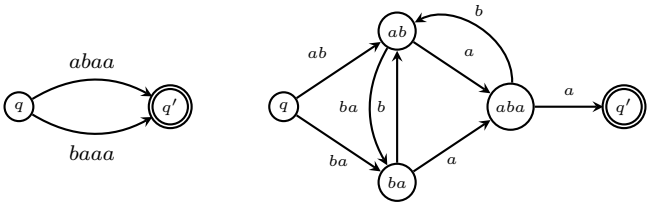


FIGURE 5.7: A DSA on the left, and the corresponding (larger) sDSA

However sDSA may not always provide as succinct a representation as DSA. We see in Figure 5.7, an example of a DSA on the left and an equivalent sDSA on the right. It is the DSA from Figure 3.3 again, with the corresponding sDSA shown. The DSA itself is not a valid sDSA, since ba is a prefix of $baaa$ and a factor of $abaa$ (also a is prefix of $abaa$ and a factor of $baaa$). Intuitively, to construct an sDSA, we need to break the transitions after ab and ba and introduce new states. This continues to happen on the next transitions from these new states, resulting in the sDSA shown.

5.3.2 Minimality for strong DSAs

We state some preliminary lemmas, leading up to the main result that any minimal sDSA is derived from a minimal DFA. The next lemma is generic and holds for all DSAs, which are not necessarily strong. We start with some notation.

For a state q of a DFA M, we define $L^M(q)$ to be the words accepted by M starting from q and call it the *residual language* of the state. Similarly, for a DSA $\mathcal{A}$ and a state q of $\mathcal{A}$, we can define the residual language $L^{\mathcal{A}}(q)$. We say that two states p, q of a DFA/DSA are *equivalent* if their residual languages are equal. In the setting of DFAs, we know that in the canonical (minimal-state) automaton, no two states are equivalent. The next lemma establishes this property even in the DSA setting.

Lemma 5.5. *No two states of a minimal DSA can be equivalent.*

Proof. Let $\mathcal{A}$ be a minimal DSA. Suppose $L^{\mathcal{A}}(p) = L^{\mathcal{A}}(q), p \neq q$. We will now construct a DSA $\mathcal{A}'$ with smaller size than $\mathcal{A}$. This will be a contradiction. To get $\mathcal{A}'$, we will re-orient all transitions going to q to now point towards p: remove each transition (r, α, q) and add the transition (r, α, p); then remove q and other unreachable states after this transformation.

Since $L^{\mathcal{A}}(p) = L^{\mathcal{A}}(q)$, this construction preserves the language. For all the states r that remain in $\mathcal{A}'$ we still have the same $\mathrm{Out}(r)$ as in $\mathcal{A}$. Finally we need to argue that $|\mathcal{A}'| < |\mathcal{A}|$. This is easy to see since q has been removed from $\mathcal{A}$, and there are no new additions to $\mathcal{A}'$. □

Now we come to properties specific to sDSAs. The key idea is to consider the tracking DFA (Definition 3.4 and Figure 3.3) and investigate equivalence between states in this tracking DFA.

Lemma 5.6. *For any minimal sDSA $\mathcal{A}$, its tracking DFA $M_{\mathcal{A}}$ cannot have a 'DSA state' equivalent to any other state, i.e. if $(p, \alpha), (q, \beta)$ are states of $M_{\mathcal{A}}$ and also equivalent, then we have both $\alpha \neq \varepsilon$ and $\beta \neq \varepsilon$.*

Proof. From Lemma 5.5, we know that no two states of any minimal DSA can be equivalent. So it is not possible to have any distinct $(p, \varepsilon), (q, \varepsilon) \in M_{\mathcal{A}}$ to be equivalent. Suppose $L^{M_{\mathcal{A}}}(p, \varepsilon) = L^{M_{\mathcal{A}}}(q, \beta)$ for some $\beta \neq \varepsilon$ (and q could be p as well). We will construct a smaller sDSA $\mathcal{A}'$.

Consider state q of $\mathcal{A}$. Due to the presence of state (q, β) in $M_{\mathcal{A}}$, there exists some outgoing label of the form $\beta\alpha$ from q in $\mathcal{A}$: that is, $\beta\alpha \in \mathrm{Out}(q)$ for some α with $|\alpha| \geq 1$. Let the corresponding transition be $q \xrightarrow{\beta\alpha} r$. To get $\mathcal{A}'$: remove this transition $q \xrightarrow{\beta\alpha} r$ and add the transition $q \xrightarrow{\beta} p$. Clearly $\mathcal{A}'$ is smaller than $\mathcal{A}$, since total size is reduced by $|\alpha|$. Language accepted is the same. $\mathcal{A}'$ is also an sDSA since the only change is replacing $\beta\alpha$ with β in $\mathrm{Out}(q)$: any string in $\overline{\mathrm{Out}}(q)$ that is a factor of β would also be a factor of $\beta\alpha$, and any prefix of β is also a prefix of $\beta\alpha$. This contradicts the assumption that $\mathcal{A}$ was a minimal sDSA, thus proving the lemma. □

We remark that the above proof does not work for general DSAs. Consider the transition $q \xrightarrow{\beta\alpha} r$ that was removed, and modified to $q \xrightarrow{\beta} p$. There could be another transition $q \xrightarrow{\beta\alpha'} r'$ in the DSA — this violates the strong DSA property since $\beta\alpha$ and $\beta\alpha'$ are outgoing labels, and the prefix β of the former appears as a suffix of the latter. On seeing the word $\beta\alpha'$ from q, the original DSA moves to r'. However, in the new DSA, as soon as α is seen the state changes to p. This can modify the language. Such a situation does not happen in an sDSA due to the restriction on the outgoing labels. We now prove the main result of this section.

Theorem 5.7. *Every minimal sDSA for a language L can be derived from the canonical DFA for L.*

Proof. Let $\mathcal{A} = (Q^{\mathcal{A}}, \Sigma, \Delta^{\mathcal{A}}, F^{\mathcal{A}})$ be an arbitrary minimal sDSA. Any minimal sDSA is well-formed and has no redundant transitions. Hence $\mathcal{A}$ can be derived from its tracking DFA $M_{\mathcal{A}}$, thanks to Proposition 5.2. If the tracking DFA $M_{\mathcal{A}}$ is canonical, we are done. Otherwise, there are distinct states of $M_{\mathcal{A}}$ that are equivalent. From Lemma 5.6, we know that a state (q, ε) of $M_{\mathcal{A}}$ cannot be equivalent to any other state. Therefore, two equivalent states are of the form (p, α) and (q, β), with α and β both non-empty. We will merge such equivalent states together to build the minimal automaton and make use of this specific structure to show that $\mathcal{A}$ can be derived from the minimized DFA. For the proof, we will show the following steps.

1. Suppose a state (q, α) is equivalent to (q, β) (with the same q), then $\alpha \not\sqsubseteq_{\mathsf{pr}} \beta$ (i.e. the two states cannot be 'tracking' the same $\mathcal{A}$-transition out of q).

2. Build a DFA $M'_{\mathcal{A}}$ by quotienting equivalent states of $M_{\mathcal{A}}$, with $[q]$ representing the equivalence class of q. The resulting automaton $M'_{\mathcal{A}}$ is the canonical DFA. We use (1) to show that every simple path from a state (q, ε) to (q, β) is preserved in the quotiented automaton $M'_{\mathcal{A}}$, from $[(q, \varepsilon)]$ to $[(q, \beta)]$.

3. From (2), we show that the set of states $S' := \{[(q, \varepsilon)] \mid q \in Q^{\mathcal{A}}\}$ forms a suffix-tracking set. The DSA derived using S' turns out to be $\mathcal{A}$, proving that $\mathcal{A}$ can be derived from the canonical DFA.

Step 1. If two states $(q, \alpha), (q, \beta) \in M_{\mathcal{A}}$ are equivalent, then $\alpha \not\sqsubseteq_{\mathsf{pr}} \beta$ (i.e. the two states are not 'tracking' the same $\mathcal{A}$-transition). In other words, two states in the tracking DFA that lie on the same path corresponding to a given DSA transition, cannot be equivalent. Suppose $\alpha \sqsubseteq_{\mathsf{pr}} \beta$. We know that $\beta \in \overline{\mathrm{Out}}(q)$, so there is a string γ such that $\beta\gamma \in \mathrm{Out}(q)$ i.e. reading γ from (q, β) leads to a 'DSA state', say (q', ε). Let s be the state reached on reading γ from (q, α). Since (q, α) and (q, β) are equivalent, the state s will be equivalent to (q', ε). From Lemma 5.6, this means $s = (q', \varepsilon)$. This gives transition labels $\alpha\gamma$ (induced by the simple path just discussed) and $\beta\gamma$ from q in $\mathcal{A}$, with $\alpha \sqsubseteq_{\mathsf{pr}} \beta$, contradicting the fact that $\mathcal{A}$ was an sDSA. Hence we must have $\alpha \not\sqsubseteq_{\mathsf{pr}} \beta$.

Step 2. For two states s, s' of $M_{\mathcal{A}}$, define $s \equiv_{M_{\mathcal{A}}} s'$ if $L^{M_{\mathcal{A}}}(s) = L^{M_{\mathcal{A}}}(s')$. We denote the equivalence class of s by $[s]$. From Lemma 5.6, $[(q, \varepsilon)] = \{(q, \varepsilon)\}$. Consider a quotient DFA $M'_{\mathcal{A}}$ based on this equivalence. States are the equivalence classes of $\equiv_{M_{\mathcal{A}}}$, $\{[s] \mid s \in M_{\mathcal{A}}\}$. The initial state is $[(q^{\mathcal{A}}_{in}, \varepsilon)]$ and final states are $\{[(q, \varepsilon)] \mid q \in F^{\mathcal{A}}\}$. Transitions are given by $\{[s] \xrightarrow{a} [s'] \mid s \xrightarrow{a} s' \in M_{\mathcal{A}}\}$. This is deterministic because whenever we have $[s_1] = [s'_1]$ and $s_1 \xrightarrow{a} s_2, s'_1 \xrightarrow{a} s'_2 \in M_{\mathcal{A}}$, we also have $[s_2] = [s'_2]$. Note that $M'_{\mathcal{A}}$ is minimal, by definition.

Consider a transition $q \xrightarrow{\alpha} q'$ of the sDSA $\mathcal{A}$, with $\alpha = a_1 a_2 \dots a_n$. This transitions gives states (q, ε), (q, a_1), $(q, a_1 a_2)$, $\dots$, $(q, a_1 \dots a_{n-1})$, (q', ε) in the tracking DFA $M_{\mathcal{A}}$. From (1), we have $[(q, a_1)], [(q, a_1 a_2)], \dots, [(q, a_1 a_2 \dots a_{n-1})]$ to be distinct states in $M'_{\mathcal{A}}$. equivalent. Hence $a_1 a_2 \dots a_i$ is a simple path from $[(q, \varepsilon)]$ to $[(q, a_1 \dots a_i)]$ that does not visit any state of the form $[(p, \varepsilon)]$ in between.

Step 3. Let $S' := \bigcup_{q \in \mathcal{A}} \{[(q, \varepsilon)]\}$. We will show that S' is suffix-tracking. Consider a simple path σ from $[(q, \varepsilon)]$ to $[(q, \beta)]$ (note that σ is also a simple path from (q, ε) to (q, β) in $M_{\mathcal{A}}$). In the tracking DFA $M_{\mathcal{A}}$, every transition moves to a state tracking the longest possible suffix: $(q, \beta) \xrightarrow{a} (q, \beta')$ in $M_{\mathcal{A}}$ means β' is the longest word in $\overline{\mathrm{Out}}(q)$ s.t $\beta' \sqsubseteq_{\mathsf{sf}} \sigma a$. From (2), β' is a simple path from $[(q, \varepsilon)]$ to $[(q, \beta')]$, in $M'_{\mathcal{A}}$. Moreover, we have $[(q, \beta)] \xrightarrow{a} [(q, \beta')]$ by definition.

We claim that the longest suffix of σa among simple paths from $[(q, \varepsilon)]$, is β', which goes from $[(q, \varepsilon)]$ to $[(q, \beta')]$. This would show suffix-compatibility of the transition. Suppose instead, the longest suffix (say σ') went from $[(q, \varepsilon)]$ to $[(q, \alpha')]$ in $M'_{\mathcal{A}}$; we have σ' to also be a simple path from (q, ε) to (q, α') in $M_{\mathcal{A}}$, and α' to be longer than β'. This is a contradiction. Hence β' is the longest simple-path suffix of σa in $M'_{\mathcal{A}}$, making $[(q, \beta)] \xrightarrow{a} [(q, \beta')]$ suffix-compatible.

The set S' is also well-formed since $\mathcal{A}$ is well-formed. Suppose it was not. That means there exist states $p \in S', q' \notin S'$, and words $\alpha \in \mathsf{SP}(p \rightsquigarrow q, S'), \beta' \in \mathsf{SP}(p \rightsquigarrow q', S')$, such that $\alpha \sqsubseteq_{\mathsf{sf}} \beta'$. Then there is a state $q'' \in S'$, and a word $\beta \in \mathsf{SP}(p \rightsquigarrow q'', S')$ such that $\beta' \sqsubset_{\mathsf{pr}} \beta$, and we have $\alpha \sqsubseteq_{\mathsf{sf}} \beta'$. Since $\alpha, \beta \in \mathrm{Out}(p)$ are paths corresponding to $\mathcal{A}$-transitions, this means $\mathcal{A}$ is not an sDSA, which is a contradiction. Hence we have S' to be suffix-tracking. In

the induced DSA $\mathcal{A}_{S'}$, every simple path from $[(q, \varepsilon)]$ to $[(q', \varepsilon)]$ will be present. Hence every transition $q \xrightarrow{\alpha} q'$ in $\mathcal{A}$ is present in $\mathcal{A}_{S'}$. Any transition out of $[(q, \varepsilon)]$ which is not in $\mathrm{Out}(q)$ in $\mathcal{A}$, will be a redundant bigger-suffix-transition, analogous to the argument in last part of the proof of Proposition 5.2. Thus $\mathcal{A}$ can be derived from $M'_{\mathcal{A}}$, which is a minimal DFA. □

Concluding remarks for Chapter 5. This chapter considered the problem of DSA minimization, revealing a set of properties that differ significantly from classical automata theory. Our investigation yielded three primary contributions that deepen our understanding of the suffix-reading model.

First, we uncovered a fundamental bottleneck in DSA minimization: the discovery that the canonical (Myhill-Nerode) DFA is an insufficient starting point for deriving a minimal DSA. This surprising observation demonstrates that the optimal structure for a suffix-reading model may require finer state distinctions than those preserved by the canonical DFA. Second, we provided a formal explanation for this difficulty by proving that the DSA minimization problem is intrinsically hard: deciding if a language has an equivalent DSA of a given size is NP-complete, a result established via a reduction from the Vertex Cover problem [Kar72].

As a constructive response to this hardness, our third contribution was the introduction of the *Strongly Deterministic Suffix-reading Automaton* (sDSA). By imposing a reasonable syntactic constraint to prevent certain pattern ambiguities, we defined a tractable subclass to strike a balance between expressive power and algorithmic manageability. Crucially, we proved that for this class, minimal automata are indeed derivable from the canonical DFA, thus restoring a desirable and powerful property for a significant subset of DSAs.

With the definition, construction, and minimization of the sequential DSA model now thoroughly analyzed, we turn our attention to a new frontier. Modern software systems are rarely monolithic and sequential; they are typically composed of multiple, interacting components. The next chapter will extend the suffix-reading paradigm from this single-stream context to the domain of concurrent systems, addressing the new set of challenges, such as the interleaving explosion, that arise when modeling parallel processes.

Chapter 6

Multi-port Deterministic Suffix-Reading Automata

The preceding chapters developed the theory of Deterministic Suffix-reading Automata as a powerful tool for the succinct specification of *sequential*, pattern-based systems. However, modern software and hardware systems are rarely monolithic; they are typically composed of multiple interacting components that operate concurrently. This chapter pivots our investigation from the sequential to the concurrent domain, addressing the new set of challenges that arise when specifying the behavior of such systems.

A primary obstacle in modeling concurrent systems with sequential automata is the 'interleaving explosion'. To capture all possible behaviors, a model must typically account for every possible ordering of asynchronous events from different components. For example, a requirement that a car can only start after receiving signals for the key in ignition (K), brake pressed (B), and transmission in park (P)—in any order—would force a standard DSA to explicitly enumerate all $3! = 6$ permutations of these events on transition labels. As the number of concurrent components grows, this approach becomes combinatorially infeasible and defeats the purpose of DSA as a concise specification.

Let us quickly recall the basic principles with which we started to define the DSA model, as this will lead us to the new definition in the concurrent setting. In Figure 6.1(a), the DSA accepts all words that end with ab, for instance bab, $bbab$, $babaab$, etc. The transition $q_0 \xrightarrow{ab} q_0$

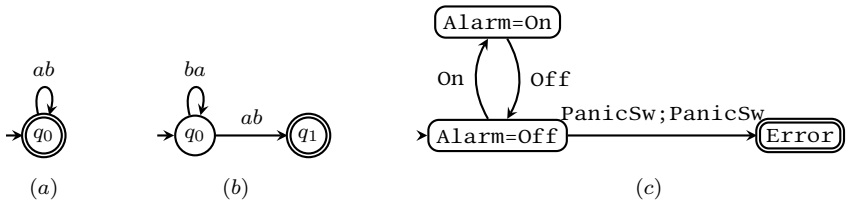


FIGURE 6.1: Examples of Deterministic Suffix-reading Automata (DSA)

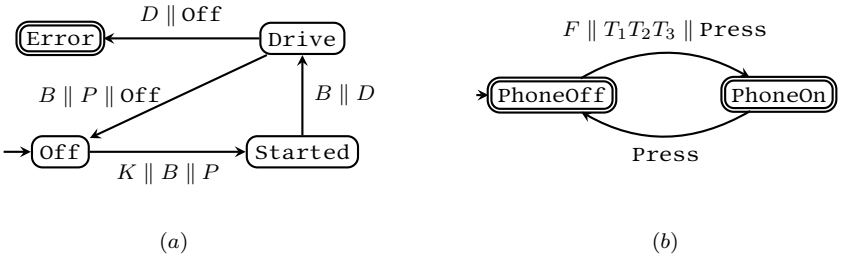


FIGURE 6.2: Concurrent suffix-based specifications as multi-port DSAs

is fired as soon as a word ending with ab is seen after reaching q_0. For instance, on the word $babaab$, the transition is first triggered on the prefix bab since this is the first time ab appears as suffix; once the transition is triggered, the prefix bab has been consumed, and now we are left with aab; on reading aab fully, the transition $q_0 \xrightarrow{ab} q_0$ is triggered again. In Figure 6.1(b), we can view the state q_0 as tracking two suffixes ba and ab. When ba is seen before ab, the self-loop is triggered; otherwise when ab is seen first, the transition to q_1 is taken. For example, on word $bbab$, the automaton starts at q_0, and on bba it fires $q_0 \xrightarrow{ba} q_0$ (since ba is a suffix of bba), then reads b and remains there. However, on $bbaab$, the run loops on bba back to q_0, and then reads the next ab to reach q_1, and accept. The final example Figure 6.1(c) is a *suffix-based specification* of a part of an automotive software: when the alarm is off, and the panic switch is pressed twice within one second, flag an error. Here, we assume there is a symbol `tick` to denote the elapse of 1 unit of time. So, on a word `tick`; `On`; `tick`; `Off`; `PanicSw`; `PanicSw`, an error should be flagged (we have added a separator symbol ; just for clarity). This specification is succinctly represented by the DSA in Figure 6.1(c).

Since transitions in a DSA wait for specific patterns to appear, DSAs are able to represent suffix-based specifications succinctly. When we consider systems with multiple components (which we call ports), the requirements can talk about different combinations of inputs. Here are two examples.

Car Security System. For a car to start, you need: key in ignition (K), brake pedal pressed (B), and transmission in park (P). The actions K, P and B can occur in any order. A DSA would need to represent this requirement using all 6 permutations. Instead, we assume an alphabet that is distributed across different *ports* – ignition, brake pedal, transmission, and denote the concurrent actions as $K \parallel B \parallel P$. Here are some requirements of a car security system.

- When the brake pedal is pressed (B), transmission is in park (P) and key is in ignition (K), the car can be started.
- If the brake pedal is pressed (B) and the transmission is moved to drive (D), the car goes to the drive mode.
- When the brake pedal is pressed (B), transmission is in park (P), and the ignition becomes off (`Off`), the car is switched off. If ignition is switched off when brake is not pressed or when transmission is in drive/reverse, signal an error.

Figure 6.2(a) illustrates the enriched DSA capturing these requirements.

Smartphone Lock Pattern. Figure 6.2(b) represents the enriched DSA modeling the requirements below for a smartphone lock.

- When the button is pressed (`Press`), fingerprint verified (F) and a correct pattern is received $(T_1T_2T_3)$, then the phone is switched on.
- If the phone is on, and the button is pressed, the phone is switched off.

To the best of our knowledge, no automaton model can succinctly represent the combination of suffix-based patterns across a concurrent system. Incorporating concurrency into the specification is paramount when requirements talk about different components of a concurrent system.

This challenge motivates the extension of our suffix-reading paradigm. The goal is to create a formalism that can directly express the logical intent of concurrent rules, such as '$K \parallel$

$B \parallel P$', thereby abstracting away the details of interleaving. This requires a new model, which we call the 'Multi-port Deterministic Suffix-reading Automaton (mDSA)'. The core idea is to partition the alphabet across multiple "ports" and introduce a parallel composition operator, '$\parallel$', into the transition syntax.

However, simply adding a new operator is a trivial syntactic change; the technical challenge lies in defining a formal operational semantics that is both theoretically sound and intuitively correct for real-world scenarios. This chapter discusses a basic semantic dilemma: should the history of events be "consumed" after a transition, or should it persist as context? The *Car Security System* example requires persistent context (the brake may need to remain pressed for a future action), while the *Smartphone Lock* example requires detecting a "fresh" button press to avoid infinite loops on old data.

To resolve this, we will show that a novel, tape-based semantics is required. This work is further motivated by practical needs in Model Based Testing (MBT) [DNSVT07, UPL12]. Industrial specification notations like Expressive Decision Tables (EDTs) [VSKA14] are used for test generation [VSZA15, AVS+20] and already employ a form of concurrent, suffix-based logic, but have historically lacked a rigorous formal semantics. A key contribution of this work is to provide such a semantics via the mDSA framework. This chapter will therefore define the mDSA model and its semantics, while subsequent chapters will extend it with outputs to fully capture and analyze notations like EDTs.

Related work. Use of automata to model concurrent systems has been explored widely. Esterel [BG92] is a synchronous programming language that provides a parallel composition operator. Harel's Statecharts [Har87] also support parallel composition of state transition systems. In both these languages, the reaction of a system in response to inputs on each port can be specified separately, and the different specifications can be composed using the parallel operator. A system model can then refer to the states of each port's specification to model the system behaviour. In mDSAs, we do not compose automata. Instead there is a single automaton, but the transitions include a parallel composition operator. More importantly, mDSA allows for suffix-based event triggers.

Petri nets [Pet66] have been studied extensively to model concurrent systems. In Petri nets, each letter of each port's alphabet will have to be represented using a place, and the places combined to model the system. There is no mechanism to include suffix-based triggers. The work that is closest to ours is Input/Output Partial Order Automata (IOPOA) [HJJ07]. In their work, transitions execute non-atomically reacting to asynchronous inputs on several ports. The main differences between IOPOA and mDSA: in mDSA transitions react atomically, transition labels can be strings representing a suffix for each port and not just letters, and different transitions may trigger on the same string at a port. For example, in the mDSA in Figure 6.2(a), the transitions from `Off` to `Started` and `Started` to `Drive` can be taken on the same input B. In an IOPOA this will require two occurences of B. IOPOA was primarily designed to enable test generation, whereas the aim of mDSAs is succinct specification of requirements.

Structure of this chapter We start with some preliminary notations. In Section 6.1, we describe in more detail the challenges in coming up with a semantics in the presence of the $\|$ operators. Section 6.2 gives the formal syntax and semantics.

Notations. Consider a special kind of alphabet $\Sigma = \langle \Sigma_1, \Sigma_2, \ldots, \Sigma_k \rangle$ such that $\Sigma_i \cap \Sigma_j = \emptyset$ for all $i, j \in \{1, \ldots, k\}$. We will call Σ_i as the alphabet of *port* i, and Σ as a *multi-port* alphabet. For instance, in the *Car Security System specification*, there are three ports: brake, transmission and ignition, with $\Sigma_{\mathsf{brake}} = \{B\}$, $\Sigma_{\mathtt{trm}} = \{P, D\}$ and $\Sigma_{\mathtt{ig}} = \{K, \mathtt{Off}\}$ respectively.

For a word $w \in \Sigma^*$, we write $\mathsf{P}_i(w)$ for the projection of w onto the set Σ_i. For instance, if $\Sigma_1 = \{a_1, a_2\}, \Sigma_2 = \{b_1, b_2\}$ and $w = a_1 b_1 a_2 a_1 b_2 b_2$, we have $\mathsf{P}_1(w) = a_1 a_2 a_1$ and $\mathsf{P}_2(w) = b_1 b_2 b_2$. Notice that if for two words w_1, w_2 suppose $\mathsf{P}_i(w_1) = \mathsf{P}_i(w_2)$ for all ports i, then w_1 and w_2 have the same order of events within each port, but could have a different ordering between letters from different ports.

6.1 Some challenges in extending to multiport

For a DSA, a configuration was given by (q, w) where q is the current state and w is the word seen after reaching q. Moreover, if a transition matches, the DSA moves to a configuration (q', ε) by fully consuming the word seen so far. Now consider the car security system example (Figure 6.2 (a)) . On receiving inputs $P; B; K$, the automaton moves to `Started` state. If the brake remains pressed and the transmission is moved to park, we would like the car to go into the drive mode. Therefore, on the word $P; B; K; D$, the automaton should go to the `Drive` state. This intention is summarized below as what we require as the run of the automaton:

$$\texttt{Off} \xrightarrow{P;B;K} \texttt{Started} \xrightarrow{D} \texttt{Drive}$$

If we had consumed the prefix $P; B; K$ while moving to the `Started` state, then the transition $B \parallel D$ would not match anymore, since there is no B after reaching `Started`. What we really need is that: the last seen input in the brake port is B and the last seen input in the transmission port is D. Therefore, one option is to simply not consume any letters and say that a transition $u_1 \parallel u_2 \parallel \cdots \parallel u_k$ matches a word w if u_i is a suffix of projection $\mathsf{P}_i(w)$. However, this creates other problems as we will now illustrate.

Consider the smartphone lock pattern specification (Figure 6.2 (b)). On a word:

$$F; T_1; T_2; T_3; \texttt{Press}$$

the phone is unlocked and the automaton goes to the `PhoneOn` state. At this state, there is an outgoing transition that listens to `Press`. If we do not consume the word seen so far, the last seen input in the button port is still `Press`, and hence the automaton would go back to the `PhoneOn` state without seeing additional inputs. In fact, this looping behaviour would continue forever, with no inputs seen. Therefore, for the outgoing transition `Press` in the `PhoneOn` state, the intention is that we need to receive a fresh signal `Press`.

To summarize: on a transition match, we do not want to consume the word seen so far

(car security specification example); but simply not consuming the word leads to problems (smartphone lock pattern example). In the following, we adopt an intermediate approach, where we mark the position θ in the word where the last transition matched. For a new transition $u_1 \parallel u_2 \parallel \cdots \parallel u_k$ to match, in addition to having each u_i as a suffix of $\mathsf{P}_i(w)$, we require at least one u_i to appear entirely after the marked position θ. We include an additional condition that every u_i appears either entirely to the left of θ or to the right of θ, to account for such examples:

$$s_0 \xrightarrow{c \parallel a} s_1 \xrightarrow{cc \parallel b} s_2$$

Action c can denote a `click`. So, the pattern cc denotes a double click. On seeing $a; c$ the automaton moves to s_1. Now if the inputs received are $b; c$, we do not want the transition $cc \parallel b$ to match. Instead, we want two clicks and a b after the last match. The condition that every pattern appears entirely after this match, or entirely before this match takes care of this situation.

6.2 Multi-port DSAs: formal syntax and semantics

We first describe the formal syntax and then proceed to an operational semantics given by a labeled transition system.

Definition 6.1 (Multi-port DSA)**.** Let $\Sigma = \langle \Sigma_1, \Sigma_2, \ldots, \Sigma_k \rangle$ be a multi-port alphabet. A *multi-port (deterministic) suffix-reading automaton* (written mDSA in short) $\mathcal{A}$ is given by a tuple $(Q, \Sigma, q_0, \delta, F)$ where Q, q_0 and F are a finite set of states, the initial state and a set of accept states, respectively. The transition relation $\delta \subseteq Q \times \Sigma_1^* \times \Sigma_2^* \times \cdots \times \Sigma_k^* \times Q$: each transition is of the form $(q, (u_1 \parallel u_2 \parallel \cdots \parallel u_k), q')$ where $u_i \in \Sigma_i^*$ (not all of them can be ε). We assume there are only finitely many transitions.

Figure 6.2 gives two mDSAs adhering to this syntax.

For the semantics, it is convenient to assume that there is a *tape* on which the automaton writes all its inputs. Initially, the tape is ε. Each time an input letter $a \in \Sigma$ is received, the

automaton appends it to the right of the tape.

Notation. For a word $w = a_1 \dots a_n$, with $a_i \in \Sigma$, we write $w(i)$ to denote the letter a_i. We remark our convention that the indices of w start from 1 (and not 0). The length of w, denoted as $|w|$, equals n (the number of letters). For $j \in \{1, \dots, n\}$ we write $w[i, j]$ to denote the substring $a_i a_{i+1} \dots a_j$.

mDSA semantics. A *configuration* of an mDSA is given by (q, w, θ) where

- q is the current state,
- w is the word read so far, from the beginning
- $\theta \in \mathbb{N}$ marks a position in the word where the last match of a transition appeared.

We refer to the pair (w, θ) as the *tape configuration*.

The formal semantics of an mDSA $\mathcal{A}$ is given by a transition system $\mathcal{S}^{\mathcal{A}} = (S, S_0, \to, S_F)$ over the configurations: the set of nodes S is the set of all configurations (q, w, θ) (notice that this set is infinite); the initial node s_0 is $(q_0, \varepsilon, 0)$, also called *the initial configuration*; S_F is the set of *accepting configurations* which are configurations (q, w, θ) where $q \in F$ and $\theta = |w|$ (signaling reaching an accepting state immediately after a match); to describe the transition relation $\to$, we first explain when a transition label matches with a tape configuration.

Definition 6.2. We say that a transition label $(u_1 \parallel u_2 \parallel \cdots \parallel u_k)$ matches a tape configuration (w, θ) if:

- u_i is a suffix of $\mathsf{P}_i(w)$ for all ports i,
- each u_i appears entirely before or entirely after θ: that is, u_i is a subword of $w[1, \theta]$ or $w[\theta + 1, |w|]$,
- at least one u_i appears entirely after θ.

Let $\rho : (q, (u_1 \parallel u_2 \parallel \cdots \parallel u_k), q')$ be an outgoing transition from q.

- In $\mathcal{S}^{\mathcal{A}}$, there is a following transition:

$$(q, w, \theta) \xrightarrow[u_1 \| u_2 \| \cdots \| u_k]{a} (q', wa, \theta') \qquad \text{if}$$

  $(u_1 \parallel u_2 \parallel \cdots \parallel u_k)$ matches the tape configuration (wa, θ), and $\theta' = |wa|$ (tape head moves to the end of the tape).

- There is a transition of the form:

$$(q, w, \theta) \xrightarrow{a} (q, wa, \theta)$$

  if no transition out of q matches the tape configuration (wa, θ). In this case, the tape head remains at its position.

The transition relation $\rightarrow$ of $\mathcal{S}^{\mathcal{A}}$ is a union over all $\xrightarrow[u_1 \| u_2 \| \cdots \| u_k]{a}$ and $\xrightarrow{a}$.

For all examples described in Section 6.1, the above semantics of transitions adheres to the required intention.

Remark 6.2.1. We remark that the labeled transition system $\mathcal{S}^{\mathcal{A}}$ defined as above is non-deterministic, since multiple transitions can, in principle, be enabled at (q, w, θ) on seeing an a. In order to make the automaton deterministic, a specific priority function can be added to resolve the non-determinism. If multiple transitions match at a configuration, the priority function is used to resolve the non-determinism. We did not explicitly add it to the syntax for technical simplicity.

Remark 6.2.2. A DSA is an mDSA with a single port. To resolve non-determinism, DSAs employ the longest match criterion: among all strings that are a suffix, pick the one with the maximum length.

Concluding remarks for Chapter 6. This chapter successfully extended the suffix-reading paradigm from the sequential domain, as developed in the first part of this thesis, to the more complex realm of concurrent systems. We began by identifying the "interleaving explosion" as a primary barrier to the succinct modeling of concurrent specifications. In response, we

introduced the *Multi-port Suffix-reading Automaton* (mDSA), a model that uses a partitioned alphabet and a parallel composition operator to abstract away from specific event orderings.

The main contribution of this chapter was the development of a novel and robust operational semantics to handle the complex requirements of concurrent event-based reasoning. By introducing a tape-based configuration, (q, w, θ), which maintains the full history of inputs along with a marker for the last transition, we resolved the fundamental dilemma between needing persistent context for some conditions and fresh trigger events for others. We have thus established a formal, well-defined model capable of specifying complex, concurrent, and pattern-based system requirements in a concise and intuitive manner.

While the mDSA model presented here can recognize intricate combinations of inputs across multiple ports, many reactive systems must also *produce outputs* or actions in response. In many practical specifications, an output generated by the system can immediately alter the state of the world, thereby influencing the conditions for subsequent transitions. To faithfully model this behaviour and to provide a formal semantics for industrial notations like EDTs, our model requires one final extension. The next chapter will introduce this feature, defining 'mDSA-with-outputs' and exploring the computationally significant consequences of adding this feedback mechanism.

Chapter 7

Multiport DSA with outputs

The previous chapter extended the suffix-reading paradigm to handle concurrent *inputs* via the Multi-port DSA model. However, this represents only half the behavior of a typical reactive system, which must not only process inputs but also *produce outputs* in response. This chapter, based on the work presented in [KSV26], introduces this final, critical feature, transforming our automaton from a passive recognizer into an active computational model.

A key motivation for this extension is the need to model the powerful feedback loops present in many system specifications. An output generated by one transition can be immediately appended to the system's history, thereby becoming part of the input context for the very next transition. Attempting to model this dynamic by encoding output values into the finite state control would lead to a combinatorial explosion in the number of states. A more direct and scalable approach is to incorporate outputs directly into the transition mechanism.

To this end, this chapter formally defines 'mDSA-with-outputs'. We will extend the transition syntax to include an output component and enhance the semantics to allow for 'epsilon-transitions'; actions that can be triggered by the current history without consuming new external input, enabling the automaton to evolve based on its own generated outputs. We will show that this small extension has significant consequences, leading to a large degree of computational power.

Structure of this chapter. The chapter will proceed as follows. We start with a motivating example. In Section 7.1, we will present the formal syntax and operational semantics for the mDSA-with-outputs model. Second, in Section 7.2 we will analyze its computational complexity, proving that the state reachability problem becomes PSPACE-complete [KSV26]. Finally, in Section 7.3 we will apply this fully-featured model to provide the first formal operational semantics for the industrial 'Expressive Decision Table (EDT)' notation [VSKA14], demonstrating the direct practical relevance of our theoretical investigation.

A motivating example. In Section 6, we have seen multi-port DSAs which process inputs across several ports and match transitions. In many situations, requirements also talk about outputting values to certain ports when transitions match. Here is an example of some requirements of a *Car Alarm module*, adapted from [VSZA15].

1. If the most recent values on *Ignition* (I) and *Alarm* (A) are `Off`, and the last two values on the Panic Switch (P) are `Press;Press`, then the value `On` is output on the port *Alarm*.

2. If the last three values on the Panic Switch are `Release;Release;Release`, and the Alarm is `On`, then output `False` on *Flash* (F) and *Alarm*.

3. If last value on *Ignition* is `On`, then `False` is output on *Flash* and `Off` is output on *Alarm*.

The first requirement checks for I : `Off` $\parallel$ A : `Off` $\parallel$ P : `Press;Press` and immediately writes A : `On`. The check-and-update is a fully atomic operation. One way to model this requirement as an mDSA would be to absorb port A as part of the state: that is the state of the mDSA is defined using values of such ports that are used both for inputs and outputs. However, this would lead to an exponential blowup in the number of states when there are multiple such ports that are both read and written on transitions. To succinctly capture requirements with outputs, we allow both reads and writes on all ports and define *mDSAs with outputs*. Later, in Section 7.3 we will use these mDSAs-with-outputs to give a semantics for the industry notation Expressive Decision Tables (EDTs).

7.1 mDSA-with-outputs: formal syntax and semantics

The syntax of mDSA-with-outputs is almost the same as that of mDSAs. The only difference is in the syntax of transitions. As before, there is a multiport alphabet $\Sigma = \langle \Sigma_1, \ldots, \Sigma_k \rangle$. An mDSA-with-outputs $\mathcal{B}$ is a tuple (Q, Σ, q_0, δ) where Q is a finite set of states, q_0 is the initial state and δ is a finite set of transitions. We do not view these automata as language acceptors, and so we do not specifically define accept states. We will instead discuss state reachability.

Transitions. A transition now looks like:

$$q \xrightarrow[O_1:b_1 \parallel O_2:b_2 \parallel \cdots \parallel O_m:b_m]{I_1:u_1 \parallel I_2:u_2 \parallel \cdots \parallel I_\ell:u_\ell} q'$$

where q, q' are states; each u_j is a word in the port alphabet of I_j; and each b_j is a letter in the port alphabet of O_j. We remark that $\{I_1, \ldots, I_k\} \cap \{O_1, \ldots, O_m\}$ may be non-empty. In other words, some of the ports may appear both as input and output in a transition.

Semantics. Configurations of $\mathcal{B}$ are the same as before, given by (q, w, θ) with q a state, w the word seen so far and θ a position of the tape head. The semantics is a transition system $\mathcal{S}^{\mathcal{B}} = (S, s_0, \to)$ where S is the set of configurations, with s_0 being the initial configuration $(q_0, \varepsilon, 0)$. Let $\rho = (q, (I_1 : u_1 \parallel \cdots I_\ell : u_\ell), (O_1 : b_1 \parallel \cdots \parallel O_m : b_m), q')$. There are three kinds of transitions in $\mathcal{S}^{\mathcal{B}}$:

- *ε-transitions.* $(q, w, \theta) \xrightarrow[\rho]{\varepsilon} (q', wb_1b_2 \ldots b_m, |w|)$ if the input condition $(I_1 : u_1 \parallel \cdots \parallel I_\ell : u_\ell)$ matches the tape configuration (w, θ); in that case move the tape head to the end of w and write all the outputs to the right of w (order of writing does not matter).
- *transitions on input letters.* if no outgoing transition from q has a label that matches the current tape configuration (w, θ), then there is no ε transition out of (q, w, θ) and the mDSA-with-outputs listens to further input letters:
  - $(q, w, \theta) \xrightarrow[\rho]{a} (q', wab_1b_2 \ldots b_m, |wa|)$ if $I_1 : u_1 \parallel \cdots \parallel I_\ell : u_\ell$ matches (wa, θ); then, move the tape head to the end of wa and write the outputs after that,

– $(q, w, \theta) \xrightarrow{a} (q, wa, \theta)$ if no transition out of q matches (wa, θ).

Notice that the transitions on input letters have the same matching semantics as in an mDSA, except now they also write all the outputs to the end of the word. The main change is the presence of ε-transitions. Since we write outputs after matching a transition, on reaching a new state q' there could already be transitions out of q' that match the current tape. Previously, in mDSA, this would not happen, since the tape head would be at the end of the word, and at least one port should see a fully fresh match.

Surprisingly, this ability to produce outputs which can in turn be consumed by input conditions, creates complex behaviours. We illustrate this additional difficulty by presenting an mDSA-with-outputs that can succinctly encode an N-bit counter.

N-bit counter.

We wish to implement addition of an N-bit counter using an mDSA-with-outputs that has 3 states $\{q^{init}, q_0, f\}$, $N+1$ ports and $\mathcal{O}(N)$ transitions. Suppose $b_{N-1}b_{N-2}\dots b_1b_0$ are the N bits, with b_{N-1} the most significant bit and b_0 the least significant bit. Addition can be implemented using N rules: for $j \in \{0, \dots, N-1\}$

- if $b_j = 0$ and $b_{j-1} = b_{j-2} = \cdots = b_0 = 1$, then change $b_j := 1$ and $b_{j-1} = b_{j-2} = \cdots = b_0 := 0$

Starting from 0 for all bits, there will be exactly one rule that will be applicable each time. Applying the relevant rule each time will lead to $b_{N-1} = b_{N-2} = \cdots = b_0 = 1$ in $2^N - 1$ steps.

To model this counter using an mDSA-with-outputs, we can use N ports $b_{N-1}, b_{N-2}, \dots, b_0$ each having two values $\{0, 1\}$. From the initial state q^{init}, we can first read a dummy letter (in a fresh port), write 0 to all ports $b_{N-1}, \dots, b_0$, and move to state q_0. From q_0 to q_0, there is a transition for each $j \in \{0, \dots, N-1\}$ in the following form:

$$q_0 \xrightarrow[b_j{:}1 \parallel b_{j-1}{:}0 \parallel \cdots \parallel b_0{:}0]{b_j{:}0 \parallel b_{j-1}{:}1 \parallel \cdots \parallel b_0{:}1} q_0$$

These transitions start a sequence of $2^N - 1$ ε transitions which finally result in all ports b_{N-1}

to b_0 being 1. To mark the end of the computation, we add an extra transition from q_0 that checks if $b_{N-1} : 1 \parallel b_{N-1} : 1 \parallel \cdots \parallel b_0 : 1$ and moves to f. Notice that the length of the tape when the automaton reaches f is exponential in the size of the input automaton. Therefore, the witness for reachability is exponential. We have crucially used the ability to have ports that can be used both as inputs and outputs. We will now pin down the complexity of the reachability problem in mDSA-with-outputs.

7.2 Complexity of state reachability

We now look at the state reachability problem: given an mDSA-with-outputs $\mathcal{B}$ and a state q, does there exist a word that reaches q? Without outputs, this problem is simply a graph reachability problem, like in a DFA or a DSA. The presence of outputs makes this problem harder, as seen in the N-bit counter.

Theorem 7.1. *Emptiness for mDSA with outputs is PSPACE-complete.*

Proof. *PSPACE upper bound.* Let $\mathcal{B}$ be an mDSA-with-outputs. Consider the transition system $\mathcal{S}^{\mathcal{B}}$ describing the semantics of $\mathcal{B}$. Each node in the transition system is of the form (q, w, θ) where w can become arbitrarily long. Therefore, the transition system, viewed as it is, is infinite. Suppose M is the maximum length among all words used in the transition labels of $\mathcal{A}$. This is the maximum length of a suffix that will be checked for in a transition. To detect whether $u_1 \parallel u_2 \parallel \cdots \parallel u_k$ matches a configuration (q, w, θ), it is sufficient to maintain the last M letters seen in each port. We now explain how this can be implemented.

Assume that in configuration (q, w, θ), every projection $\mathsf{P}_i(w)$ of the word w has at most M letters. We apply the criteria for verifying if $(u_1 \parallel u_2 \parallel \cdots \parallel u_k)$ matches the tape configuration (w, θ) as in Definition 6.2. Now, when we append $a \in \Sigma_i$ to the end of w: if wa contains $M + 1$ letters in port i, that is $|\mathsf{P}_i(wa)| = M + 1$, then let w' be the word obtained by deleting from wa the first occurrence of a letter in Σ_i. Let b be this letter. If b appeared at a position strictly greater than θ in wa, then we do not need to change the tape head ($\theta' = \theta$); otherwise, b appears on or before θ, and we reduce tape position by 1 ($\theta' = \theta - 1$). This gives

a new tape configuration (w', θ'). Transitions point to the new truncated configuration. Call the new truncated transition system as $\mathcal{S}^{\mathcal{B}}_{\mathsf{finite}}$.

This bounds the tape length to $M \times k$, where k is the total number of ports. So, the total size of the truncated transition system is exponential in the size of the input. To verify emptiness, it is sufficient to guess a path in this transition system. Since each node requires polynomial space, and the length of the witness is bounded by an exponential in the size of the input, we deduce a PSPACE upper bound for the emptiness problem, following standard complexity theoretic arguments.

PSPACE-hardness. For the hardness, we give a reduction from this problem: given DFAs $D_1, D_2, \ldots, D_n$ over a common alphabet Σ, the DFA-intersection-emptiness asks if there exists a word in the language of all the n DFAs. This is known to be PSPACE-complete [Koz77]. Let $D_j = (Q_j, q_j^{init}, \delta_j, F_j)$ be the description of DFA D_j. The mDSA-with-outputs $\mathcal{B}^D$ that we will construct makes use of n ports to store the state reached by each of the DFAs.

We will have one input port I with Σ as alphabet, and n ports S_j (for $j \in \{1, \ldots, n\}$), with $\Sigma_j = Q_j$ being the states of DFA D_j. We add a special port L with alphabet $\$$ that will be used for book-keeping. Let s^{init} be the initial state of $\mathcal{B}^D$. There is an initial transition that reads the special character $\$$ and writes the initial state of each DFA to the respective ports, and moves to s_0:

$$s^{init} \xrightarrow[S_1:q_1^{init} \parallel S_2:q_2^{init} \parallel \cdots \parallel S_n:q_n^{init}]{L:\$} s_0$$

From s_0 we have a gadget that reads the next input and then sequentially updates the states of each DFA by writing the new states in the respective ports.

- the process starts with a transition $s_0 \xrightarrow[L:\$]{I:a} s_1^a$;
- there are states s_j^a for all $j \in \{1, \ldots, n\}$, and for all $a \in \Sigma$;
- for every transition (q_j, a, q_j') in D_j we have: $s_j^a \xrightarrow[L:\$ \parallel S_j:q_j']{L:\$ \parallel S_j:q_j} s_{j+1}^a$, with $s_{n+1}^a := s_0$.

When an a is read at s_0, the automaton moves to s_1^a and the tape head moves to the position of a. Since the same transition also writes a $\$$ to L, there will be an outgoing transition from

s_1^a that matches: this transition would be the one corresponding to the current state of D_1. Then, the automaton goes to s_2^a, and the process continues until the automaton comes back to s_0, giving rise to a sequence of n ε-transitions from s_1^a back to s_0.

This is the basic construction that simulates the n DFAs using an mDSA-with-outputs. To detect acceptance, we can add special ports $B_1, B_2, \ldots, B_n$. Whenever we reach an accepting state in Q_j we write 1 to this port, and when we reach a non-accepting state, we write 0. Add a transition $B_1 = 1 \parallel B_2 = 1 \parallel \cdots \parallel B_n = 1$ from s_0 to a special state f. The intersection of $D_1, \ldots, D_n$ is empty iff f is reachable.

For every letter a and every port j there is a state s_j^a. There are transitions s_j^a to s_{j+1}^a for each every transition in D_j. Total number of states is $(|\Sigma| \cdot n) + 2$, and the total number of transitions is proportional to $\sum_j \delta_j$. Therefore, the overall construction is polynomial-time.

□

7.3 Expressive decision tables

As mentioned earlier in this document, Expressive Decision Tables (EDTs) [VSKA14] are a tabular notation for specifying software requirements and have been used in various industrial settings [VSZA15, AVS+20]. An EDT specification consists of tables where columns specify input, output and input/output (I/O) ports, and rows specify the relationship between input and output values on these ports. Each port is associated with an alphabet and each cell in the table consists of a string of letters from this alphabet. The original work on EDTs contained timing constraints in the cells and interpreted the EDT over a notion of discrete-time. In this document we consider EDT without time. A cell in an EDT table is said to match when the string given in that cell is a suffix of the values seen for the input port corresponding to that cell's column. A row of an EDT matches when all the cells corresponding to that row match and one cell matches on the last input. Table 7.1 gives the EDT specification of the Car Alarm module described in the beginning of this chapter. We write F for `Off` and `False`, N for On, P for `Press` and R for `Release`.

In the example EDT the input ports are prefixed with *in* and output ports are prefixed with

TABLE 7.1: EDT for an Alarm module

s.no	in I	in P	in A	out A	out F
1	F	P;P	F	N	
2		R;R;R	N	F	F
3	N			F	F

out. Input/Output ports are prefixed with both *in* and *out*. These are the ports that appear on both sides of the EDT, as inputs as well as outputs.

A *test case* for a requirement is a sequence of values that match the EDT row corresponding to that requirement. The use of EDTs and test case generation algorithms from EDT specifications have been widely studied [VSZA15, AVS+20]. However, the notation lacks a rigorous formal semantics, except for a brief description of key elements of formal semantics in [VSKA14]. Hence, none of the test case algorithms use formal techniques like model checking and instead rely on random generation using heuristics. These algorithms, therefore, cannot prove that an EDT row cannot be matched and also find it hard to match certain rows. We overcome both these limitations by providing a formal semantics for EDTs (without timing) through a translation to mDSA-with-outputs.

7.3.1 Translating EDT to mDSA-with-outputs

An EDT with C columns and R rows is translated to an mDSA, with one state q_0 and an alphabet $\Sigma = \langle \Sigma_1 \cdots \Sigma_{|C|} \rangle$. Each Σ_i corresponds to the port of column i and the letters of the alphabet correspond to the values that the signal can take, prefixed by $\langle Portname :\rangle$. Thus if the Port name of the first column is A and its alphabet is $a_1, a_2, \cdots, a_k$ then the corresponding alphabet $\Sigma_1 = A : a_1, A : a_2 \cdots A : a_k$. Each EDT row is translated to an mDSA as follows:

- The mDSA has exactly one state q_0
- There is a transition corresponding to each row with q_0 as both the source and target state.
- The label of each transition consists of all the strings in the cells of the input columns of the row combined using the $\|$ operator.

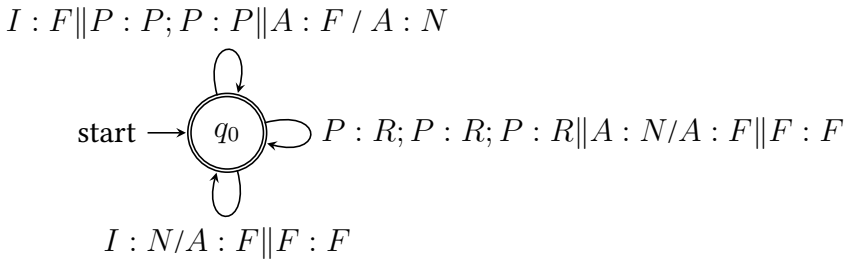


FIGURE 7.1: mDSA of the Alarm EDT

- The output of the transitions are the strings in the cell corresponding to the output columns, combined using the $\|$ operator.

The mDSA in Figure 7.1 is an mDSA specification corresponding the EDT given in Table 7.1

To generate a test for a certain row r, we can first add a special state q_r, and make the transition corresponding to this row point to q_r instead of q_0. A test for r then reduces to a witness for the reachability of q_r.

7.3.2 Experiments

We have implemented a prototype test generator for EDTs using the translation of EDTs to mDSA-with-outputs [AK25]. In our implementation, we translate the given EDT $\mathcal{T}$ into an mDSA-with-outputs $\mathcal{B}^{\mathcal{T}}$ and simulate the semantics of $\mathcal{B}^{\mathcal{T}}$ using the finite transition system $\mathcal{S}^{\mathcal{B}^{\mathcal{T}}}_{\text{finite}}$ described in the proof of Theorem 7.1.

First we look at a couple of small EDTs. Suppose we want a testcase for Row 1 in Table 7.1. Encoding this in our tool gives a testcase $(P; P)$ for Row 1, when default values for I and A are set to F. Next we look at Table 7.2, a partial representation of a wiper module in a car. The tool finds a testcase (park;notpark) for Row 2.

TABLE 7.2: EDT for a Wiper module

sno	in ignition	in wiperswitch	in parksensor	in error	out wipercmd	out error
1	on	on		false	wipe	
2			(park;notpark)		dontwipe	true

We now present an EDT for which our tool can solve the test generation problem, whereas randomized techniques will have low success probability. The EDT is a slight variation of the

TABLE 7.3: Implementing a binary counter for 3 bits

T	t_2	t_1	t_0	S_T	t_2	t_1	t_0	S_T
✓								+
			0	+			1	−
		0	1	+		1	0	−
	0	1	1	+	1	0	0	−

N bit counter presented in Section 7.1. We depict a part of the EDT for three bits in Table 7.3. In addition we add a row to check for all $t_i = 1$, and also have an extra letter in the port T. Therefore there is exactly one sequence that makes all the t_i equal to 1. Randomized techniques have a low probability of generating such a test case. Our prototype implementation generates a test case for $N = 5$.

Our goal with the experiments was not to present a scalable tool for EDT test generation. We aimed to create a simulator for mDSA-with-outputs, which we have applied on EDT test case generation. As future work, we plan to investigate scalable algorithms for reachability in mDSA-with-outputs.

Concluding remarks for Chapter 7. This chapter presented the final evolution of the suffix-reading automaton model: the 'mDSA-with-outputs'. By incorporating the ability for transitions to atomically produce outputs that are written back to the history tape, we created a model capable of specifying complex, reactive, and concurrent systems, as shown in [KSV26]. The central new dynamic introduced was the feedback loop enabled by 'epsilon-transitions', which allow the automaton to execute a cascade of actions based on its own generated outputs without requiring external stimuli.

The impact of this feature was captured in the chapter's main theoretical result: the proof that state reachability for mDSAs-with-outputs is PSPACE-complete [KSV26]. The N-bit counter example demonstrated how the history tape can be used as a working memory of exponential size, formally capturing the computational power unlocked by this input-output mechanism.

The practical need to provide a formal foundation for the industrial 'Expressive Decision Table (EDT)' notation [VSKA14] motivated the theoretical extension from DSA to mDSAs-

with-outputs. With the introduction and analysis of the mDSA-with-outputs, our development of the suffix-reading automata family is now complete. We have journeyed from a foundational model for sequential patterns to a powerful formalism for concurrent, reactive systems. The final chapter will now step back to summarize the overarching contributions and implications of this entire body of work, and to outline promising avenues for future research.

Chapter 8

Conclusion

This thesis set out to address a fundamental and persistent challenge in computer science: the concise and intuitive formal modelling of complex systems. While Deterministic Finite Automata (DFAs) provide the theoretical bedrock for specifying and verifying regular properties, their practical application is often hampered by the state explosion problem. For specifications that are pattern-intensive or inherently concurrent, the number of states required in a classical DFA can become unmanageably large, rendering the resulting models difficult to create, comprehend, and analyze. Existing extensions to DFAs, such as symbolic or generalized automata [VHL+12, GM99], offer partial solutions by tackling alphabet size or fixed-sequence matching, but they fall short in scenarios where system behaviour is dependent on sparsely distributed patterns separated by irrelevant sequences of events.

The research presented in this thesis introduces and develops a new automaton paradigm (suffix-reading) to fill this gap. We began by formalizing the **Deterministic Suffix-reading Automaton (DSA)** [KSVV26], a model that enriches classical automata with the ability to trigger transitions based on suffixes of the input word read so far. This work has established the theoretical foundations of this model, extended it to the crucial domain of concurrent systems with the **Multi-port DSA (mDSA)** [KSV26], and demonstrated its practical utility by providing the first formal semantics for an industrial specification language (EDT). In doing so, this research offers a novel perspective on automata theory and its application, bridging the gap between high-level system requirements and executable formal models.

Summary of Contributions

The contributions of this thesis are twofold, corresponding to the theoretical investigation of the single-stream DSA model and its extension and application to concurrent systems.

1. Deterministic Suffix-reading Automata: Modelling Pattern-Intensive Specification

The first major contribution of this work was the introduction, formalization, and analysis of the Deterministic Suffix-reading Automaton. Unlike traditional automata that process input letter-by-letter, a DSA operates on a "wait-and-jump" principle. It can remain in a state, passively consuming input, until a sequence of symbols matching a transition's label appears as a suffix of the word seen so far. This capability is particularly powerful for "pattern-intensive" languages, where key events are interspersed with arbitrary, "don't care" sequences.

We formally proved that DSAs recognize exactly the class of regular languages. Their value lies not in greater expressive power, but in succinctness. We established that for certain families of languages, DSAs can be exponentially more concise than their minimal DFA equivalents (Lemma 3.6, Chapter 3). To facilitate meaningful comparison, we defined the **total size** of a DSA as the sum of its states, edges, and the lengths of its word-based labels, a more faithful measure of complexity than state count alone.

A significant technical contribution was the development of a **DFA-to-DSA derivation procedure** (Chapter 4). This constructive algorithm provides a systematic method for obtaining a language-equivalent DSA from a given DFA by identifying and "suppressing" intermediate states. We identified sufficient conditions for this conversion to be correct, formalizing them through the idea of **suffix-tracking sets**, via **suffix-compatible transitions**, and **well-formed sets**. These conditions ensure that collapsing a path of DFA transitions into a single, suffix-reading DSA transition preserves the original language (Theorem 4.10).

The investigation into finding a *minimal* DSA for a given language yielded several important results. We demonstrated that, unlike for DFAs, a minimal DSA is not necessarily unique (Lemma 5.1). More surprising was the discovery that starting our derivation procedure from the canonical (minimal) DFA of a language does not guarantee a minimal DSA. In some cases,

a smaller DSA can be derived from a larger, non-minimal DFA (Figures 5.2, 5.3). This insight reveals a fundamental complexity in DSA minimization: the optimal structure for a suffix-reading model may be obscured by the structure of a minimal letter-by-letter model. Building on this, we proved that the problem of deciding if a given language has a DSA of total size less than or equal to a given bound k is **NP-complete** (Theorem 5.3). This hardness result underscores the challenge of optimal DSA synthesis and justifies the use of non-canonical methods, such as our derivation procedure, for practical applications.

In response to these complexity challenges, we defined a restricted subclass, **Strongly Deterministic Suffix-reading Automata (sDSAs)**, which imposes a syntactic constraint on outgoing labels to prevent prefix-suffix ambiguities. For this class, we proved a positive result: every minimal sDSA *can* be derived from the canonical DFA (Theorem 5.7). This finding delineates a tractable subset of DSAs for which minimization is more straightforward and provides a path towards avoiding the complexity of the general case.

2. Multi-port DSA: Handling Concurrency and Enabling Practical Application

The second part of this thesis extended the suffix-reading paradigm to concurrent systems, which represent a major source of state explosion for classical models. We introduced the **Multi-port Deterministic Suffix-reading Automaton (mDSA)**, a model where the alphabet is partitioned across multiple ports, and transitions are triggered by a conjunction of suffix patterns occurring across these independent streams. A transition labeled $u_1 \| u_2$, for instance, fires only when a suffix matching u_1 has been observed on port 1 and a suffix matching u_2 has been observed on port 2 (in any order across the ports).

A key challenge was defining a coherent operational semantics. A naive approach of simply consuming matched prefixes was shown to be inadequate for specifications where past events on one port must remain context for future events on another (Section 6.1). Our solution was to equip the mDSA with a memory of the input history on a tape, along with a marker θ, indicating the position of the last transition match. A new transition can fire only if at least one of its component patterns appears entirely after this marker, resolving the ambiguity between needing "fresh" events versus persistent context (Definition 6.2).

To model modern reactive systems, we further extended the model to **mDSAs-with-outputs**, allowing transitions to atomically write values to ports upon firing. This feature, where outputs can immediately influence the conditions for subsequent transitions, makes the model more powerful but also computationally more complex. We demonstrated that these automata are powerful enough to succinctly encode an N-bit counter, where the length of the execution trace grows exponentially with the size of the automaton. This led to our second major complexity result: **state reachability for mDSAs-with-outputs is PSPACE-complete** (Section 7.2).

Finally, we grounded this theoretical work in a practical application by using mDSAs-with-outputs to provide the first formal operational semantics for **Expressive Decision Tables (EDT)**, a tabular notation used in industry for specifying software requirements [VSKA14, AVS+20]. We developed a translation from the core components of the EDT notation into an mDSA-with-outputs, thereby making EDT specifications amenable to formal analysis [KSV26, Section 5.1]. Test case generation for an EDT row reduces to the state reachability problem in the corresponding mDSA. We implemented a prototype tool based on this translation, demonstrating its ability to automatically generate test cases for intricate, concurrent specifications that would be challenging for randomized or heuristic-based methods (Section 7.3.2).

Limitations and Future Work

While this thesis establishes a solid foundation for suffix-reading automata, it also indicates the boundaries of current work and opens several promising avenues for future research.

The NP-completeness of DSA minimization and the PSPACE-completeness of mDSA reachability are significant theoretical hurdles. Our work justifies the use of heuristics, but the development of sophisticated approximation algorithms with provable performance guarantees is an important next step. It would also be valuable to identify broader classes of languages or automata, beyond sDSAs, for which minimization is tractable.

The operational semantics of mDSAs, while precise, are more complex than those of classical automata. This trade-off between expressive convenience and semantic simplicity war-

rants further investigation, including studies to assess the "readability" and practical usability of the mDSA model for engineers. The PSPACE-hardness of the reachability problem also indicates a potential benefit in exploring techniques from symbolic reachability analysis.

Looking forward, this work suggests several exciting research directions:

- **Automata Learning:** A compelling direction is to develop learning algorithms, in the spirit of Angluin's L*, to automatically infer DSA or mDSA models from system observations. Such a technique would enable the modeling of black-box systems, facilitating their analysis and verification without requiring manual specification.
- **Real-Time and Probabilistic Extensions:** Many real-world systems have real-time constraints. Extending the mDSA model with clocks and timing constraints, as hinted at by the original context of EDTs [VSKA14], would vastly increase its applicability to cyber-physical and embedded systems.
- **Hybrid Models:** As DSAs tackle pattern-length complexity and symbolic automata handle alphabet-size complexity, a hybrid model combining both paradigms is a natural and powerful extension. A "symbolic DSA" could use predicates on characters within its suffix-based word labels, offering a unified solution to two orthogonal sources of state explosion.
- **Enhanced Tooling and Scalable Analysis:** The prototype mDSA tool demonstrates feasibility [AK25], but scalable analysis for industrial-sized problems will require more advanced techniques. Exploring symbolic model checking, abstraction-refinement, and other static analysis methods to tackle the PSPACE-hard reachability problem is a critical avenue for future practical impact. Furthermore, investigating closure properties (e.g., union, intersection) directly on the DSA model could lead to more efficient composition and analysis algorithms.

Concluding Remarks

The journey from the abstract theory of computation to the concrete practice of software engineering is fraught with challenges of scale and complexity. This thesis attempts to bridge that

divide by proposing a new automaton model designed with the structure of modern software specifications in mind. The suffix-reading paradigm moves away from the low-level, letter-by-letter view of computation and toward a higher-level, pattern-centric perspective that aligns more closely with how developers and system architects reason about system behaviour.

By establishing the theoretical properties of DSAs, uncovering the complexities of their minimization, extending them to handle the concurrency inherent in modern systems, and applying them to a practical industrial notation, this work offers a complete, end-to-end research narrative. It demonstrates that the pursuit of more succinct and intuitive formal models is not merely an academic exercise but a vital step toward building more reliable and robust software systems. The theory and tools for suffix-reading automata presented here can hopefully provide a new lens through which to view regular languages, and enable a new instrument in the toolkit of the verification scientist and the software engineer.

Bibliography

[AK25] Arjun Arul and R. Keerthan. mdsa-based test generator for edt. https://github.com/KeerthanR/mDSA/tree/main/code, 2025.

[AVS+20] Supriya Agrawal, R. Venkatesh, Ulka Shrotri, Amey Zare, and Sagar Verma. Scaling test case generation for expressive decision tables. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*, pages 364–374. IEEE, 2020.

[BG92] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[BHV04] Ahmed Bouajjani, Peter Habermehl, and Tomás Vojnar. Abstract regular model checking. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, pages 372–386, 2004.

[BJNT00] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, pages 403–418, 2000.

[BM63] Janusz A. Brzozowski and Edward J. McCluskey. Signal flow graph techniques for sequential circuit state diagrams. *IEEE Trans. Electron. Comput.*, 12(2):67–76, 1963.

[Bro24] BrowserStack. What is model-based testing in software testing. https://www.browserstack.com/guide/model-based-testing, 2024.

[CGK+18] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. *Model checking, 2nd Edition.* MIT Press, 2018.

[D'A] Loris D'Antoni. Symbolic automata. https://pages.cs.wisc.edu/~loris/symbolicautomata.html.

[DNSVT07] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, WEASELTech '07, page 31–36, New York, NY, USA, 2007. Association for Computing Machinery.

[DV17] Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, pages 47–67, 2017.

[Eil74] Samuel Eilenberg. *Automata, languages, and machines. A.* Pure and applied mathematics. Academic Press, 1974.

[Fer09] Henning Fernau. Algorithms for learning regular expressions from positive data. *Information and Computation*, 207(4):521–541, 2009.

[GH96] Noa Globerman and David Harel. Complexity results for two-way and multi-pebble automata and their logics. *Theor. Comput. Sci.*, 169(2):161–184, 1996.

[GH01] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 412–416, 2001.

[GM99] Dora Giammarresi and Rosa Montalbano. Deterministic generalized automata. *Theoretical Computer Science*, 215(1-2):191–208, 1999.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[Has91] Kosaburo Hashiguchi. Algorithms for determining the smallest number of non-terminals (states) sufficient for generating (accepting) a regular language. In *Automata, Languages and Programming, 18th International Colloquium, ICALP91, Madrid, Spain, July 8-12, 1991, Proceedings*, pages 641–648, 1991.

[HJJ07] Stefan Haar, Claude Jard, and Guy-Vincent Jourdan. Testing input/output partial order automata. In *Proceedings of the 19th IFIP TC6/WG6.1 International Conference, and 7th International Conference on Testing of Software and Communicating Systems*, TestCom'07/FATES'07, page 171–185, Berlin, Heidelberg, 2007. Springer-Verlag.

[HMU07] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition.* Pearson international edition. Addison-Wesley, 2007.

[HW04] Yo-Sub Han and Derick Wood. The generalization of generalized automata: Expression automata. In *Implementation and Application of Automata, 9th International Conference, CIAA 2004, Kingston, Canada, July 22-24, 2004, Revised Selected Papers*, pages 156–166, 2004.

[JR93] Tao Jiang and Bala Ravikumar. Minimal NFA problems are hard. *SIAM J. Comput.*, 22(6):1117–1141, 1993.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, pages 85–103, 1972.

[Koz77] Dexter Kozen. Lower bounds for natural proof systems. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, page 254–266, USA, 1977. IEEE Computer Society.

[KSV26] R. Keerthan, B. Srivathsan, and R. Venkatesh. An automaton model to succinctly represent suffix-based specifications of a concurrent system. In Salem Lahlou and Madhavan Mukund, editors, *Networked Systems*, pages 129–145, Cham, 2026. Springer Nature Switzerland.

[KSVV24] R. Keerthan, B. Srivathsan, R. Venkatesh, and Sagar Verma. Deterministic suffix-reading automata. In Antonis Achilleos and Adrian Francalanza, editors, *Proceedings Fifteenth International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2024, Reykjavik, Iceland, 19-21 June 2024*, volume 409 of *EPTCS*, pages 70–87, 2024.

[KSVV26] R Keerthan, B Srivathsan, R Venkatesh, and Sagar Verma. Deterministic suffix-reading automata. *Logical Methods in Computer Science*, Volume 22, Issue 2, May 2026.

[LLGS18] Wenbin Li, Franck Le Gall, and Naum Spaseski. A survey on model-based testing tools for test case generation. In Vladimir Itsykson, Andre Scedrov, and Victor Zakharov, editors, *Tools and Methods of Program Analysis*, pages 77–89, Cham, 2018. Springer International Publishing.

[LMS22] Sylvain Lombardy, Victor Marsault, and Jacques Sakarovitch. *Awali, a library for weighted automata and transducers (version 2.2)*, 2022. Software available at http://vaucanson-project.org/Awali/2.2/.

[MMW09] Mehryar Mohri, Pedro J. Moreno, and Eugene Weinstein. General suffix automaton construction algorithm and space bounds. *Theor. Comput. Sci.*, 410(37):3553–3562, 2009.

[Pet66] Carl Adam Petri. Communication with automata. DTIC Research Report AD0630125, 1966.

[UPL12] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, August 2012.

[VHL+12] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj S. Bjørner. Symbolic finite state transducers: algorithms and applications. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 137–150, 2012.

[VSKA14] R. Venkatesh, Ulka Shrotri, G. Murali Krishna, and Supriya Agrawal. EDT: A specification notation for reactive systems. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6, 2014.

[VSZA15] R. Venkatesh, Ulka Shrotri, Amey Zare, and Supriya Agrawal. On generating test cases from EDT specifications. In Leszek A. Maciaszek and Joaquim Filipe, editors, *Evaluation of Novel Approaches to Software Engineering - 10th International Conference, ENASE 2015, Barcelona, Spain, April 29-30, 2015, Revised Selected Papers*, volume 599 of *Communications in Computer and Information Science*, pages 1–20. Springer, 2015.

[ZLL02] Marc K. Zimmerman, Kristina Lundqvist, and Nancy G. Leveson. Investigating the readability of state-based formal requirements specification languages. In *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 33–43, 2002.

Appendix A

Bounded test generation

This appendix illustrates a solver-based approach to test case generation, in the context of EDTs. The encoding is based on a version of the semantics prior to the introduction of mDSA-with-outputs, from an unpublished document. But the idea is readily configurable using the semantics detailed in this thesis as well; the corresponding rewriting is omitted.

A.1 Mealy machines

We recall the definition of a Mealy machine below.

Definition A.1 (Mealy machine)**.** Let Σ and Γ be finite input and output alphabets respectively. A *Mealy machine* is a tuple $(Q, q^0, \Sigma, \Gamma, \delta)$ where Q is a finite set of states, q^0 is the initial state and $\delta : Q \times \Sigma \to \Gamma^* \times Q$ is a partial transition function. A transition $q \xrightarrow[w]{a} q'$ maps the source state q and input letter $a \in \Sigma$ to the output word $w \in \Gamma^*$ and target state q'.

A run of a Mealy machine is a sequence of transitions starting from the initial state: $q_0 \xrightarrow[w_0]{a_0} q_1 \xrightarrow[w_1]{a_1} \cdots \xrightarrow[w_k]{a_k} q_{k+1}$. Since the machine is deterministic, there is a unique run for every input word $a_0 a_1 \ldots a_k \in \Sigma^*$. The semantics of the Mealy machine is given by the set of paired words $\{(a_0, w_0)(a_1, w_1) \cdots (a_k, w_k)\}$ obtained from each run of the above form. We denote the semantics of Mealy machine M by $\mathcal{L}(M)$.

A.2 Basic EDT

We look at a fragment of EDT that contains only a basic feature: each row can check for atomic conditions on the input side and output a single value on the output side. Table A.1 gives a basic EDT, which will be used as a running example in this section. We will formalize what it means to say that a row (which we will call a rule) matches in a given sequence of inputs, and what to do if multiple rules match. Based on this definition, we will get a Mealy machine representation for basic EDTs.

TABLE A.1: A Basic EDT $\mathcal{T}_1$

sno	I_1	I_2	S_1	S_2	S_1	S_2	O_1
1	a				1		
2	b		1		0	1	
3		c	0	1			o_1
4		c					o_2

Let $\mathcal{I} = \{I_1, I_2, \dots\}$ be a set of input variables, $\mathcal{S} = \{S_1, S_2, \dots\}$ a set of local variables (also called *state* variables) and $\mathcal{O} = \{O_1, O_2, \dots\}$ a set of output variables. We denote by $\mathcal{V}$ the set $\mathcal{I} \cup \mathcal{S} \cup \mathcal{O}$ of all variables. For each variable $X \in \mathcal{I} \cup \mathcal{S} \cup \mathcal{O}$, let Σ_X be a finite alphabet denoting the set of values that X can take. We will call the union $\mathcal{I} \cup \mathcal{S}$ as *input-side variables* and $\mathcal{S} \cup \mathcal{O}$ as *output-side variables*. Each variable is assumed to have a specified *default value*. For Table A.1, let us take the default values $I_1 = a'$, $I_2 = c'$ (both these letters a', c' do not appear in any rule), $S_1 = 0$, $S_2 = 0$ and $O_1 = o_1$.

A *condition* is a term of the form $(X = b)$ where $X \in \mathcal{I} \cup \mathcal{S}$ is an input-side variable and $b \in \Sigma_X$. A *guard* is a conjunction of conditions such that there is at most one condition involving each input-side variable: $(I_1 = b) \wedge (S_1 = 1)$, with the symbol $\wedge$ denoting conjunction, is the guard coming from row 2. An *assignment* is of the form $Y := b$ where $Y \in \mathcal{S} \cup \mathcal{O}$ is an output-side variable and $b \in \Sigma_Y$. An *update* is a set of assignments such that there is at most one assignment per output variable: $S_1 := 0 \wedge S_2 := 1$ is the update of row 2. For any guard G and update U, we write $\mathrm{Var}(G)$, $\mathrm{Var}(U)$ respectively for the set of variables present in G and U.

Definition A.2 (Basic EDT)**.** A *basic rule* (also called a *row*) R is a guarded update of the form $G \Rightarrow U$, where G is a guard on input-side variables, and U is an update to output-side variables. A *basic EDT* is an ordered set of rules $R_1, \ldots, R_m$, where each R_i is of the form $G_i \Rightarrow U_i$. Rule R_i is said to have *higher priority* than rule R_j if $i < j$.

The running example can be written as the rules: $R_1 : (I_1 = a) \Rightarrow (S_1 := 1)$, $R_2 : (I_1 = b) \wedge (S_1 = 1) \Rightarrow (S_1 := 0) \wedge (S_2 := 1)$ and so on for R_3 and R_4. Intuitively, a stand-alone basic rule $G \Rightarrow U$ says that whenever the input-side guard G becomes true, perform the output-side update U. However as an EDT comprises of a set of rules, multiple rules can simultaneously become true. Moreover, changes to local variables are assumed to be "instantaneous" and hence an update U_i can immediately trigger a new guard G_j whose update U_j triggers a G_l and so on. We will first define the semantics of rule match given a stream of inputs and local variables, for which we need some preliminary notions.

A *partial valuation* of $\mathcal{V} = \mathcal{I} \cup \mathcal{S} \cup \mathcal{O}$ is a partial function mapping $X \in \mathcal{V}$ to $a \in \Sigma_X$. Not all $X \in \mathcal{V}$ needs to have a value. We use partial valuations to model signals that only change the value of some of the variables. However at any point in time, all the variables have a defined value in our semantics. For a partial valuation u, we write $dom(u)$ to denote the domain of u. Consider a sequence $\pi : u_0 u_1 \ldots u_n$ of partial valuations of the entire set of variables $\mathcal{V}$. We will say that π is *relevant* if it starts from the valuation u_0 that maps every variable to its default value. We define v_π, a total valuation over input-side variables $\mathcal{I} \cup \mathcal{S}$: for $X \in \mathcal{I} \cup \mathcal{S}$, $v_\pi(X) = u_i(X)$ where $i \leq n$ is the largest index with $X \in dom(u_i)$ — essentially this gives the *last seen* value for each input-side variable at π. This valuation will be needed for the guard match semantics.

Definition A.3 (Semantics of rule match)**.** Let $\pi : u_0 u_1 \ldots u_n$ be a relevant sequence of partial valuations of $\mathcal{V}$. A condition $X = a$ matches at π if $v_\pi(X) = a$. A guard G matches at π if every condition $(X = a)$ of G matches at π. A guard is said to *freshly match* at π if it matches at π and $X \in dom(u_n)$ for at least one $X \in \mathrm{Var}(G)$. A rule $G \Rightarrow U$ matches at π if G freshly matches at π.

For example: sequence $\pi_1 : u_0 \langle I_1 = a \rangle$ matches rule 1 of Table A.1, but it does not match

any other rule. The sequence $\pi_1' : u_0 \langle I_1 = a\rangle \langle S_1 = 1\rangle$ obtained after the trigger of rule 1, does not match rule 1 since this is not a fresh match. A new occurrence of a is needed to trigger rule 1 again. The sequence $\pi_2 : u_0 \langle I_1 = a\rangle \langle S_1 = 1\rangle \langle I_1 = b\rangle \langle S_1 = 0, S_2 = 1\rangle \langle I_2 = c\rangle$ matches two rules: 3 and 4. The updates of all rules need to be performed at π_2, however there is a conflict in the updates: rule 3 updates O_1 to o_1 and rule 4 updates O_1 to o_2. In such a case of conflict, the rule with the higher priority gets triggered, here it will be rule 3. We have incorporated the updates to state variables already in π_2: notice that the first $I_1 = a$ is followed by $S_1 = 1$ due to trigger of rule 1. Notice also that there is an alternation between inputs and state changes.

Definition A.4 (Conflict between rules, Semantics of rule trigger)**.** Two rules R_i and R_j are said to be in *conflict*, written as $R_i \# R_j$, if they perform different updates on some variable: $U_i(Y) \neq U_j(Y)$ for some $Y \in \mathcal{S} \cup \mathcal{O}$.

A rule R_i is *triggered* at sequence π if R_i matches at π and no higher priority row conflicting with R_i matches at π: that is, there is no R_j with $j < i$ such that $R_i \# R_j$ and R_j matches at π. A sequence π is said to be *stable* if no rule is triggered at π, otherwise it is *unstable*.

Sequences π_1 and π_2 are unstable since row 1 is triggered at π_1 and row 3 is triggered at π_2. For each unstable sequence π of partial valuations over $\mathcal{V}$, a natural update valuation U_π over output-side variables $\mathcal{S} \cup \mathcal{O}$ is defined: for each $Y \in \mathcal{S} \cup \mathcal{O}$, we have $U_\pi(Y) = U_i(Y)$ for some R_i that is triggered at π. Due to the conflict-free property, the choice of i does not matter. For example $U_{\pi_1} = \langle S_1 = 1\rangle$ and $U_{\pi_2} = \langle O_1 = o_1\rangle$. The sequence π_2 has a special property. After each prefix that ends with inputs, the immediate next set of state values is given by the update function of the prefix. Suppose $\pi_3 := u_0 \langle I_1 = a\rangle \langle S_1 = 1\rangle \langle I_1 = b\rangle$. The next letter in the sequence is $\langle S_1 = 0, S_2 = 1\rangle$ which is essentially U_{π_3}. This gives a notion of valid sequences which describe the executions of the EDT.

Definition A.5 (Semantics of basic EDT)**.** The set of *executions* of an EDT are defined inductively as follows: the initial sequence u_0 is an execution; for every stable sequence π, the one-step extension πu is an execution for every partial valuation u such that $dom(u) \subseteq \mathcal{I}$; for every unstable sequence π, the sequence πu where u is U_π, is an execution. The set $\mathcal{L}(\mathcal{T})$ of

all executions of $\mathcal{T}$ gives the semantics of basic EDT $\mathcal{T}$. EDT $\mathcal{T}$ is said to be *consistent* if from any unstable sequence π, there is a bounded sequence of updates w such that πw is a stable sequence.

A.2.1 Mealy machines for basic EDTs

The guards on rules depend solely on the values of the input-side variables, and not on the actual value of the outputs. Moreover, to track if a guard matches at π, it is enough to know valuation v_π giving the last seen values, and the set of variables that have a fresh value, that is the set $dom(u_n)$. This motivates the next definition: a *configuration* of a basic EDT is a pair $(v,)$ where v is a valuation of input-side variables $\mathcal{I} \cup \mathcal{S}$ and $\subseteq \mathcal{I} \cup \mathcal{S}$ is a subset of input-side variables, maintaining where the last change occurred. The configuration associated to $\pi : u_0 \dots u_n$ is $(v_\pi, _\pi)$ where v_π is as defined before and $_\pi = dom(u_n)$. The semantics of a rule match can be deciphered entirely from the configuration. We rewrite it for completeness: a rule $G \Rightarrow U$ *matches* at a configuration $(v,)$ if (1) guard G is true: $v(X) = b$ for every $(X = b)$ in G, and (2) it was a fresh match: $\mathrm{Var}(G) \cap \ \neq \emptyset$. The semantics of rule trigger remains the same. A configuration is stable if no rule is triggered in it, and it is unstable otherwise. As before, the update function $U_{(v,)}$ is defined as the aggregated updates of the (non-conflicting) rules that are triggered at $(v,)$. Observe that the number of configurations is finite. We use these configurations to build a Mealy machine $M_\mathcal{T}$.

Definition A.6 (Mealy machine $M_\mathcal{T}$ of basic EDT $\mathcal{T}$)**.** The input alphabet Σ_{in} is the set of partial valuations of $\mathcal{I}$ union a special symbol τ to denote an "internal" action. The output alphabet Γ_{out} is the set of partial functions of $\mathcal{S} \cup \mathcal{O}$ union the special symbol τ.

States of this Mealy machine are the configurations of EDT $\mathcal{T}$ union a special state $\{\bot\}$ which is the initial state. Stable states wait for inputs, and unstable states perform the updates given by the rules that are triggered in that state. This gives rise to the transition relation, which we make precise next.

From the initial state there is a transition $\bot \xrightarrow[u_0^{s,o}]{u_0^i} (u_0^{i,s}, \mathcal{I} \cup \mathcal{S})$, where u_0^i, $u_0^{s,o}$ and $u_0^{i,s}$ are u_0 restricted to $\mathcal{I}$, $\mathcal{S} \cup \mathcal{O}$ and $\mathcal{I} \cup \mathcal{S}$ respectively. Transitions for all other states are as

follows. From stable states $(v,)$ there is a transition $(v,) \xrightarrow[\tau]{\alpha} (v', ')$ for each $\alpha \in \Sigma_{\text{in}}$, where $v'(X) = \alpha(X)$ for every $X \in dom(\alpha)$ and $v'(X) = v(X)$ otherwise, and $' = dom(\alpha)$. From unstable states $(v,)$ there is a unique transition $(v,) \xrightarrow[\theta]{\tau} (v', ')$ where $v'(X) = U_{(v,)}(X)$ for $X \in dom(U_{(v,)})$ and $v'(X) = v(X)$ otherwise; $' = dom(U_{(v,)}) \cap \mathcal{S}$ and $\theta = U_{(v,)}$.

Let χ be the mapping that eliminates τ from the semantics of $M_{\mathcal{T}}$: $\chi((\alpha, \tau)) = \alpha$ and $\chi((\tau, \theta)) = \theta$. We extend χ to finite words: $\chi(uv) = \chi(u)\chi(v)$, and to languages of finite words: $\chi(L) = \{\chi(w) \mid w \in L\}$.

Theorem A.7. *$\mathcal{L}(\mathcal{T})$ equals $\chi(\mathcal{L}(M_{\mathcal{T}}))$.*

The EDT $\mathcal{T}$ will be consistent (Definition A.5) iff $M_{\mathcal{T}}$ has no cycle of unstable states.

A.3 Test case search using SAT

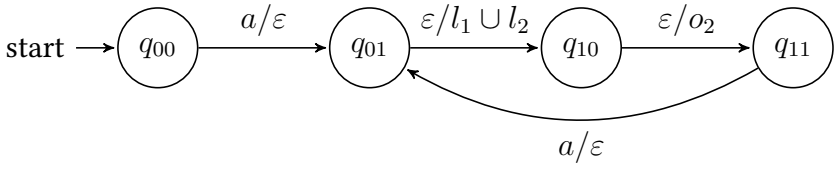
A test case for a row is a sequence of inputs that triggers the particular row. Given an EDT $\mathcal{T}$ and a row r_i of $\mathcal{T}$ as input, the *test generation problem* is to decide if there exists an execution $\pi \in \mathcal{L}(\mathcal{T})$ such that r_i is triggered in π.

Objective: Construct a propositional formula that holds true iff a given row in an EDT has a test case within a certain bound

A.3.1 Example

$\mathcal{I}_1$	in S_1	in S_2	out S_1	out S_2	O
a			l_1	l_2	
	l_1				
		l_2			o_2

Given the above EDT, its corresponding Mealy Machine looks like



where each of the states corresponds to the following configurations

$$\begin{aligned} q_{00} &= \bot, \\ q_{01} &= \{(a, \bot, \bot), \{a\}\}, \\ q_{10} &= \{(a, l_1, l_2), \{l_1, l_2\}\} \\ \text{and} \quad q_{11} &= \{(a, l_1, l_2), \emptyset\} \end{aligned}$$

In order to construct the propositional formula required, each state is represented by vectors $\{(v_{\mathcal{I}_1}, v_{S_1}, v_{S_2}), (\lambda_{\mathcal{I}_1}, \lambda_{S_1}, \lambda_{S_2}), (p_{r_1}, p_{r_2}, p_{r_3}, p_{\mathsf{st}})\}$ with each entry a propositional variable. Each v_X denotes whether the variable X is currently holding its default value or not. This representation works here since each variable has only one non-default value, but can be extended to a general case using binary encoding. Each λ_X on the other hand, denotes whether X was one of the variables to have changed value or not, on the transition from the previous state to the current one.

We also have additional variables to denote whether a state is *stable* or *unstable* (p_{st}), and whether or not a row r_i has matched (p_{r_i}).

At the initial state, we have

$$I(s) := (\neg v_X \wedge \neg \lambda_X) \forall X$$

Recalling the semantics of a row matching, we represent the transition relation. In this case,

$$v_{\mathcal{I}_1} \wedge \lambda_{\mathcal{I}_1} \Leftrightarrow p_{r_1} \tag{A.1}$$

$$p_{r_1} \Rightarrow (v'_{\mathcal{I}_1} \wedge v'_{S_1} \wedge v'_{S_2}) \wedge (\lambda'_{S_1} \wedge \lambda'_{S_2} \wedge \neg \lambda'_{\mathcal{I}_1}) \tag{A.2}$$

$$v_{S_1} \wedge \lambda_{S_1} \Leftrightarrow p_{r_2} \tag{A.3}$$

$$p_{r_2} \Rightarrow (v'_{S_1} \wedge \neg \lambda'_{S_1}) \wedge (v_{\mathcal{I}_1} \Leftrightarrow v'_{\mathcal{I}_1}) \tag{A.4}$$

$$v_{S_2} \wedge \lambda_{S_2} \Leftrightarrow p_{r_3} \tag{A.5}$$

$$p_{r_3} \Rightarrow (v'_{S_2} \wedge \neg\lambda'_{S_2}) \wedge (v_{\mathcal{I}_1} \Leftrightarrow v'_{\mathcal{I}_1}) \tag{A.6}$$

$$p_{r_1} \vee p_{r_2} \vee p_{r_3} \Leftrightarrow \neg p_{\mathsf{st}} \tag{A.7}$$

$$p_{\mathsf{st}} \Rightarrow v'_{\mathcal{I}_1} \wedge \lambda'_{\mathcal{I}_1} \wedge \neg\lambda'_{S_1} \wedge \neg\lambda'_{S_2} \wedge (v_{S_1} \Leftrightarrow v'_{S_1}) \wedge (v_{S_2} \Leftrightarrow v'_{S_2}) \tag{A.8}$$

$$T(s, s') := (1) \wedge (2) \wedge (3) \wedge (4) \wedge (5) \wedge (6) \wedge (7) \wedge (8)$$

Let us consider a case where we are looking for a test case for the third row of the EDT, with bound $k = 2$. For a path $s_0, \ldots, s_k$ to be a valid path of the Mealy Machine, the following formula must be satisfied

$$M_2 := I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2)$$

The path (with bound 2) is a test case for r_3 iff the following formula is satisfiable

$$\mathsf{Test}_2 := M_2 \wedge p_{r_3}(s_2)$$

In general, we can construct $\mathsf{Test_k}$ for any bound k and given row r_i of an EDT $\mathcal{T}$ (after associating a unique output corresponding to each row, with o_i for r_i).

A.3.2 General case

As in the example, each state is represented by vectors of propositional variables
$\{(v_{I_1}, \ldots, v_{I_p}, v_{S_1}, \ldots, v_{S_q}), (\lambda_{I_1}, \ldots, \lambda_{I_p}, \lambda_{S_1}, \ldots, \lambda_{S_q}), (p_{r_1}, \ldots, p_{r_n}, p_{\mathsf{st}})\}$

The initial state is constrained by the formula

$$I(s) := (\neg v_X \wedge \neg\lambda_X)\forall X$$

Formulae for EDT semantics:

$$\begin{aligned}
\bigvee_{i=1}^{n} p_{r_i} &\Leftrightarrow \neg p_{\mathsf{st}} \\
p_{\mathsf{st}} &\Rightarrow \bigwedge_{i=1}^{q}((v_{S_i} \Leftrightarrow v'_{S_i}) \wedge (\neg\lambda'_{S_i})) \wedge \bigvee_{j=1}^{p} \neg(v_{I_j} \Leftrightarrow v'_{I_j}) \wedge \bigwedge_{j=1}^{p}(\neg\lambda'_{I_j} \Leftrightarrow (v_{I_j} \Leftrightarrow v'_{I_j}))
\end{aligned}$$

$\forall r_i = (G_i \Rightarrow U_i)$,

$$\begin{aligned}
p_{r_i} &\Leftrightarrow (\bigwedge_{X \in G_i} v_X \wedge \bigvee_{X \in G_i} \lambda_X) \wedge (\bigwedge \neg p_{r'_i}) \forall i' < i, r'_i \# r_i \\
p_{r_i} &\Rightarrow \bigwedge_{X \in U_i} (v'_X \wedge \lambda'_X) \wedge \bigwedge_{X \notin U_i} ((v_X \Leftrightarrow v'_X) \wedge \neg\lambda'_X)
\end{aligned}$$

The first formula marks a state as stable or not depending on whether a row has matched. The second formula handles updates if the state is stable; it constrains local variables to remain unchanged while at least one of the input values must change and be marked as newly changed.

There are two formulae for each row r_i, the first describing the conditions for it to match and the second dealing with the update function. The first formula essentially checks if all the requirements for the row are true, with at least one of them becoming true on the last step, and that none of the higher priority rows in conflict with r_i are currently matched. The second formula simply updates all variables mentioned in the row output, marking them as newly changed, while others remain the same and are marked as unchanged.

The formula for the transition relation of the Mealy Machine, $T(s, s')$ is given by the conjunction of the formulae for the semantics.

We then define M_k constraining $s_0, \ldots, s_k$ to be a valid path in the Mealy Machine $M_{\mathcal{T}}$ corresponding to the EDT $\mathcal{T}$

$$M_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

The path of length k corresponds to a test case for r_i if the following formula is satisfiable

$$\mathsf{Test_k} := M_k \wedge p_{r_i}(s_k)$$