RWTH Aachen University

Bachelor Thesis in Computer Science

Minimizing Mealy Machines With Dependent Inputs

Author:
Mirela Mileva
Matr.-Nr.: 368610

Supervisor:
Prof. Dr. Thomas Noll

Co-Supervisor:
Prof. Dr. Joost-Pieter Katoen

The present work was submitted to the

Chair for Software Modeling and Verification

August 2019

Zentrales Prüfungsamt/Central Examination Office

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Name, Vorname/Last Name, First Name

Matrikelnummer (freiwillige Angabe)
Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel
I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Ort, Datum/City, Date

Unterschrift/Signature

*Nichtzutreffendes bitte streichen

*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Ort, Datum/City, Date

Unterschrift/Signature

Abstract

Finite State Machines (FSMs) are a mathematical model of computation used to represent a control execution flow. Minimizing them serves as an essential gateway for improving efficiency and effectiveness in their intricate design. Besides already formalized strategies that employ standard concepts of state equivalence based on equality of the input/output mappings computed, further optimization can be executed.

Exploiting the fact that FSMs might introduce a specific behavior in which inputs depend on each other, we can obtain more aggressive optimization techniques. To this aim, we introduce a binary relation which represents the input alphabet dependencies. By further considering present studies, the thesis discusses techniques on how this contributes to additional optimization and reduction of the state and transition space of a Finite State Machine. Moreover, we show the strengths of our approach, providing formal proofs, and demonstrating the methodology with particular examples. Also, we supply an algorithmic implementation by developing our optimization strategy into a ready to use program.

Acknowledgments

Throughout the writing of this thesis, I have received a great deal of support and assistance and want to take this opportunity to express my sincere gratitude for all the help.

Firstly, I want to thank my thesis advisor, Prof. Dr. Thomas Noll of the Chair for Software Modeling and Verification at RWTH Aachen University, who proposed the topic and the methodology in particular to me. I am sincerely thankful for the excellent cooperation, all of the opportunities I was given to conduct my research and all the support.

I gratefully acknowledge the help, technical assistance, and all the necessary facilities in my work provided by Markus Frohme at TU Dortmund.

Finally, I also want to thank my parents for the great encouragement, wise support, and loving attention, which enabled me to pursue a degree.

Contents

Chapter 1

Introduction

Finite State Machines (FSMs) are a method of profitably modeling hardware as well as software systems - aircraft, automobiles, robotics, and many others. They are applied to a broader class of problems to design sequential logic circuits. We commonly compare them to mechanical or electrical machines and refer to them as a simplified computing system for modeling many of the machines in our everyday life. FSMs are used in different domains to describe algorithms, sequential logic circuits, communication protocols, etc. They serve as a high-level abstraction of complex logic [Aufenkamp, 1958], [Aufenkamp and Hohn, 1957].

A possible application use-case of state machines are ATMs. Their interface consists of a series of buttons. To meet the system's needs, the software must be capable of responding to each pressed button. Nevertheless, handling each possible combination might result in an unmanageable number of cases for analysis. Another scenario is modeling microwave ovens where the system reacts to input through the door latch mechanism. To avoid flawed logic by testing various variables combinations, we can use a state machine representation of the system. It serves a high-level starting point for the design process.

A general model associated with a state machine is depicted in Figure 1.1. It consists of memory to store the current state in the machine and a combinatorial logic part containing two elements: the next state and output decoder. The next state decoder computes the next state, and the output decoder generates the corresponding output.

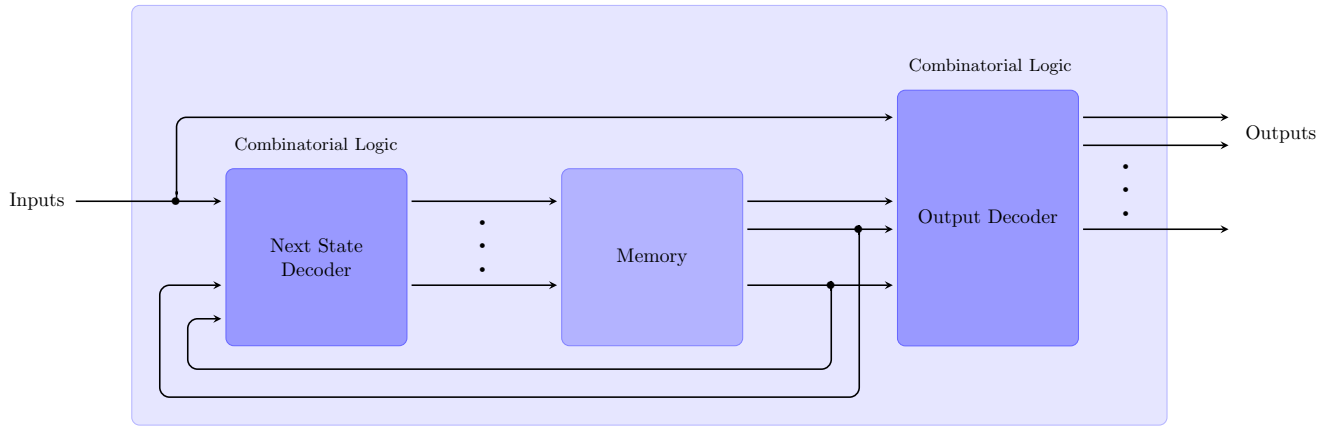


Figure 1.1: Block diagram of a Finite State Machinet (Redrawn from Cummings [2002]).

One class of Finite State Machines are Finite State Acceptors. They either accept or reject an input stream and produce a unique run of the machine for it. However, judging whether a program is valid is not always sufficient for particular applications. Imagine modeling an elevator. Merely "Accepting" or "Rejecting" the user's input would not give us the expected behavior. That is, it is necessary to get a response from the system as a reaction to the data, e.g., a voice message indicating the completed action or the performed action itself. To model systems that produce an output, we need a different class of Finite State Machines.

Such machines that accept an input and return an output are called sequential machines or transducers. The last name is due to the observation that the whole process of receiving input and turning it into an output serves as a translation procedure. While the finite state control and an input tape with a read head are still present, the accept/reject indicator is not required anymore. An output tape replaces it with a write head.

Designing such state machines is challenging. Due to the underlying complex logic, the respective model becomes large, i.e., the construction might contain many internal states and hundreds of transitions between them. Thus, techniques for reducing the state and the transition space play a crucial role in the designed systems.

Current synthesizing approaches to remove redundant (trace-equivalent) states can be further extended. A possibility for that is a potential merging of non-trace-equivalent pairs of states. Crucial is that the behavior of the state machine remains unchanged. Knowing that not every input stream which can be read by a state machine is allowed and considering a set of input sequences that can not be produced gives us extra freedom in the minimization approaches. Thus, further merging possibilities without violation of the behavior of the machine can be enabled.

This thesis sheds new light on the minimization of Finite State Transducers. Our experimental set up bears a close resemblance to previous investigated work, but we further exploit the fact that the model under design could have explicit restrictions. Therefore, particular input sequences never occur and thus can be neglected. We develop the methodology to minimize a state machine by not only removing redundant states but also by obtaining more aggressive minimization techniques that go beyond state merging. Thereby, we contribute to the achievement of complete results when performing the developed procedure step-by-step. As we have not found any counterexamples, we also expect that the algorithm enables finding an optimal solution, i.e., the Finite State Transducer with a minimal number of states.

The presented work is divided into six chapters. Chapter 2 gives a brief overview of Mealy Machines as a class of Finite State Transducers. Moreover, preliminary results are discussed, and essential concepts for the developed algorithm are introduced. The new methodology in the Mealy Machines Minimization is extensively described in Chapter 3. Also, the developed algorithm and diverse examples that demonstrate each proposed procedure are presented. Chapter 4 looks at the step-by-step application of the algorithm on two models. A discussion of the soundness and completeness of the minimizing approach falls inside the scope of Chapter 5. Moreover, the chapter outlines our assumptions that the discussed algorithm finds the best existing solution. Chapter 6 summarizes experimental results and finally, our conclusions are drawn in Chapter 7.

Chapter 2

Background

The following chapter introduces general concepts that are used in the proposed approach. Besides a formal definition of Finite State Machines, the chapter presents current minimization methods and outlines their limitations, which we further try to overcome.

Firstly, we concentrate on Finite State Transducers as an underlying concept of the presented methodology. They are modeled using two basic concepts - Mealy and Moore Machines that can be transformed into each other. Thus, the performed class of translations by both sequential networks is identical [Klimovich and Solov'ev, 2010]. While the output by Mealy Machines is a function of the present state and the current input, Moore Machines return an output once they enter a state.

To describe the problem addressed by this bachelor thesis, we focus further on Mealy Machines. Their design enables an immediately available output as soon as an input transition occurs and generally requires fewer states for synthesis than Moore Machines [Klimovich and Solov'ev, 2010].

2.1 Preliminary Definitions

A formal definition of Mealy Machines, which we adapt to our needs, has been proposed by Hopcroft et al. [2006]. The presented model serves as a vital component of the given approach in this thesis.

2.1.1 Mealy Machine

Definition 2.1. A Mealy Machine is a sex-tuple, $\mathfrak{M} = (Q, q_0, \Sigma, \Omega, \delta, \omega)$, consisting of

- a finite set of states Q,
- an initial state $q_0 \in Q$,
- a finite input alphabet Σ,
- a finite output alphabet Ω,
- a partial transition function $\delta : Q \times \Sigma \nrightarrow Q$, and
- a partial output function $\omega : Q \times \Sigma \nrightarrow \Omega$.

An additional requirement is that $\delta(q,a)$ is defined iff $\omega(q,a)$ is defined. Moreover, the latter two functions δ and ω are extended to

$$\delta^* : Q \times \Sigma^* \nrightarrow Q \text{ and } \omega^* : Q \times \Sigma^* \nrightarrow \Omega^*$$

with

$$\delta(q,\epsilon) := q,\ \delta^*(q,a.w) := \delta^*(\delta(q,a),w),$$
$$\omega(q,\epsilon) := \epsilon,\ \omega^*(q,a.w) := \omega(q,a) \cdot \omega^*(\delta(q,a),w).$$

The function computed by $\mathfrak{M}$, $\rho_{\mathfrak{M}} : \Sigma^* \nrightarrow \Omega^*$, is given by $\rho_{\mathfrak{M}}(w) := \omega^*(q_0,w)$.

There are different ways to specify the behavior of a Finite State Machine. One common approach which we refer to in the thesis to facilitate the understanding of the theoretical definitions is the usage of a graphical representation.

Figure 2.1 introduces an example for a Mealy Machine $\mathfrak{M} = (Q, q_0, \Sigma, \Omega, \delta, \omega)$ with $Q = \{q_0, q_1\}$, $\Sigma = \{a, b\}$, $\Omega = \{o_1, o_2\}$. The input and output for each transition are separated by a colon. As illustrated, $\delta(q_0,a) = q_1$, $\delta(q_0,b) = q_0$, $\delta(q_1,b) = q_1$, $\omega(q_0,a) = o_1$, $\omega(q_0,b) = o_2$, $\omega(q_1,b) = o_2$.

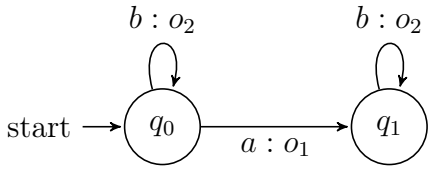


Figure 2.1: An example for a Mealy Machine.

2.1.2 Disabling Relation

Definition 2.2. Given an input alphabet Σ, a disabling relation is a binary relation of type $\triangleright \subseteq \Sigma \times \Sigma$ [1]. We will further describe all input streams over Σ and $\triangleright$ by $\Sigma^*_{\triangleright}$ and refer to them as *allowable*. The definition is formalized for a given Σ and $\triangleright$ as follows:

$$\Sigma^*_{\triangleright} = \{w = a_1...a_k \in \Sigma^* |\ \forall i \in \{1,...,k-1\} \text{ and } j \in \{i+1,...,k\}, a_i \not\triangleright a_j\}.$$

If an empty disabling relation is given, i.e., $\triangleright = \emptyset$, the unrestricted Σ^* has to be considered.

2.1.3 Mealy Machines Equivalence

Let Σ be a finite input alphabet and $\triangleright$ a disabling relation, $\triangleright \subseteq \Sigma \times \Sigma$.

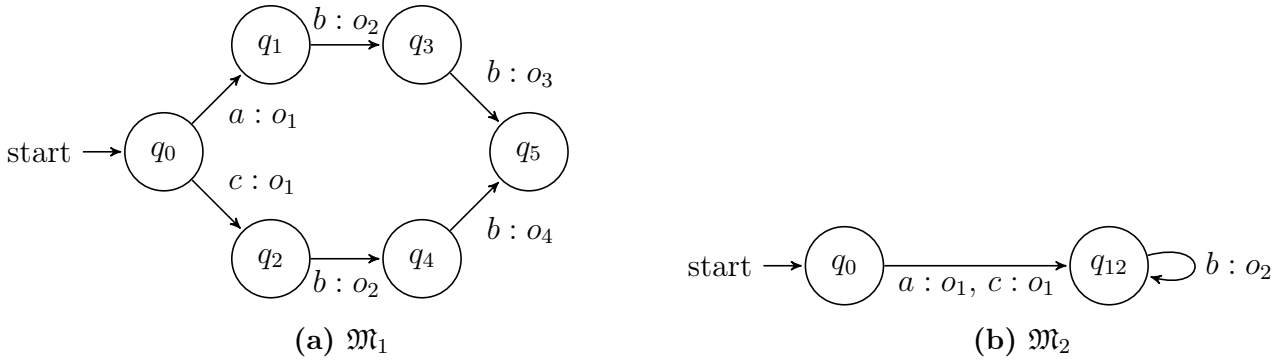
Definition 2.3. Two machines $\mathfrak{M}_1, \mathfrak{M}_2$ over Σ are called $\triangleright$-equivalent if $\rho_{\mathfrak{M}_1}(w) = \rho_{\mathfrak{M}_2}(w)$ for all $w \in \Sigma^*_{\triangleright}$. They are called trace-equivalent if they are $\emptyset$-equivalent, that is $\rho_{\mathfrak{M}_1}(w) = \rho_{\mathfrak{M}_2}(w)$ for all $w \in \Sigma^*$.

The notion can be generalized to $\triangleright$-equivalence of exemplary states q_1, q_2 by respectively considering $\omega^*(q_1,w)$ and $\omega^*(q_2,w)$ for all $w \in \Sigma^*_{\triangleright}$.

[1] $a \triangleright b \equiv$ an occurrence of a excludes a later occurrence of b.

In summary, two Mealy Machines are equivalent if there is no $w \in \Sigma_{\triangleright}^*$ separating them, i.e., no $w \in \Sigma_{\triangleright}^*$ for which both machines return different output.

The described concept is illustrated in Figure 2.2. Whereas both machines produce the same output for all $w \in \Sigma_{\triangleright}^*$ according to the disabling relation $\triangleright = \{(b, b)\}$, the input sequence $a \cdot b \cdot b \in \Sigma_{\triangleright}^*$ violates the equivalence if we consider the empty disabling relation $\triangleright' = \emptyset$, i.e., $\rho_{\mathfrak{M}_1}(a \cdot b \cdot b) \neq \rho_{\mathfrak{M}_2}(a \cdot b \cdot b)$.



(a) $\mathfrak{M}_1$ **(b)** $\mathfrak{M}_2$

Figure 2.2: A model demonstrating the effect of the properties of a considered disabling relation.

2.2 Foundations And Limitations Of Current Minimization Approaches

Previous work has extensively focused on state reduction of Finite State Transducers as a significant problem in the synthesis of sequential circuits [Hopcroft, 1971], [Gören and Ferguson, 2007], [Higuchi and Matsunaga, 1996]. The idea of removing trace-equivalent states has been broadly studied, and reliable approaches have been taken. The conducted studies provide partition algorithms where states are "divided" into equivalent blocks according to the input/output mappings computed. Each partition procedure is executed until no further refinement is possible. Consequently, later on, each partition contains only trace-equivalent states.

The synthesis of Mealy Machines is also involved in reliability engineering as an essential application domain. Redundancy concepts play a vital role in the design of safety-critical systems such as aerospace systems [Noll et al., 2017]. Deciding when to activate which redundancy and which component should be replaced in case of failures, wrong sensor readings, etc. is challenging. Noll et al. [2017] present the application of Non-Deterministic Dynamic Fault Trees for modeling possible failure behavior of such systems. Their synthesis is further used to model Mealy Machines from which appropriate recovery strategies can be derived. In the presented scenario inputs describe sets of basic failure events of the underlying fault tree, and outputs are sequences of recovery actions that claim spare components.

The described approach introduces concepts for reducing the state space of a Mealy Machine to maximize the overall system reliability over time. The presented methodology enables, in addition to the usage of the standard concept of trace-equivalence between states, further optimization. This is accomplished by exploiting the fact that (standard) fault trees exhibit a monotonic behavior, i.e., each basic event could occur only once. Thus, the fundamental concept described in the paper is

to identify states that may have transitions with disagreeing outputs. An additional constraint is that at least one of the transitions is disabled for either one of the two states. The concept of states orthogonality is defined as a primary concept in the presented approach [Noll et al., 2017].

The definition is formalized with the following function. It returns for each state $q \in Q$ the set of *disabled* input symbols according to $\triangleright$, i.e., the collection of input symbols which cannot occur after visiting q.

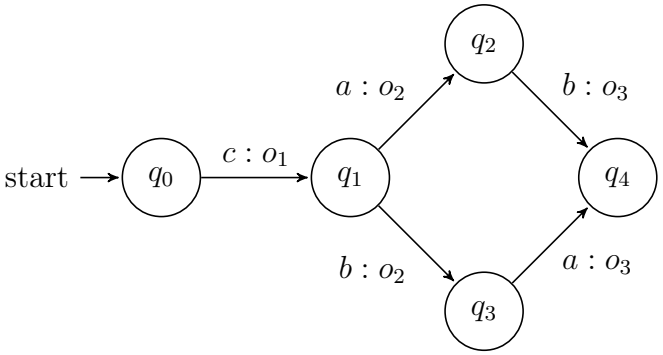
$$\triangle : Q \to 2^{\Sigma} : q \mapsto \{b \in \Sigma \mid \forall w = a_1 ... a_k \in \Sigma^* : \delta^*(q_0, w) = q \Rightarrow \exists i \in \{1, ..., k\} : a_i \triangleright b\}.$$

2.2.1 Orthogonal States

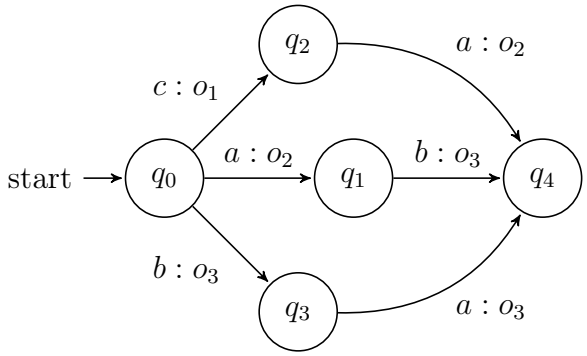
Two states q_1 and $q_2 \in Q$ with $q_1 \neq q_2$ are considered as orthogonal with respect to $a \in \Sigma$ if it applies that

$$a \in \triangle(q_1) \cup \triangle(q_2).$$

To demonstrate the terminology, we refer to Figure 2.3. Let $\Sigma = \{a, b, c\}$ and $\mathfrak{M}_1, \mathfrak{M}_2$ be as depicted.



(a) $\mathfrak{M}_1$



(b) $\mathfrak{M}_2$

Figure 2.3: Orthogonal states in a Mealy Machine.

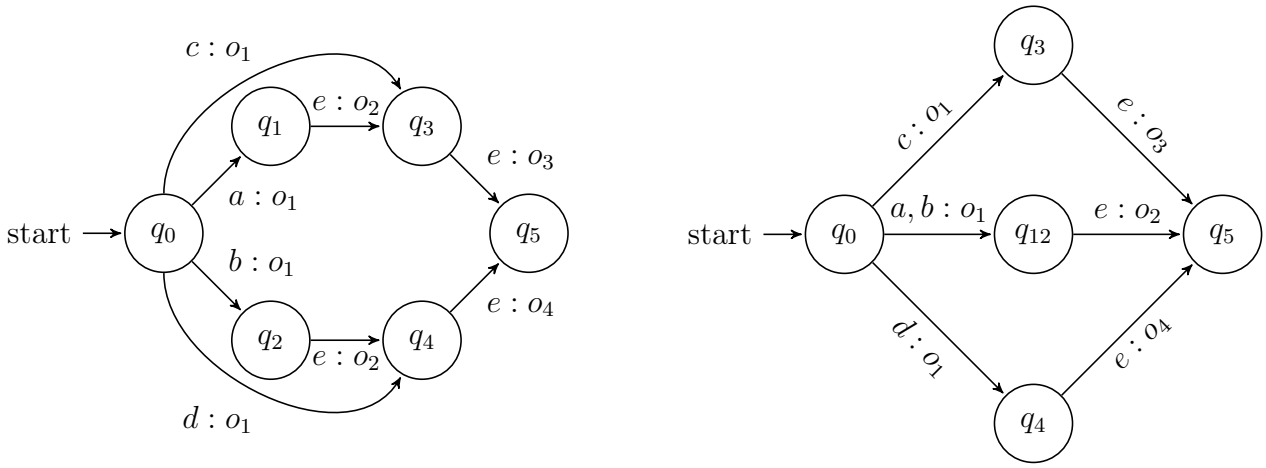
In the given Mealy Machine $\mathfrak{M}_1$ the states q_2 and q_3 are orthogonal w.r.t a, b, c because it holds that $\triangle(q_2) = \{c, a\}$, $\triangle(q_3) = \{c, b\}$ and thus $\triangle(q_2) \cup \triangle(q_3) = \Sigma$. However, in $\mathfrak{M}_2$ the states q_2 and q_3 are not orthogonal w.r.t. a because $a \notin \triangle(q_2) \cup \triangle(q_3)$, but orthogonal w.r.t. b and c.

2.2.2 Merging Rule

The key observation made by Noll et al. [2017] is that two states q_1 and q_2 can be merged if, for every $a \in \Sigma$,

- q_1 and q_2 are orthogonal with respect to a or
- $\omega(q_1, a) = w(q_2, a)$ and $\delta(q_1, a)$ and $\delta(q_2, a)$ can be recursively merged.

In [Noll et al., 2017] a soundness proof is given by showing that whenever two states can be merged, they are $\triangleright$-equivalent. Unfortunately, the reverse implication is generally false, i.e. merging of states is incomplete w.r.t. $\triangleright$-equivalence. To illustrate this limitation we consider the example Mealy Machine shown in Figure 2.4 and assume $\Sigma = \{a, b, c, d, e\}$, $\Omega = \{o_1, o_2, o_3, o_4\}$, $\triangleright$ reflexive.



(a) $\mathfrak{M}_1$ **(b)** $\mathfrak{M}_2$

Figure 2.4: Equivalent Mealy Machines $\mathfrak{M}_1$ and $\mathfrak{M}_2$.

According to [Noll et al., 2017], states q_3 and q_4 are not $\triangleright$-equivalent in $\mathfrak{M}_1$ as they provide different outputs on input e (o_3 and o_4, respectively). Consequently, q_1 and q_2 are not $\triangleright$-equivalent either because their e-transitions lead to states that are not $\triangleright$-equivalent. However, as shown in Figure 2.4b, there is a $\triangleright$-equivalent Mealy Machine $\mathfrak{M}_2$ with fewer states that can not be acquired through the state merging approach presented in the paper.

Besides incompleteness, the underlying concept of the given methodology in [Noll et al., 2017] is fault trees. Their monotonic behavior serves as a reflexive disabling relation and other binary relations with custom properties are not taken into account.

For our Minimization Algorithm, we used a variation of the described procedure in [Noll et al., 2017]. Specifically, we introduce a more general minimization framework that preserves completeness and handles any given disabling relation.

Chapter 3

Main Contributions

The main topic of this section is the developed algorithm. We define all functions in pseudo-code. Please refer to Appendix A for the actual implementation.

Since previous work has solely focused on standard partition refinement algorithms [Aufenkamp, 1958], [Aufenkamp and Hohn, 1957] or minimization through a reflexive relation [Noll et al., 2017], the following sections give an extensive look into a pioneering algorithm. It handles any Mealy Machine and disabling relation with custom properties.

3.1 Embedding Concepts Definitions And Explanations

Throughout this thesis, we use the following pre-definitions to improve the readability of subsequent procedures. For a given state $q \in Q$ in a Mealy Machine $\mathfrak{M} = (Q, q_0, \Sigma, \Omega, \delta, \omega)$ we define a set of incoming and outgoing transitions. The given function definitions formalize these notions

$$incomingTransitions(q) = \{(q', a) \mid \exists a \in \Sigma, \exists q' \in Q : \delta(q', a) = q\},$$

$$outgoingTransitions(q) = \{(a, q') \mid \exists a \in \Sigma, \exists q' \in Q : \delta(q, a) = q'\}.$$

3.2 History-Based State Output Equivalence

The subject, which we shall refer to as History-Based State Output Equivalence is essential for the presented algorithm in the thesis. More precisely, if two states are recognized as History-Based Output Equivalent, it is in general safe to merge them, i.e., the output behavior of the Mealy Machine will not change.

Informally, we identify two states as History-Based Output Equivalent if they produce the same output for each *allowable* input sequence for both states according to a given disabling relation. This notion is formalized with the following definition:

Definition 3.1. $q_1 \approx_H q_2$ iff $\omega^*(q_1, w) = \omega^*(q_2, w)\ \forall w \in (\Sigma \setminus (\triangle(q_1) \cup \triangle(q_2)))^*_{\triangleright}$.

The decision whether $q_1 \approx_H q_2$ for $q_1, q_2 \in Q$ is made with Algorithm 1. To achieve soundness, we consider a restriction of δ and ω to only *allowable* inputs for

each state. The termination of the procedure in case of cyclic Mealy Machines is ensured through a set of considered pairs. In it, each already handled pair of states is put.

Algorithm 1 History-Based State Output Equivalence

Input: $\mathfrak{M} = (Q, q_0, \Sigma, \Omega, \delta|_{\triangle}, \omega|_{\triangle})$, a disabling relation $\triangleright$, consideredPairs and $q_1, q_2 \in Q$

Output: $true \equiv q_1 \approx_H q_2$ or $false \equiv q_1 \not\approx_H q_2$

procedure hEquivalence
  if $(q_1, q_2) \notin$ consideredPairs **then**
    consideredPairs.add((q_1, q_2));
    for each $a \in \Sigma$ with $\exists q' : (a, q') \in outgoingTransitions(q_1)$ **do**
      if $\forall q'' : (a, q'') \notin outgoingTransitions(q_2)$ AND $a \notin \triangle(q_2)$ **then**
        return false;
    for each $a \in \Sigma$ with $\exists q' : (a, q') \in outgoingTransitions(q_2)$ **do**
      if $\forall q'' : (a, q'') \notin outgoingTransitions(q_1)$ AND $a \notin \triangle(q_1)$ **then**
        return false;
    for each $a \in \Sigma$ with $\exists q', \exists q'' : (a, q') \in outgoingTransitions(q_1)$, $(a, q'') \in outgoingTransitions(q_2)$ **do**
      if $\omega(q_1, a) \neq \omega(q_2, a)$ **then**
        return false;
      else hEquivalence($\mathfrak{M}, \triangleright$, consideredPairs, $\delta(q_1, a), \delta(q_2, a)$)
  return true;

Figure 3.1 serves as an example for History-Based Output Equivalence. It demonstrates the role of the disabling relations entries. In case of $\triangleright = \{(a, a), (b, b), (c, c)\}$, Algorithm 1 determines that $b \in \triangle(q_3)$, $a \in \triangle(q_2)$. Also, as the c-transitions from both states output the same symbol o_3 and lead to trivially History-Based Output Equivalent states, Algorithm 1 concludes that $q_1 \approx_H q_2$. However, if the disabling relation is not reflexive q_2 and q_3 are not History-Based Output Equivalent. Thus, merging them would change the behavior of the Mealy Machine. For example, the input stream $b \cdot b^+$ would be accepted in consequence.

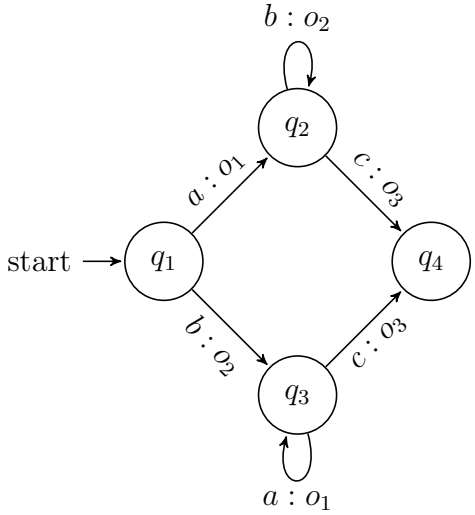


Figure 3.1: Mealy Machine $\mathfrak{M}$. In case of $\triangleright = \{(a,a),(b,b),(c,c)\}$, $q_1 \approx_H q_2$. If the disabling relation $\triangleright = \emptyset$, then $q_1 \not\approx_H q_2$.

3.3 Auxiliary Algorithms

The following auxiliary procedures are an integral part of the Minimizing Algorithm. We presume that creation and deletion operations directly modify the considered Mealy Machine.

3.3.1 Elimination Of Isolated States

During the minimization procedure, individual states may be no longer reachable from the initial state. Thus, these states, called further *isolated*, can be extracted with the following algorithm presented in Figure 3.2.

Algorithm 2 Elimination Of Isolated States

Input: $\mathfrak{M} = (Q, q_0, \Sigma, \Omega, \delta, \omega)$
Output: $\mathfrak{M} = (Q', q_0, \Sigma, \Omega, \delta', \omega')$ without *isolated* states

procedure DELETEISOLATED
  for each $q \in Q$ **do**
    if $\neg\exists w \in \Sigma^* : \delta(q_0, w) = q$ **then**
      delete all $\delta(q, a) = q'$ and $\omega(q, a) = o$;
      delete q;
  return $\mathfrak{M}$;

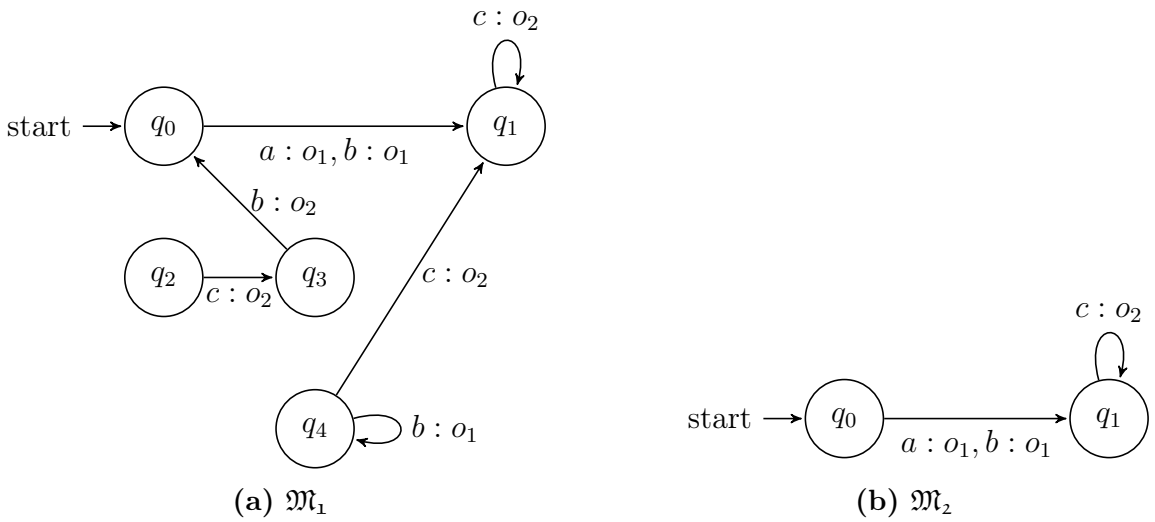


Figure 3.2: Mealy Machine before ($\mathfrak{M}_1$) and after ($\mathfrak{M}_2$) elimination of the *isolated* states $q_2,\ q_3,\ q_4$.

3.3.2 Elimination Of Disabled Transitions

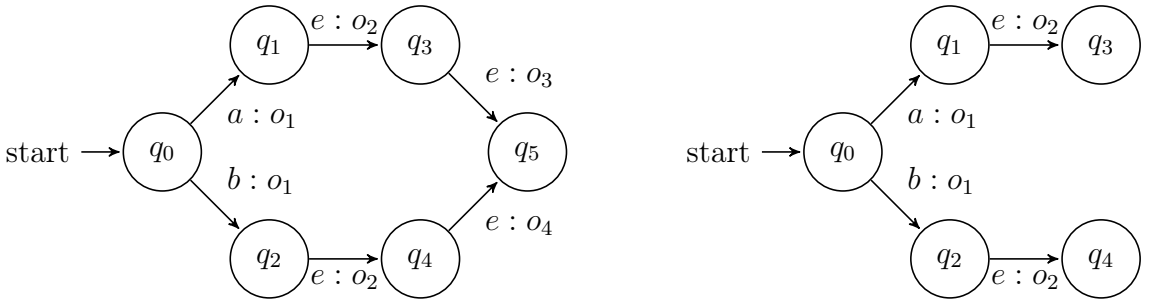
A *disabled* transition will further denote a transition from a state q, for which a transition input a is *disabled* under the given disabling relation, i.e., $\exists q' \in Q$: $(a, q') \in outgoingTransitions(q)$ and $a \in \triangle(q)$. Otherwise, the transition is *enabled*. To ensure that an observed Mealy Machine contains only *enabled* transitions, we use Algorithm 3.

Algorithm 3 Elimination Of Disabled Transitions

Input: $\mathfrak{M} = (Q, q_0, \Sigma, \Omega, \delta, \omega)$ and a disabling relation $\triangleright$
Output: $\mathfrak{M} = (Q, q_0, \Sigma, \Omega, \delta', \omega')$ without *disabled* transitions

```
procedure eliminateDisabled
    while {no changes} do
        for each q ∈ Q do
            for each a ∈ Σ, q' ∈ Q with δ(q, a) = q' do
                if a ∈ △(q) then
                    delete δ(q, a) and ω(q, a);
    return 𝔐;
```

The whole procedure is demonstrated in Figure 3.3 for an example Mealy Machine $\mathfrak{M}$ and a reflexive disabling relation $\triangleright = \{(a, a), (b, b), (e, e)\}$. The input e cannot be read in states q_3, q_4 according to $\triangleright$. Thus, elimination of transitions is possible.



(a) $\mathfrak{M}_1$ **(b)** $\mathfrak{M}_2$

Figure 3.3: Mealy Machine before ($\mathfrak{M}_1$) and after ($\mathfrak{M}_2$) elimination of the *disabled* transitions $e : o_3$ and $e : o_4$.

3.3.3 State Splitting

A fundamental concept in the presented Minimization Algorithm to obtain an extended merging rule for Mealy Machines with dependent inputs is State Splitting.

Intuitively, for a considered state in a Mealy Machine, we split it into n new states, where n is the number of different transitions to this state, concerning δ and ω. Then, we copy all outgoing transitions for each of the freshly created states. In case a state was already created for an analyzed input/output pair, the following transitions are redirected. To ensure termination, we ignore self-loops.

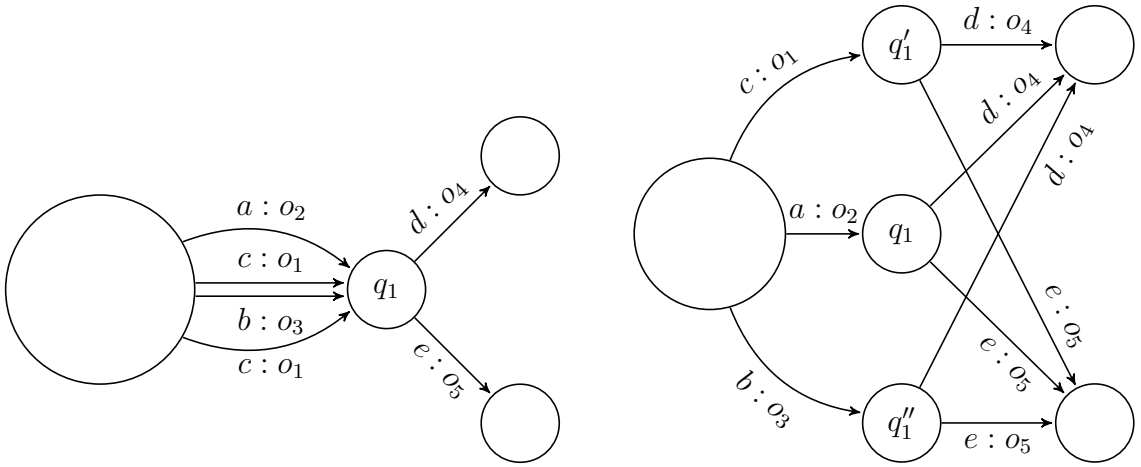
Algorithm 4 State Splitting

Input: $\mathfrak{M} = (Q, q_0, \Sigma, \Omega, \delta, \omega)$ and a state $q \in Q$
Output: $\mathfrak{M} = (Q', q_0, \Sigma, \Omega, \delta', \omega')$ after performing state splitting on q, resultingStates

procedure SPLITSTATE
  Set resultingStates = $\{q\}$;
  List consideredTransitions;
  if $|incomingTransitions(q)| > 1$ **then**
    for each $(q', a) \in incomingTransitions(q)$ with $\omega(q', a) = o$ and $q \neq q'$ **do**
      if $(a, o) \notin$ consideredTransitions **then**
        consideredTransitions.add((a, o));
        delete $\delta(q', a) = q$;
        create q_{new} with $\delta(q', a) = q_{new}$
        resultingStates.add(q_{new});
        for each $(b, q'') \in outgoingTransitions(q)$ with $\omega(q, b) = o'$ **do**
          create $\delta(q_{new}, b) = q''$;
          create $\omega(q_{new}, b) = o'$;
      else
        create $\delta(q', a) = q_{old}$ such that $\exists q'' \in Q : \delta(q'', a) = q_{old}$ with $\omega(q'', a) = o$, $q_{old} \in$ resultingStates;

for each $(b, q'') \in outgoingTransitions(q)$ with $\omega(q, b) = o'$ **do**
delete $\delta(q, b) = q''$;
delete $\omega(q, b) = o'$;
delete q;
return $\{\mathfrak{M}$, resultingStates$\}$;

After introducing the algorithm formally, we now refer to Figure 3.4. To demonstrate the procedure, we perform State Splitting on q_1 in $\mathfrak{M}_1$ with $\Sigma = \{a, b, c, d, e\}$ and $\Omega = \{o_1, o_2, o_3, o_4, o_5\}$. For each unique incoming transition to q_1 a new state is created, and each outgoing transition is copied for each freshly created state.



(a) $\mathfrak{M}_1$ **(b)** $\mathfrak{M}_2$

Figure 3.4: A demonstration of State Splitting on q_1 in $\mathfrak{M}_1$. After the execution of the procedure, $\mathfrak{M}_2$ is obtained.

3.3.4 State Merging

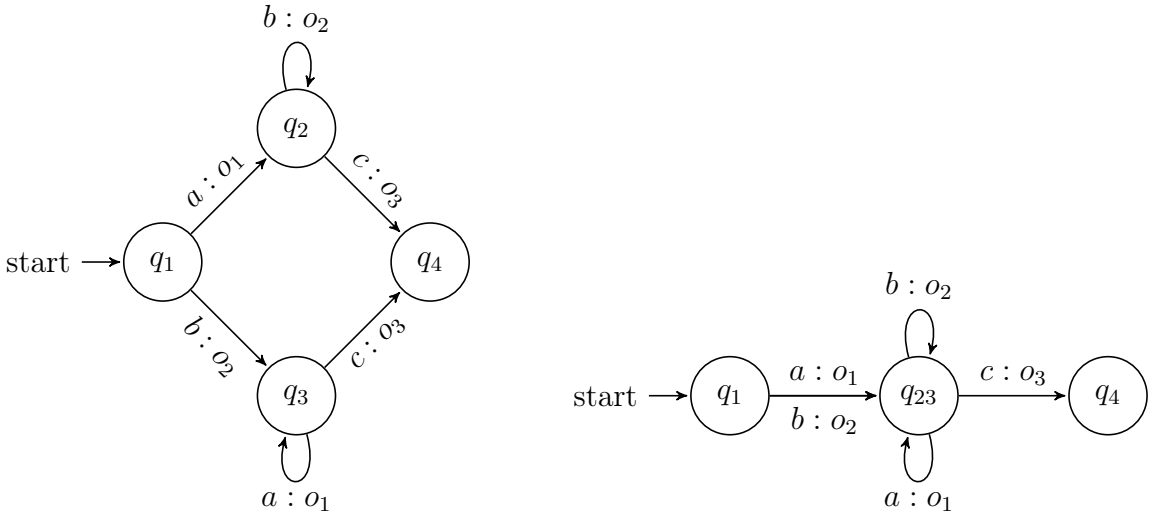
When two states are determined to be History-Based Output Equivalent, we perform a merging as described in Algorithm 5 and depicted in Figure 3.5.

Algorithm 5 State Merging

Input: $\mathfrak{M} = (Q, q_0, \Sigma, \Omega, \delta, \omega)$, History-Based Output Equivalent states $q_1, q_2 \in Q$ and a disabling relation $\triangleright$
Output: $\mathfrak{M} = (Q', q_0, \Sigma, \Omega, \delta', \omega')$ after merging q_1 into q_2

procedure MERGE
for each $(q', a) \in incomingTransitions(q_1)$ **do**
delete $\delta(q', a) = q_1$;
create $\delta(q', a) = q_2$;
$\triangleright$ do nothing if a transition already exists

for each $(a, q') \in outgoingTransitions(q_1)$ with $\omega(q_1, a) = o$ and $a \notin \triangle(q_1)$ **do**
delete $\delta(q_1, a) = q'$;
delete $\omega(q_1, a) = o$;
create $\delta(q_2, a) = q'$;
create $\omega(q_2, a) = o$;
delete q_1;
return $\mathfrak{M}$;



(a) $\mathfrak{M}_1$ **(b)** $\mathfrak{M}_2$

Figure 3.5: Mealy Machine before ($\mathfrak{M}_1$) and after ($\mathfrak{M}_2$) merging q_2 into q_3.

3.3.5 Merging The Sink States To Predecessors

Algorithm 6 is used to merge the Sink States to their predecessors. As the name suggests, a Sink State is a state without outgoing transitions. However, performing the procedure is sound only if the disabling relation $\triangleright$ is reflexive. In either case, we will violate the equivalence rule as newly created loop transitions can be executed more than once, which generally has not been possible before the modifications.

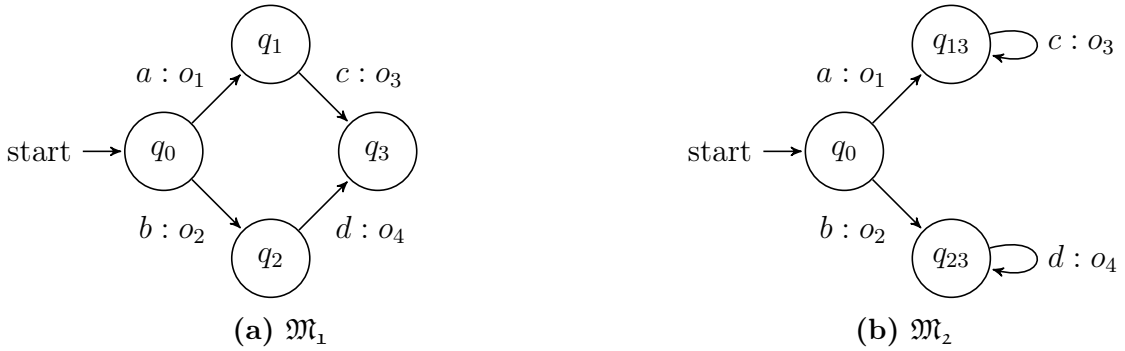
Algorithm 6 Merging Sink States to Predecessors

Input: $\mathfrak{M} = (Q, q_0, \Sigma, \Omega, \delta, \omega)$ and a disabling relation $\triangleright$
Output: $\mathfrak{M} = (Q', q_0, \Sigma, \Omega, \delta', \omega)$ after merging all Sink States to their predecessors

procedure MERGESINKSTATES
if {$\triangleright$ reflexive} **then**
for each $q \in Q \setminus \{q_0\}$ **do**
if $outgoingTransitions(q) = \emptyset$ **then**

```
            if (∃(q', a) ∈ incomingTransitions(q) : |outgoingTransitions(q')|>1)
then
                return 𝔐;
            else
                for each (q', a) ∈ incomingTransitions(q) do
                    delete δ(q', a) = q
                    create δ(q', a) = q'
            delete q;
    return 𝔐;
```

To illustrate the procedure, we consider the example in Figure 3.6. As q_3 has no otgoing transitions, it is a Sink State in the Mealy Machine $\mathfrak{M}_1$. If we consider the reflexive disabling relation $\triangleright = \{(a,a),(b,b),(c,c),(d,d)\}$, it is safe to merge q_3 to its predecessors.



(a) $\mathfrak{M}_1$ **(b)** $\mathfrak{M}_2$

Figure 3.6: Mealy Machine before ($\mathfrak{M}_1$) and after ($\mathfrak{M}_2$) merging the Sink State q_3 to its predecessors.

3.4 Minimization Algorithm

After completing the needed definitions and presenting all auxiliary algorithms, we are now able to introduce the minimization algorithm. We start by describing the idea beyond informally.

For optimality reasons, the procedure is accomplished iteratively. Thus, while we can perform a change and produce a not already handled Mealy Machine, we execute the algorithm. So it is guaranteed that all possible resulting Mealy Machines will be considered. To choose the best result, i.e., the Mealy Machine with the smallest number of states, all results from each run of the algorithm are saved. Note that not all results contain fewer states than the initial Mealy Machine.

The algorithm starts by calculating a traversing order in which pairs of states are considered. To obtain efficiency, we exclude pairs containing the same elements and commutative pairs. In parallel, a MINIMIZE-procedure is executed. By ensuring in advance that only *enabled* transitions are present the algorithm checks for each pair of states in the computed traversing order whether they are History-Based Output Equivalent as described in Subsection 3.2. If so, both states are merged, and the algorithm proceeds to execute with the modified Mealy Machine. In either case, both states are split according to Algorithm 4. To inspect whether the split can

contribute to a better final result, we check History-Based Output Equivalence for each pair of states from the resulting sets after splitting by extracting beforehand the *disabled* transitions. If such a pair is found, the states in it are merged. Otherwise, the split is reverted, and the algorithm goes back to the situation before splitting.

To improve efficiency, after each minimization iteration, the procedure executes the Algorithms for deleting *isolated* states 2, removing *disabled* transitions 3 and merging Sink States to predecessors 6. Then, it looks at the obtained Mealy Machine and checks whether it was considered before. If not, it adds it to a set to look at it later for further optimization. When the set of Mealy Machines cannot be altered anymore, the procedure CHOOSEMINIMALMEALY is called to find the best solution, returning the Mealy Machine with the fewest number of states.

For a more fundamental understanding of the given algorithm, we presume that after each merging/splitting, the traversing order is adjusted to the obtained changes. Thus, each appearance of a merged state is replaced with the state after merging. In either case, the tuple which contains a split state is deleted and therefore considered only in the following iterations.

Algorithm 7 Mealy Machine Minimization

Input: A $\mathfrak{M} = (Q, q_0, \Sigma, \Omega, \delta, \omega)$ and a disabling relation $\triangleright$
Output: A minimized $\mathfrak{M}' = (Q', q_0', \Sigma, \Omega, \delta', \omega')$ with $\rho_{\mathfrak{M}}(w) = \rho_{\mathfrak{M}'}(w)$ $\forall w \in \Sigma_{\triangleright}^*$

```
procedure OPTIMALMINIMIZATION
    List checkedAutomata;
    𝔐_curr = 𝔐;
    do
        var traversingOrder := CALCULATEPAIRS(Q);
        var 𝔐_min := MINIMIZE(𝔐_curr, ▷, traversingOrder)
        𝔐_min = ELIMINATEDISABLED(𝔐_min);
        𝔐_min = DELETEISOLATED(𝔐_min);
        𝔐_min = MERGESINKSTATES(𝔐_min);
        if !(checkedAutomata.contains(𝔐_min)) then
            checkedAutomata.add(𝔐_min);
            𝔐_curr = 𝔐_min;
        else 𝔐_curr = ∅
    while 𝔐_curr ≠ ∅
    return CHOOSEMINIMALMEALY(checkedAutomata, ▷);
```

Input: A $\mathfrak{M} = (Q, q_0, \Sigma, \Omega, \delta, \omega)$, a disabling relation $\triangleright$ and a traversingOrder
Output: $\mathfrak{M}' = (Q', q_0, \Sigma, \Omega, \delta', \omega')$

```
procedure MINIMIZE
    𝔐 = ELIMINATEDISABLED(𝔐);
    for each (q1, q2) ∈ traversingOrder do
        if (q1 ≈H q2) then
            𝔐 = MERGE(𝔐, q1, q2, ▷);
        else
            var 𝔐init := 𝔐;
            if q1 ≠ q0 then
                var splitResultFirst := SPLITSTATE(𝔐, q1);
                var resultingStatesFirst :=SPLITRESULTFIRST.GETRESULTINGSTATES();
                𝔐 = SPLITRESULTFIRST.GETMEALY();
            if q2 ≠ q0 then
                var splitResultSecond := SPLITSTATE(𝔐, q2);
                var resultingStatesSecond :=SPLITRESULTSECOND.GETRESULTINGSTATES();
                𝔐 = SPLITRESULTSECOND.GETMEALY();
            var crossProduct := resultingStatesFirst × resultingStatesSecond;
            var splitSuccessful := false;
            𝔐 = ELIMINATEDISABLED(𝔐);
            for each (p1, p2) ∈ crossProduct do
                if (p1 ≈H p2) then
                    𝔐 = MERGE(𝔐, p1, p2, ▷);
                    splitSuccessful = true;
            if (!splitSuccessful) then
                return 𝔐init;
            else
                return 𝔐;
```

Input: A finite set of states Q
Output: $Q \times Q$ without pairs with same elements and reversed pairs

```
procedure CALCULATEPAIRS
    Set pairs := ∅;
    for each q1 ∈ Q do
        for each q2 ∈ Q do
            if (q1 ≠ q2 AND !(pairs.contains(q1, q2))) then
                pairs.add(q1, q2);
    return pairs;
```

Input: A set of Mealy Machines checkedAutomata and a disabling relation $\triangleright$
Output: $\mathfrak{M}_{min} \in$ checkedAutomata with the minimal number of states

procedure CHOOSEMINIMALMEALY
    var $\mathfrak{M}_{min}$;
    var minStates := $+\infty$;
    for each $\mathfrak{M} \in$ checkedAutomata **do**
        if ($\mathfrak{M}$.SIZE() $\leqslant$ minStates) **then**
            minStates = $\mathfrak{M}$.SIZE();
            $\mathfrak{M}_{min} = \mathfrak{M}$;
    return $\mathfrak{M}_{min}$;

Chapter 4

Minimization Algorithm Examples

In the following chapter, we give illustrative examples to demonstrate Algorithm 7. For the Mealy Machine $\mathfrak{M}_{\mathfrak{first}}$ in Section 4.1 no State Splitting is needed to obtain minimal results. However, this is not the case for the Mealy Machine in Section 4.2.

We omit giving the Mealy Machine after each ELIMINATEDISABLED-procedure. Instead, *disabled* transitions are ignored.

4.1 Minimization Without Splitting

Initially, we consider the Mealy Machine $\mathfrak{M}_{first}$ given in Figure 4.1 with the irreflexive disabling relation $\triangleright = \{(a, e), (b, d)\}$. An entry point is the calculation of a traversing order. For $\mathfrak{M}_{first}$ this would be $\{(q_0, q_1), (q_0, q_2), (q_0, q_3), (q_1, q_2), (q_1, q_3), (q_2, q_3)\}$. With this traversing order we call the procedure MINIMISE to obtain potential minimization.

Beforehand, we think about the $\triangle(q_i)$-sets, $i \in [0, 3]$ to make it easier to understand the follow-up process. It is clear that $\triangle(q_0) = \emptyset$, $\triangle(q_1) = \{e\}$, $\triangle(q_2) = \{d\}$, $\triangle(q_3) = \emptyset$.

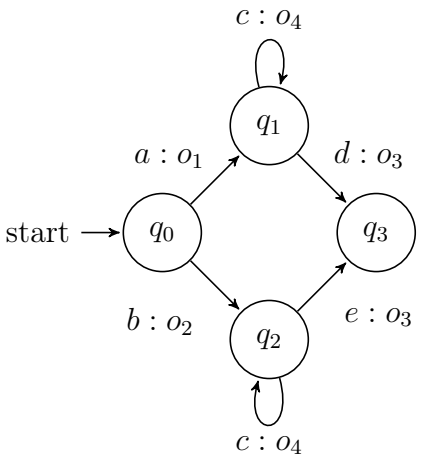


Figure 4.1: Initial Mealy Machine $\mathfrak{M}_{first}$.

Because $outgoingTransitions(q_0)$ contains the input set $\{a, b\}$ and $\{a, b\} \subsetneq \triangle(q_1)$, $\triangle(q_2)$, $\triangle(q_3)$, it follows that $q_0 \not\approx_H q_1, q_2, q_3$. Also, as each of the states q_1, q_2, q_3 has only one not self-loop incoming transition, splitting is not possible. Thus, the algorithm proceeds with the pair (q_1, q_2).
As the $c-$transitions from both states are self-loops, $d \in \triangle(q_2)$ and $e \in \triangle(q_1)$, $q_1 \approx_H q_2$. Thus, we can merge both states obtaining the Mealy Machine in Figure 4.2 and adjusting the remaining traversing order to $\{(q_{12}, q_3), (q_{12}, q_3)\}$.

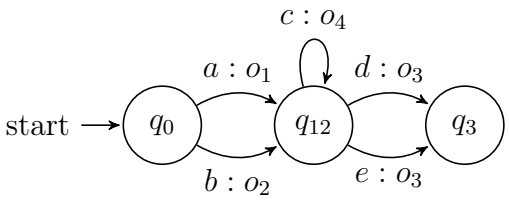


Figure 4.2: Mealy Machine $\mathfrak{M}_{first_1}$.

$\triangle(q_3) = \emptyset$, q_3 is a Sink State and $outgoingTransitions(q_{12})$ contains the input set $\{c, d, e\}$. Thus, $q_{12} \not\approx_H q_3$. Therefore the algorithm splits q_{12} and q_3. Obviously, this will result in the already considered Mealy Machine $\mathfrak{M}_{first}$ for which neither $q_1 \approx_H q_3$, nor $q_2 \approx_H q_3$. Hence, the split will be reverted and the Mealy Machine $\mathfrak{M}_{first_1}$ returned as final result from this iteration. This is the only possible split and it obtains the initial Mealy Machine which has already been considered. Moreover, as there are no *disabled* transitions, *isolated* states and $\triangleright$ is not reflexive, the iterations are terminated.

The result in Figure 4.3 is returned as the minimal and the only one obtained Mealy Machine $\mathfrak{M}_{first_1}$ with $\rho_{\mathfrak{M}_{first}}(w) = \rho_{\mathfrak{M}_{first_1}}(w)$ for all $w \in \Sigma^*_{\triangleright}$.

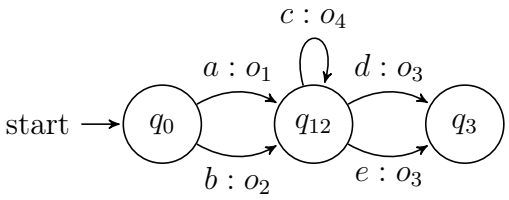


Figure 4.3: Minimal Mealy Machine $\mathfrak{M}_{first_1}$ with $\rho_{\mathfrak{M}_{first}}(w) = \rho_{\mathfrak{M}_{first_1}}(w)$ for all $w \in \Sigma^*_{\triangleright}$.

4.2 Minimization With Splitting

The second example would be the already handled Mealy Machine in Section 2.2.2 where the limitations of preliminary studies were discussed. As we have seen, the minimization approach conducted by Noll et al. [2017] is generally incomplete concerning Mealy Machines equivalence. Thus, we convince in the benefits of Algorithm 7, executing it with the example Mealy Machine from Section 2.2.2 step-by-step. Furthermore, we consider the reflexive disabling relation $\triangleright = \{(a, a), (b, b), (c, c), (d, d), (e, e)\}$.

For $\mathfrak{M}_{second}$ the traversing order would be $\{(q_0, q_1), (q_0, q_2), (q_0, q_3), (q_0, q_4), (q_0, q_5), (q_1, q_2), (q_1, q_3), (q_1, q_4), (q_1, q_5), (q_2, q_3), (q_2, q_4), (q_2, q_5), (q_3, q_4), (q_3, q_5), (q_4, q_5)\}$. We proceed further with the procedure MINIMISE.

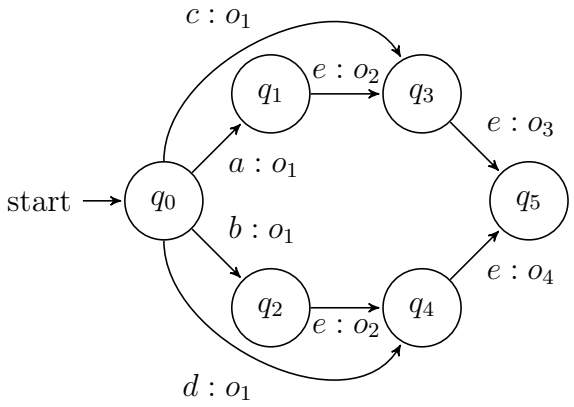


Figure 4.4: Initial Mealy Machine $\mathfrak{M}_{second}$.

Since $e \notin \triangle(q_0)$ and not read from q_0, it follows that $q_0 \not\approx_H q_1, q_2$. Moreover, it applies that neither q_0 can be split as initial state nor q_1, q_2 as they have one incoming transition. Thus, the algorithm proceeds with the pair (q_0, q_3). Both states are not History-Based Output Equivalent as well. Hence, a splitting of q_3 as shown in Figure 4.5 is performed.

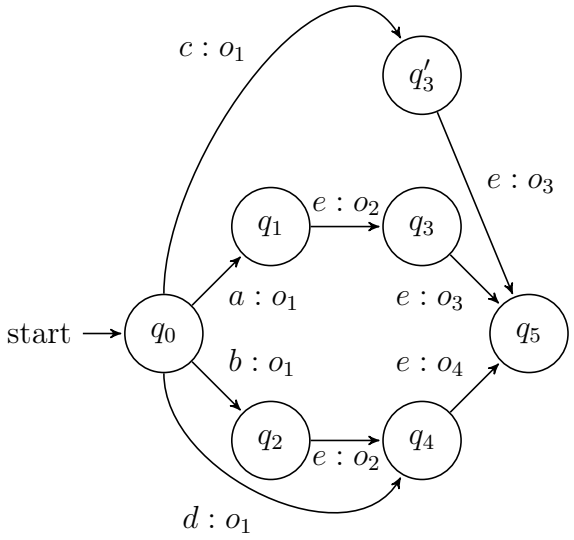


Figure 4.5: Mealy Machine $\mathfrak{M}_{second_1}$.

The cross product of the resulting sets from splitting is then $\{(q_0, q_3), (q_0, q'_3)\}$. History-Based Output Equivalence is still not given because, e.g., $d \notin \triangle(q_3)$, $\triangle(q'_3)$ and d not read from q_3, q'_3. Therefore, the split is reverted and the algorithm checks further the pair (q_0, q_4). Because $q_0 \not\approx_H q_4$ holds, a splitting of q_4 is performed as shown in Figure 4.6.

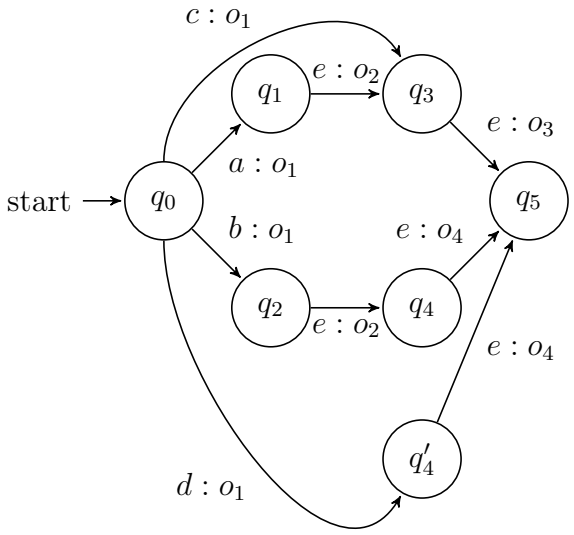


Figure 4.6: Mealy Machine $\mathfrak{M}_{second_2}$.

A similar problem like in the case above appears as $c \notin \triangle(q_4)$, $\triangle(q'_4)$ and c not read from q_4, q'_4. Once again, the split is reverted and the initial Mealy Machine obtained.

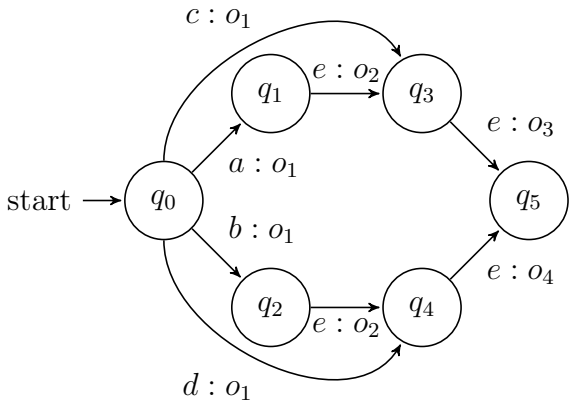


Figure 4.7: Mealy Machine $\mathfrak{M}_{second_3} = \mathfrak{M}_{second}$.

As $\triangle(q_5) = \{e\}$, q_5 a Sink State and $outgoingTransitions(q_0)$ contains the set $\{a, b, c, d\}$, it follows that $q_0 \not\approx_H q_5$. Since both incoming transitions to q_5 contain the input e, q_5 will not be split and the algorithm continues with the next state q_1. Moreover, $\delta(q_1, e) = q_3$, $\delta(q_2, e) = q_4$ and $\omega(q_1, e) = \omega(q_2, e)$. However, because $\omega(q_3, e) \neq \omega(q_4, e)$, it follows $q_1 \not\approx_H q_2$.
The next pair in the traversing order is (q_1, q_3). Initially, both states are not History-Based Output Equivalent as $\omega(q_1, e) \neq \omega(q_3, e)$. Therefore, q_3 is split.

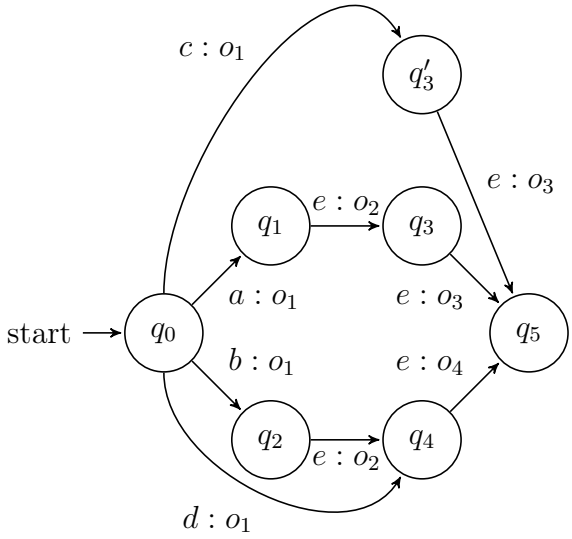


Figure 4.8: Mealy Machine $\mathfrak{M}_{second_4} = \mathfrak{M}_{second_1}$.

It can be seen that $\triangle(q_3)$ has changed to $\{a, e\}$. Thus, since q_1 only reads the input e, both states are recognized as History-Based Output Equivalent. In addition, we merge them obtaining the Mealy Machine in Figure 4.9 and proceed with the adjusted traversing order $\{(q_{13}, q_4), (q_{13}, q_5), (q_2, q_{13}), (q_2, q_4), (q_2, q_5), (q_{13}, q_4), (q_{13}, q_5), (q_4, q_5)\}$.

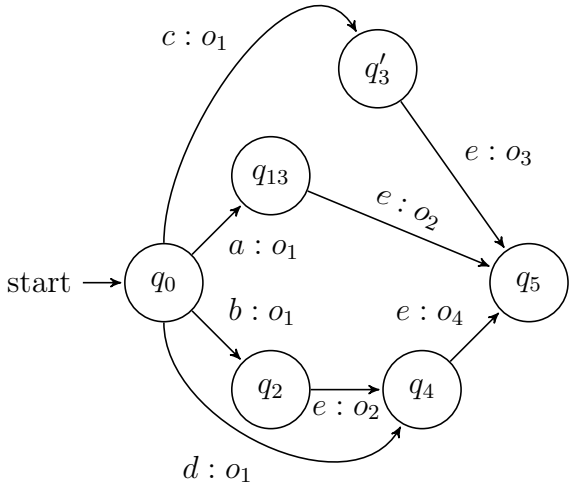


Figure 4.9: Mealy Machine $\mathfrak{M}_{second_5}$.

Consequently, as $\omega(q_{13}, e) \neq \omega(q_4, e)$, q_{13} and q_4 are not History-Based Output Equivalent. Thus, another split will be executed in the next step. As q_{13} has only one incoming transition, we perform State Splitting only on q_4.

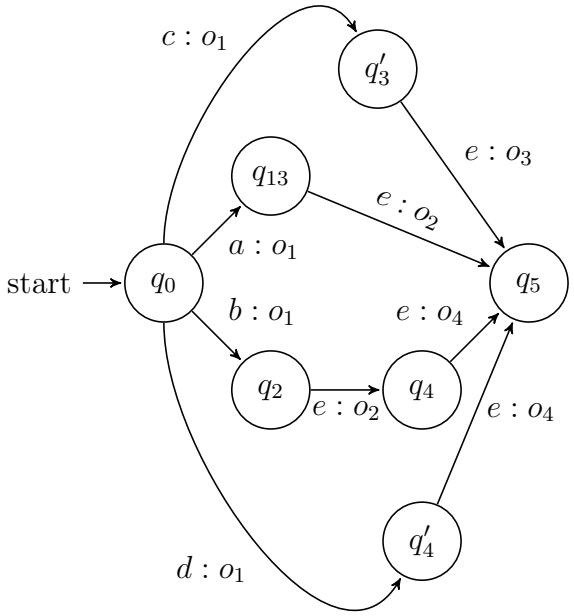


Figure 4.10: Mealy Machine $\mathfrak{M}_{second_6}$.

The cross product $\{(q_{13}, q_4), (q_{13}, q'_4)\}$ is calculated. After the split it applies that $e \in \triangle(q_4)$ and because q_{13} reads only e, $q_{13} \approx_H q_4$. As a result both states are merged and the remaining traversing order gets $\{(q_{134}, q_5), (q_2, q_{134}), (q_2, q_{134}), (q_2, q_5), (q_{134}, q_{134}), (q_{134}, q_5), (q_{134}, q_5)\}$.

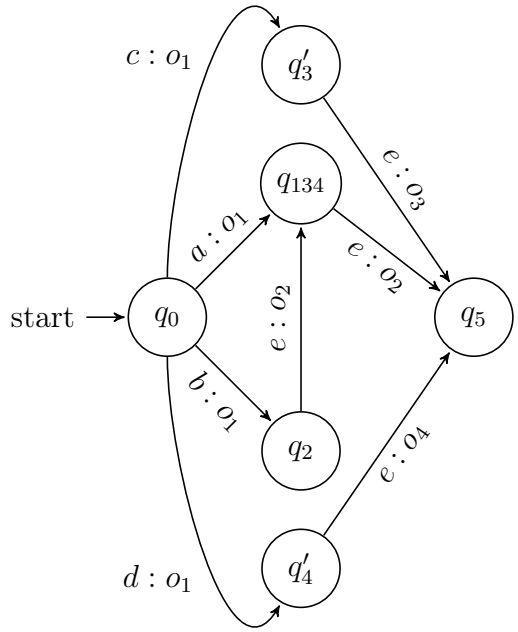
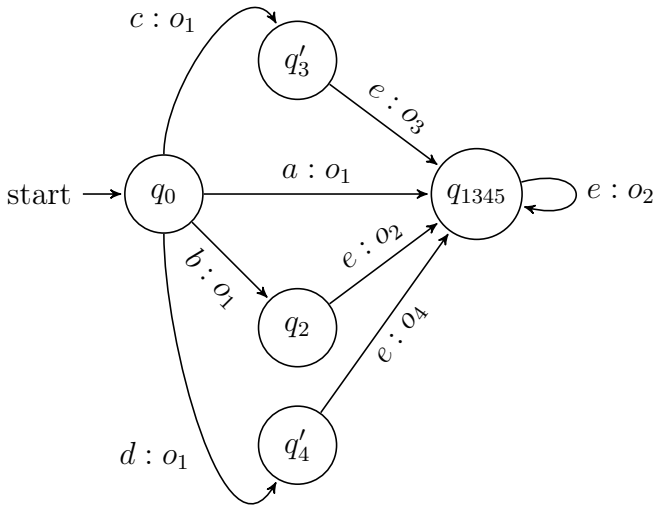


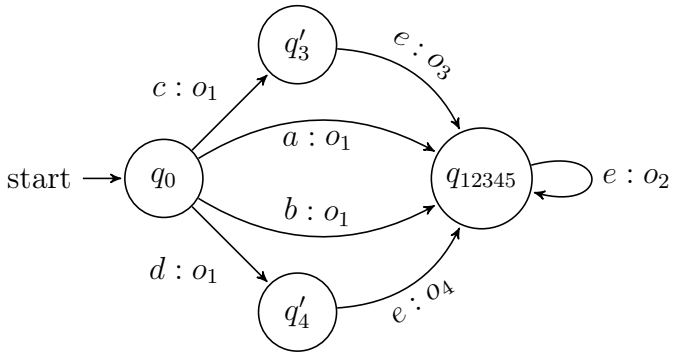
Figure 4.11: Mealy Machine $\mathfrak{M}_{second_7}$.

We check if $q_{134} \approx_H q_5$ holds. Since $outgoingTransitions(q_{134}) = \{(e, q_5)\}$, q_5 is a Sink State and $e \in \triangle(q_5)$, another merge is possible.



The traversing order $\{(q_2, q_{1345}), (q_2,q_{1345}), (q_2, q_{1345}), (q_{1345}, q_{1345}), (q_{1345}, q_{1345}), (q_{1345}, q_{1345})\}$ remains to be considered, i.e., only (q_2, q_{1345}) has to be checked.
As $\omega(q_{1345}, e) = \omega(q_2, e)$ and $\delta(q_{1345}, e) = \delta(q_2, e)$, both states are History-Based Output Equivalent and their merge is possible.

Figure 4.12: Mealy Machine $\mathfrak{M}_{second_8}$.

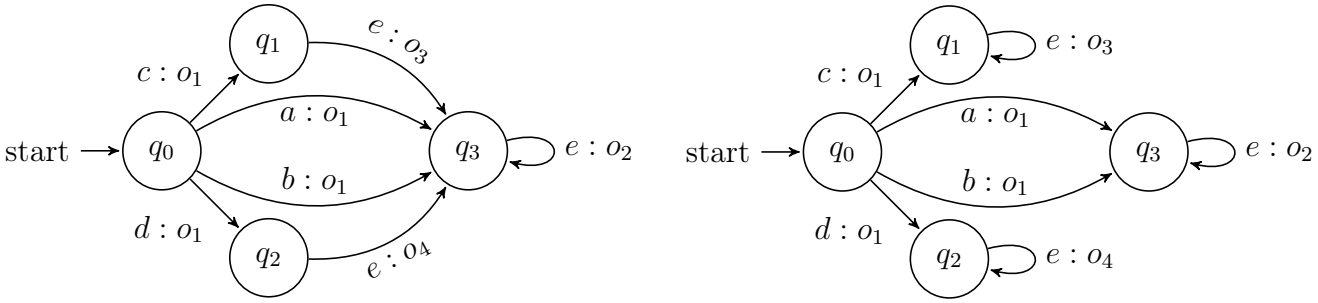


As no more pairs contain different states in the traversing order, this iteration is terminated, and the resulting $\mathfrak{M}_{second_9}$ is obtained. Moreover, as there are no *disabled* transitions, *isolated* states and Sink States, $\mathfrak{M}_{second_9}$ is the first Mealy Machine added to the set checkedAutomata.

Figure 4.13: First obtained Mealy Machine $\mathfrak{M}_{second_9}$ with $\rho_{\mathfrak{M}_{second}}(w) = \rho_{\mathfrak{M}_{second_9}}(w)$ for all $w \in \Sigma^*_{\triangleright}$.

Similarly, the next iterations are performed to obtain all possible different minimization results. After collecting the resulting Mealy Machines from each iteration, a final result has to be chosen.

Instead of describing thoroughly every iteration, we agree with the following checkedAutomata-set equal to isomorphism as obtained by the implementation of the algorithm.



(a) $\mathfrak{M}_1 = \mathfrak{M}'_{second}$ **(b)** $\mathfrak{M}_2$

Figure 4.14: The final set checkedAutomata = $\{\mathfrak{M}_1, \mathfrak{M}_2\}$.

Because Algorithm 7 returns in case of state equality the last Mealy Machine with minimal number of states, $\mathfrak{M}_2$ is the optimized result. Furthermore, $\rho_{\mathfrak{M}}(w) = \rho_{\mathfrak{M}_i}(w)$ for all $w \in \Sigma_{\triangleright}^*$, $i \in [1, 2]$.

Chapter 5

Soundness and Completeness

5.1 History-Based Output Equivalence (Algorithm 1)

Theorem 5.1. *Let $\mathfrak{M} = (Q, q_0, \Sigma, \Omega, \delta, \omega)$ be a Mealy Machine with a pair of states q_1 and q_2 and a disabling relation $\triangleright$. Then,*

$$\text{hEquivalence}(\mathfrak{M}, \triangleright, \emptyset, q_1, q_2) = \textit{true iff } q_1 \approx_H q_2.$$

Proof. For each $a \in \Sigma$, *1* or *2*, or *3* holds.

1. Assume q_1 reads $a \in \Sigma$ ($a \notin \triangle(q_1)$) and there is no a-transition from q_2 (the opposite case is analog). If $a \notin \triangle(q_2)$, $\triangleright$-equivalence can be invalidated by a, therefore hEquivalence returns false. In either case the considered word cannot be read starting from q_2 (or q_1, respectively) and $\triangleright$-equivalence cannot be violated.

2. Assume that both q_1 and q_2 have a a-transition ($a \notin \triangle(q_1) \cap \triangle(q_2)$). Therefore, according to Algorithm 1, $\omega(q_1, a) = \omega(q_2, a)$ and $\delta(q_1, a) \approx_H \delta(q_2, a)$. Thus, $\triangleright$-equivalence holds.

3. Assume that neither q_1 nor q_2 reads $a \in \Sigma$. It is then trivial that $\triangleright$-equivalence cannot be violated by reading x.

To guarantee the termination of the recursive Algorithm 1, we added the base condition $(q_1, q_2) \notin$ consideredPairs. Because the Mealy Machine is finite, all infinite sequences of input symbols result from loops. Therefore it is guaranteed that hEquivalence is terminating because either, if loops exist, already handled pairs of states will be considered, or if the Mealy Machine is not cyclic, states without outgoing transitions, i.e., Sink States, will be reached. □

5.2 Elimination Of Isolated States (Algorithm 2)

Modifications made by the procedure cannot contribute to violating the Mealy Machines behavior. If no path from the initial state to a particular other state exists, then the latter can not be reached and thus can be removed. As we check for each state if there is a path to it, starting from the initial one, we do not need to repeat

the procedure iteratively. Thus, each state has to be considered only once, and complete results are apparent.

5.3 Elimination Of Disabled Transitions (Algorithm 3)

When a transition is *disabled* for a state, we remove it as it cannot be used. Because a *disabled* transition can not be consequently *enabled* for a reachable state whenever another *disabled* transition is deleted, the order in which states are considered is not critical.

Furthermore, since the elimination of a transition may contribute to the appearance of new *disabled* transitions, the procedure must be repeated until nothing changes, which is guaranteed by the finiteness of the Mealy Machine.

5.4 State Splitting (Algorithm 4)

For a state q with $|incomingTransitions(q)| > 1$, the Procedure SPLITSTATE checks the number n of unique transitions, i.e., transitions with different input/output mappings, and creates n fresh states. Moreover, each predecessor node of q still uses the same transitions as before. Although $\triangle$-sets can change, all split states will be History-Based Output Equivalent as containing the same outgoing transitions, i.e., only the third $foreach$-case in Algorithm 1 will be considered. Thus, the preservation of the Mealy Machines behavior after State Splitting is guaranteed.

5.5 State Merging (Algorithm 5)

The correctness of MERGE is evident as only applied to History-Based Output Equivalent states - each incoming transition is preserved, all outgoing transitions are transferred by extraction of copies. Besides, the old states with their corresponding incoming and outgoing transitions are deleted.

5.6 Merging The Sink States To Predecessors (Algorithm 6)

Theorem 5.2. *Let $\mathfrak{M}_1 = (Q, q_0, \Sigma, \Omega, \delta, \omega)$ be a Mealy Machine with a pair of states q_1 and q_2 such that q_2 is a Sink State and q_1 has one outgoing transition being of the form $\delta_1(q_1, a) = q_2$, $\omega_1(q_1, a) = o$. Let further $\mathfrak{M}_2 = (Q', q_0, \Sigma, \Omega, \delta', \omega)$ be a Mealy Machine with equal states and transitions as $\mathfrak{M}_1$, except for turning outgoing transitions of q_1 into loop transitions using Algorithm 6. Then $\rho_{\mathfrak{M}_1}(w) = \rho_{\mathfrak{M}_2}(w)$ $\forall w \in \Sigma^*_{\triangleright}$, where $\triangleright$ is reflexive.*

Proof. Let $w = a_1 \cdot ... \cdot a_k \in \Sigma^*_{\triangleright}$ be an input stream accepted by a Mealy Machine with a disabling relation $\triangleright$. Then it holds that $a_i \not\triangleright a_j$ for any $i < j$. Without loss of generality, we presume all states in $\mathfrak{M}_1$ are reachable and $\mathfrak{M}_1$ visits both q_1 and q_2.

Assume $\mathfrak{M}_1$ visits q_1 upon reading a_i for some $i < k$. By definition, $\mathfrak{M}_2$ also visits q_1 upon reading a_i. Now consider a_{i+1}. By the construction of $\mathfrak{M}_2$ it holds that $\delta_1(q_1, a_{i+1}) = q_2$, $\omega_1(q_1, a_{i+1}) = o$ and $\delta_2(q_1, a_{i+1}) = q_1$, $\omega_2(q_1, a_{i+1}) = o$. Since q_2 is a Sink State we obtain that $\delta_1(q_2, a_j),\ \omega_1(q_2, a_j),\ j > i+1$ are not defined. Because we perform the merging only for reflexive relations, we have then by definition of q_1 and $\mathfrak{M}_2$ that the loop-transition $\delta_2(q_1, a_{i+1})$ is not allowed. Thus, we can conclude that $\rho_{\mathfrak{M}_1}(w) = \rho_{\mathfrak{M}_2}(w)\ \forall w \in \Sigma^*_{\triangleright}$. □

5.7 Mealy Machine Minimization (Algorithm 7)

5.7.1 Soundness

After verifying each procedure which modifies the Mealy Machine, it remains to be outlined why all of them work correctly in combination, i.e.,

$$\rho_{\mathfrak{M}}(w) = \rho_{\text{OPTIMALMINIMIZATION}(\mathfrak{M}(w))}\ \forall w \in \Sigma^*_{\triangleright}$$

for a given Mealy Machine $\mathfrak{M}$ and a disabling relation $\triangleright$.

Without loss of generality, we look at a random traversing order and show that the returned Mealy Machine is equivalent to the initial one.

The procedure MINIMISE is called for a calculated traversing order. Because HEQUIVALENCE is sound as only executed for a Mealy Machine without *disabled* transitions, we know that the considered states are $\triangleright$-equivalent and thus, they are mergeable. In either case SPLITSTATE is executed. As it preserves the behavior of the Mealy Machine, it cannot invalidate the correctness of the Minimisation Algorithm. Moreover, we have already shown that ELIMINATEDISABLED, DELETEISOLATED and MERGESINKSTATES are sound. As all underlying algorithms are correct, the soundness of Algorithm 7 is preserved.

5.7.2 Completeness

In the following subsection, we go even further and show that apart from being sound, Algorithm 7 is also complete. Thus, executing it step-by-step, we always find a minimized Mealy Machine if it exists. As counterexamples have not been found, we also expect optimality, i.e., finding the best existing solution. First of all, we make observations about the used procedures.

1. The consideration order of the states is crucial for the final result - both for completeness and optimality.

    1.1. When we merge two states, no new orders can be derived because we only delete states. Therefore, adjustments in the traversing order are needed - we have to replace each additional appearance of the merged states with the newly obtained state.

    1.2. When we split two states, new orders can be derived. In such cases, the previous states must be deleted, and the resulting split states added to the traversing order. Thus, for each obtained Mealy Machine, the algorithm should be repeated with the freshly created states.

2. If two considered states q_1 and q_2 are split, then a pair of the resulting states can become History-Based Output Equivalent. This can contribute to the History-Based Output Equivalence only of their predecessors and not of the successor states because each successor state has the same incoming transitions as before the split. However, if the predecessors have at the moment of consideration identical transitions to q_1 and q_2 and q_1 and q_2 have not been equivalent then, neither the predecessors have been in consequence. Nevertheless, after the split of q_1, q_2 and eventual merging, all predecessors with identical transitions must be accordingly rechecked, because then they could have become History-Based Output Equivalent.

3. Through performing splitState *disabled* transitions and thereby Sink States may occur.

4. When State Splitting obtains History-Based Output Equivalence, saving the changes is intact, because we then can not get worse.

    4.1. For each obtained Mealy Machine which has not been considered before we call the procedure optimalMinimization iteratively to handle the Split States. Since all Split States from any state q are then History-Based Output Equivalent, the not merged ones can be subsequently identified as equivalent and merged in the next procedure calls.

    4.2. If two states q_1 and q_2 have been History-Based Output Equivalent but split in consideration with other states, the resulting states from splitting will still be History-Based Output Equivalent because transitions are only subsequently *disabled* and not *enabled*. Thus, possible History-Based Output Equivalence cannot get lost through splitting.

After making these observations, we now use them to show completeness. Besides, the completeness of eliminateDisabled and deleteIsolated has been outlined in Section 5.2 and 5.3.

For each pair of states (q_1, q_2) one of the following cases appears:

- $q_1 \approx_H q_2$. Because Algorithm 1 is sound, it detects it.
- $q_1 \not\approx_H q_2$, i.e., $\exists w \in \Sigma_{\triangleright}^* : \omega^*(q_1, w) \neq \omega^*(q_2, w)$.

In the second case, both states cannot be directly merged. Nevertheless, the problem might be that there is more than one path to a state and orthogonality w.r.t. an input symbol is therefore not given. However, an equivalent Mealy Machine with fewer states can still exist, as depicted in Figure 2.4.

To avoid such situations, we introduced the procedure splitState as an intermediate step. Let the set of all unique transitions, i.e., transitions with different input/output mappings and not self-loop transitions be described as follows:

$$uniqueTransitions(q) = \{(q_1, a) \mid q \neq q_1,\ (q_1, a) \in incomingTransitions(q) \text{ and}$$

$$\forall (q_2, b) \in incomingTransitions(q),\ q_2 \neq q_1 \Rightarrow a \neq b \text{ or } \omega(q_1, a) \neq \omega(q_2, b)\}.$$

By State Splitting we induce for any Split States q_1 and q_2 into $q_{1_1}...q_{1_k}$, $q_{2_1}...q_{2_l}$, $k,\ l \in \mathbb{N}_{\geq 2}$

$$|uniqueTransitions(q_1)| = k,\ |uniqueTransitions(q_2)| = l$$

that

$$|uniqueTransitions(q_{1_1})| = \ ... \ = |uniqueTransitions(q_{1_k})| = \ ... \ =$$

$$|uniqueTransitions(q_{2_1})| = \ ... \ = |uniqueTransitions(q_{2_p})| = 1.$$

It is then clear that History-Based Output Equivalence coincides with the ability to merge any pair (q_1', q_2') with $|outgoingTransitions(q_1')| = |outgoingTransitions(q_2')| = 1$ if we exclude self-loops.

Thus, to reach completeness, the goal is to perform State Splitting for each pair of states that is initially not History-Based Output Equivalent. Because of observation 2, we ensure that we do not omit any merging opportunities. Moreover, according to observation 1, we iteratively call the algorithm with the freshly created states, so that further merging possibilities are identified if given.

Also, it is guaranteed that modifying the Mealy Machine is always performed at the right moment:

- For Split States, this is explained in observation 4.
- In all other cases, the modifying rule is independent of the optimalMinimization iterative calls. Thus, the possibility of optimization can be directly identified.

Finally, we clarify why we expect for Algorithm 7 to be optimal. This is presumed by the iterative call of the procedure when modifications obtain new Mealy Machines. Thus, we look for all possible minimization solutions and then return the optimal. Because the Mealy Machine is finite, the termination is guaranteed since at some point splitState will not cause History-Based Output Equivalence, and no states would be mergeable. In such cases, the same Mealy Machine will be obtained, and the algorithm will not be iteratively called for it. Also, after all possible mergings/splittings are performed, eliminateDisabled, deleteIsolate and mergeSinkStates contribute to further minimization improvement.

Chapter 6

Evaluation

This chapter focuses on evaluating the quality of the results and their reliability. We test our approach with Mealy Machine models either provided by the German Aerospace Center (DLR) [1] or own models specially created for testing purposes. All benchmarks can be found in the *resources* folder of the project (see Appendix A), and a graphical visualization helps to track out the changes.

For the comparison, we consider a reflexive disabling relation and outline the trade-off between standard approaches based on trace-equivalence and the expanded synthesis methodology presented in this thesis. However, we do not count empty outputs as useless. Thus, removing them is not a minimization step that indeed preserves equivalence.

Figures 6.1 and 6.2 show the outcome of the algorithm on the tested benchmarks, and convince in the effectiveness of our approach. It can be seen that there is a significant improvement in comparison with classical trace-equivalence minimization. Nevertheless, it cannot be concluded that the number of states influences the percentage of state space reduction. Instead, the considered reflexive disabling relation contributes to the notable differences.

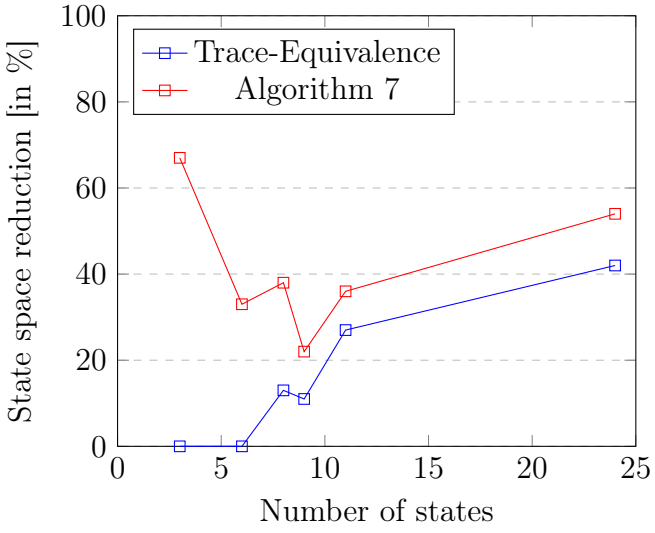


Figure 6.1: Trade-off between trace-equivalence and the developed Minimization Algorithm 7.

[1] German Aerospace Center (DLR): `https://www.dlr.de/dlr/en/desktopdefault.aspx/tabid-10002/`. *Accessed: 2019-25-07.*

Benchmarks	Initial number of states	Number of states after partition refinement	State space reduction	Number of states after Algorithm 7	Total state space reduction	Running time
graph1	6	6	0%	4	33%	0.616s
graph2	3	3	0%	1	67%	0.549s
synthesized	8	7	13%	5	38%	0.903s
synthesized2	9	8	11%	7	22%	0.999s
mdp	11	8	27%	5	36%	1.194s
mdp2	24	14	42%	11	54%	1.468s

Figure 6.2: Results from executing Algorithm 7 on six benchmarks in comparison to standard partition refinement algorithms.

The exact minimization can be tracked with the help of Figure 6.2. It reveals that there are certain situations in which the number of states can be reduced by half. Apart from this slightly disappointing performance, our findings appear to be well substantiated by the minimization results. Given that our findings are based on a limited number of benchmarks, the results from such analyses should be, therefore, treated with the utmost caution.

Chapter 7

Conclusion

7.1 Discussion

Algorithms for automata minimization have been broadly studied and proven for both efficiency and effectiveness. However, there is a special class of automata, so-called output-extended automata for which new methodologies have been considered.

Besides searching for equivalent states in Finite State Machines with standard partition refinement approaches, this thesis has investigated more aggressive minimization techniques that go beyond state merging. A starting point in our approach was the study conducted by Noll et al. [2017]. We have taken into account the methodology described there and further unrolled it to embrace a broader range of use-cases. Considerable insight has been gained concerning the algorithm's completeness. Based on the implementation developed in [Noll et al., 2017], we have devised a strategy which takes any binary relation, which we refer to as a disabling relation, and considers it as a starting point for the minimization. The developed methodology still achieves the trace-equivalence minimization and discusses the notion of orthogonal states with a rule for merging them. Besides, we tried to solve the problems in the present findings. Indeed, the strategy presented in [Noll et al., 2017] gives a sound methodology but does not succeed to achieve completeness. Thus, our goal was to expand the merging rule.

The presented approach establishes a kind of normal form. Furthermore, we used State Splitting as an intermediate step. Indeed, the normal form might increase the number of states for some collected transducers while the algorithm is executed, but we ensured by giving a formal proof of soundness and completeness that the obtained final results are always correct and further complete. Also, by considering every possible minimization option, we made considerable progress concerning optimality.

Moreover, our approach has important implications in finding solutions for the presented future work in [Noll et al., 2017]. Because the model deals with any binary relation without claiming it to be reflexive or have any custom properties, it could solve more problems considered as a cumbersome in the present approaches. For example, observing the Mealy Machines as a fundamental component in developing FDIR strategies in reliability engineering for aerospace systems [Noll et al., 2017], we made it possible to include notions of repair, such as restarts.

7.2 Outlook

In the future, consideration should be made about how to make the presented algorithm more efficient. A possible direction could be considering more than two states in each iteration and testing them for History-Based Output Equivalence. However, we can take into account that heuristic approaches might be needed to deal with State Splitting in such cases when choosing which pair of split states should be merged if more than one is present. We can consider the lack of proof as well as counterexamples that the algorithm is optimal as another critical part. Thus, this is a direction to work in. Also, one needs to wonder if the consideration order of states plays a role to obtain minimal results.

Moreover, further properties of a given binary relation could be examined to find the possibility to adopt minimization strategies depending on that. We have seen, that Merging The Sink States To Predecessors as described in Chapter 3.3.5 is one possible approach when the given disabling relation is reflexive. However, there might be other tricks in various potential cases that contribute to decreasing the number of states in the considered automaton.

Another future direction could be adapting more real-world use-cases besides those obtained from aerospace systems as we have seen that Finite State Machines have broader application in everyday life, e.g., in modeling microwave ovens, ATMs, elevators etc. Thus, we can study how the merging rules presented in the thesis can be transferred to handle different scenarios.

Appendix A

Implementation Details

This appendix provides implementation details of the algorithms presented in this thesis. Moreover, it discusses the publicly available software packages embedded in our program. The source code of the program and further usage possibilities are accessible in GitLab [1].

A.1 General Information

In the implementation of the algorithm for Minimizing Mealy Machines With Dependent Inputs, we applied an object-oriented approach in the Java programming language. Moreover, provided basic data and generic structures play a significant role in the developed program.

For project management, we chose Maven as it encompasses the project object model, facilitates handling of the programs life cycle Compilation → Unit testing → Build, and unburdens the dependency management. The latter was crucial because the Java-based project manages external dependencies.

A.2 Embedded libraries

The apparatus is based on three core libraries used to simplify the implementation of the algorithms and to provide more functionality such as DOT-parsing and visualization. An installation roadmap can also be found in the GitLab repository.

A.2.1 AutomataLib

An essential element of the implementation [2] is the usage of the AutomataLib library for incremental automata construction [Dortmund University of Technology, 2015]. It provides support for modeling graph-based structures, different automata theory algorithms, as well as serialization and visualization via Graphviz.

AutomataLib comes with many base classes to facilitate modeling an automaton. The main goal of AutomataLib which we made use of, is the precise mapping between mathematical concepts and Java implementation.

[1] GitLab Repository: `https://git.rwth-aachen.de/mirela.mileva66/bachelor-arbeit-i2`

[2] We have not used an official release but the AutomataLib Version 0.9-SNAPSHOT which is not available in the central Maven Repository and must be therefore locally added as jar-file.

To develop our algorithm, we chose the class FastMealy as it enables modifications of the Mealy Machine, which is essential for the provided work. Moreover, it builds Mealy Machines from a finite, incrementally growing set of inputs/outputs. As illustrated in Listing A.1, the FastMealy-class consists of crucial concepts which serve as a means of characterizing specific properties of the Mealy Machine. Furthermore, the library implements classes for describing states and transitions in a Mealy Machine and provides basic functionality to deal with them, as shown in Listings A.2, A.3.

In the implemented approach, we could set transitions to each state in a FastMealy Machine as "attributes". Besides, each transition object provides a container with information about the successor state and the output. Then, the complete source to target transition is set with setTransition(S state, I input, T transition).

```
public class FastMealy<I, O> extends
    AbstractFastMutableDet<FastMealyState<O>, I,
    MealyTransition<FastMealyState<O>, O>, Void, O> implements
    MutableMealyMachine<FastMealyState<O>, I,
    MealyTransition<FastMealyState<O>, O>, O>,
    StateLocalInputMealyMachine<FastMealyState<O>, I,
    MealyTransition<FastMealyState<O>, O>, O> {
  public FastMealy(Alphabet<I> alphabet) {
    super(alphabet);
  }

  public FastMealyState<O>
    getSuccessor(MealyTransition<FastMealyState<O>, O> transition) {
    return (FastMealyState)transition.getSuccessor();
  }

  public O getTransitionOutput(MealyTransition<FastMealyState<O>, O>
    transition) {
    return transition.getOutput();
  }

  public MealyTransition<FastMealyState<O>, O>
    createTransition(FastMealyState<O> successor, O properties) {
    return new MealyTransition(successor, properties);
  }

  public void setTransitionOutput(MealyTransition<FastMealyState<O>,
    O> transition, O output) {
    transition.setOutput(output);
  }

  protected FastMealyState<O> createState(Void property) {
    return new FastMealyState(this.inputAlphabet.size());
  }
}
```

Listing A.1: FastMealy implementation (as defined in [Dortmund University of Technology, 2015]).

```
public abstract class AbstractFastState<T> extends
    AbstractMutableNumericID {
  private final ResizingArrayStorage<T> transitions;

  public AbstractFastState(int initialNumOfInputs) {
    this.transitions = new ResizingArrayStorage(Object.class,
    initialNumOfInputs);
  }

  public final boolean ensureInputCapacity(int capacity) {
    return this.transitions.ensureCapacity(capacity);
  }

  public final void setTransitionObject(int inputIdx, T transition) {
    this.transitions.array[inputIdx] = transition;
  }

  public final void clearTransitionObjects() {
    for(int i = 0; i < this.transitions.array.length; ++i) {
      this.clearTransitionObject
      (this.getTransitionObject(i));
      this.transitions.array[i] = null;
    }

  }

  public final T getTransitionObject(int inputIdx) {
    return this.transitions.array[inputIdx];
  }

  public String toString() {
    return "s" + this.getId();
  }
}
```

Listing A.2: Implementation of a state in a Mealy Machine (as defined in [Dortmund University of Technology, 2015]).

```
public class MealyTransition<S, O> {
  private final S successor;
  private O output;

  public MealyTransition(S successor, O output) {
    this.successor = successor;
    this.output = output;
  }

  public O getOutput() {
    return this.output;
  }

  public void setOutput(O output) {
    this.output = output;
  }
  public S getSuccessor() {
    return this.successor;
  }
}
```

Listing A.3: Implementation of a transition in a Mealy Machine (as defined in [Dortmund University of Technology, 2015]).

Moreover, we use the AutomataLib built-in future for visualizing automata and graphs with the Graphviz DOT tool. Rendering and displaying the Mealy Machines in the application within the Java interface not only helps us to follow the minimization changes but also completes our program.

A.2.2 Graphviz - Graph Visualization Software

Automatic graph drawing has essential applications in different domains. Graphviz [Ellson and Gansner, 2016] has many useful features for detailed diagrams. It enables defining options for colors, fonts, node layouts, etc. Besides, the software supports manually editing files as raw text or within a graphical editor. It takes a text description of a graph and provides a graphical representation that could be exported in several useful formats.

Our application also makes use of the Graphviz layout programs and has built-in visualization support. As the vital functionality was already implemented in the AutomataLib library, we have further adapted it to the needs of the program. This not only enables seeing the graph after changes done by the Minimization Algorithm but also serves as an entry- and end-point of the program. Thus, when the user imports a DOT-based file, we visualize it before any changes are done and display as a final report the Mealy Machine after the minimization so that the user can better retrace the changes (see Figure A.1).

A.2.3 JPGD - Java-based Parser for Graphviz Documents

Besides rendering Graphviz Documents from file formats, our application further modifies the described Mealy Machine. Thus, it was crucial to read to Graphviz document and to create an easy-to-use data structure.

The mentioned functionality is achieved with the pure Java JPDG parser [Merz, 2016] which transforms such a graph description from a DOT format into a data

structure containing the definition of the graph with its nodes and edges, which we refer to as states and transitions.

However, to enable the functionality within the application, the JPGD has to be installed locally as it is not released in the Central Maven Repository.

A.3 Algorithm Implementation

Our code structure includes six packages; each of them implements a basic functionality critical for the algorithm. For each of the packages, we further provide essential implementation details.

DOTPARSING: The primary task of the package is to read a graph DOT-representation and to create a data-structure, initializing all states and transitions as textually described. Moreover, all inputs are taken, and a finite alphabet is created from them. Thus, DOTPARSING serves as an entry point of our program and enables the use of an external source file besides manually creating a Mealy Machine object. Therefore, it gives us a Mealy Machine object on which we could execute our algorithm.

RELATION: RELATION defines a disabling relation as a fundamental part of our approach. The disabling relation serves as an extension of an underlying binary relation. In our implementation, the disabling relation is over the finite input alphabet of the defined Mealy Machine.

The DISABLINGRELATION-class gives us the main functionality to decide if two states are orthogonal w.r.t. an input symbol and, therefore, could potentially be merged. This is achieved by computing all paths that lead to some state q from the initial one. Also, we check which inputs are read on each path and so define a set of DISABLEDINPUTS, i.e., inputs that the Mealy Machine cannot read after visiting q.

```
package relation;

public class DisablingRelation extends Relation {

  public DisablingRelation(List<Pair> relationEntries) {
    super(relationEntries);
  }

  public Set<String> disables(String val) {
    return getSecondArgs(val);
  }

  public Set<String> disabledInputs(FastMealyState state, FastMealy
   mealy) {
    Set<String> disabledFinal = new HashSet<>();
    Set<String> inputs = symsOnEachPath(state, mealy);
    for (String input : inputs) {
      Set<String> disabledInputs = disables(input);
      disabledFinal.addAll(disabledInputs);
    }
    return disabledFinal;
  }
```

```
  public Set<String> symbolsOnEachPath(FastMealyState state, FastMealy
   mealy) {
    List<List<String>> allInputs =
   StateHandler.findAllPaths((FastMealyState)
   mealy.getInitialState(), state, mealy);
    Set<String> orthStates = new HashSet<>();

    for (Object alph : mealy.getInputAlphabet()) {
      boolean found = true;
      boolean flag = false;
      for (List<String> path : allInputs) {
        flag = true;
        if (!path.contains(alph)) {
          found = false;
          break;
        }
      }

      if (found && flag)
        orthStates.add((String) alph);
    }
    return orthStates;
  }

  public void setReflexive(FastMealy mealy) {

    for (String symbol : (Collection<String>)
   mealy.getInputAlphabet()) {
      this.addToRel(Pair.of(symbol, symbol));
    }
  }
}
```

Listing A.4: Abstract from the implementation of a disabling relation.

EQUIVALENCE: The package contains the classes HistoryEquivalence, TraceEquivalence, ComputedFunctionEquivalence.

Basically, ComputedFunctionEquivalence is used for testing purposes. It checks Mealy Machines Equivalence before and after performing the Minimization Algorithm. This is achieved by computing all *allowable* input sequences in both Mealy Machines, i.e. all $w \in \Sigma_{\triangleright}^{*}$ for a given Σ and $\triangleright$, and comparing them.

TraceEquivalence is a feature of the AutomataLib library, which checks trace-equivalence using a standard partition refinement algorithm as the ones described in [Aufenkamp, 1958] and [Aufenkamp and Hohn, 1957]. To increase efficiency, we make use of the already implemented approach, which enables initial minimization that could significantly reduce the number of states and transitions in a Mealy Machine.

HistoryEquivalence is the main factor in the algorithm and is used to check if two states in a Mealy Machine are History-Based Output Equivalent according to a disabling relation. The implementation is entirely borrowed from the pseudo-code introduced in Algorithm 1.

STATES: The package provides basic functionality to handle states in Mealy Machines. As we have already seen, our algorithm must collect information about the

preceding and subsequent states of a given state as well as all paths to it, starting from the initial state. Moreover, for example, to check orthogonality between states, we need to compute a set of inputs on the outgoing transitions for each state. We added all this functionality in the class StateHandler.

To gather the needed information in a separate object, we implemented the class TransDesc to make it easier to access all properties of a transition such as the transition itself, the successor/predecessor state as well as the input/output pair.

mealy: The mealy-package contains only the class mealyHandler, where we added the procedures eliminateDisabled, deleteIsolated, merge, splitState and mergeSinkStates. For their implementation we strictly adhered to the pseudo-code introduced in Chapter 3.

However, an essential function of the class is copy, which enables creating a low-level copy of the considered Mealy Machine, and returns a mapping between the two automata. To achieve this, we used the AutomatonLowLevelCopy-class from the AutomataLib library and further adapted it.

```
public static <S1, S2, I, T1, T2, SP, TP> Map<S1, S2>
 copy(UniversalAutomaton<S1, ? super I, T1, ? extends SP, ? extends
 TP> in,
Collection<? extends I> inputs, MutableAutomaton<S2, I, T2, ? super
 SP, ? super TP> out) {

  //creates a low-level copy of the automaton
  final Mapping<S1, S2> orig =
 AutomatonLowLevelCopy.copy(AutomatonCopyMethod.
  STATE_BY_STATE, in, inputs, out);
  final Map<S1, S2> copy =
 Maps.newHashMapWithExpectedSize(in.size());

  for (final S1 s: in) {
    copy.put(s, orig.get(s));
  }

  return copy;
}
```

Listing A.5: Implementation of a low-level automaton copy (adapted from [Dortmund University of Technology, 2015]).

Now it remains to present the main Minimization Algorithm and explain why copying a Mealy Machine is crucial to accomplish our approach.

minimiser: The Minimiser-class in the package is the central entity implementing the algorithm for minimizing Mealy Machines according to a given disabling relation. It makes use of different methods to provide a complete solution.

As the procedure otptimalMinimisation is called iteratively while new Mealy Machines are obtained, we needed to implement a variation of an "equal"-function which checks if the visual representation of a Mealy Machine is contained in a list.

This is established by verifying the following three conditions.

- Trace-equivalence with the comparing machine,
- equal number of transitions and
- equal number of states.

```
private static boolean isContained(FastMealy currentMealy,
 List<FastMealy> checkedAutomata) {

  for (FastMealy mealy : checkedAutomata) {

    if (TraceEquivalence.checkEquivalence(mealy, currentMealy,
 mealy.getInputAlphabet()) &&
        mealy.getStates().size() == currentMealy.getStates().size()
 &&
        MealyHandler.numberOfTransitions(mealy) ==
 MealyHandler.numberOfTransitions(currentMealy)) {
        return true;
    }
  }

  return false;
}
```

Listing A.6: The function checks whether a Mealy Machine has already been considered, i.e., if it is contained in a list of Mealy Machines.

As the initial Mealy Machine is modified during the execution of the algorithm, we used mappings to avoid unexpected behavior. The mToTempMMap enables handling merged or split states. Whenever a state is merged, we map it to the new state so that we can keep a connection between both machines before and after modifications. When we split a state, we let it have a null-value and handle the changes in subsequent calls of the procedure.

```
private static FastMealy minimise(FastMealy m, DisablingRelation
 disablingRelation, Queue<Pair<FastMealyState<String>,
 FastMealyState<String>>> traversingOrder) {

  final Alphabet<String> inputs = m.getInputAlphabet();

  FastMealy<String, String> tempM = new FastMealy<>(inputs);
  final Mapping<FastMealyState<String>, FastMealyState<String>>
 mapping = AutomatonLowLevelCopy.copy(AutomatonCopyMethod.
  STATE_BY_STATE, m, inputs, tempM);

  Map<FastMealyState<String>, FastMealyState<String>> mToTempMMap =
 new HashMap<>();

  // start with the identity mapping
  for (FastMealyState<String> s : (Collection<FastMealyState>)
 m.getStates()) {
    mToTempMMap.put(s, mapping.get(s));
  }
```

```
    // actual "algorithm"
    while (!traversingOrder.isEmpty()) {
      final Pair<FastMealyState<String>, FastMealyState<String>> pair
    = traversingOrder.remove();
      final FastMealyState<String> tempQ1 =
    mToTempMMap.get(pair.getFirst());
      final FastMealyState<String> tempQ2 =
    mToTempMMap.get(pair.getSecond());

      // skip null elements resulted from splitting or identical state
    comparison
      if (tempQ1 == null || tempQ2 == null || tempQ1.equals(tempQ2)) {
        continue;
      }

      Pair resultingOrderAndMealy =
    equivalenceHandler(disablingRelation, inputs, tempM, mToTempMMap,
    tempQ1, tempQ2);
      tempM = (FastMealy<String, String>)
    resultingOrderAndMealy.getFirst();
      mToTempMMap = (Map<FastMealyState<String>,
    FastMealyState<String>>) resultingOrderAndMealy.getSecond();

    }

    return tempM;
  }
```

Listing A.7: MINIMIZE-procedure as described in 3.4.

Also, we check for each pair of states if both are History-Based Output Equivalent. If so, the procedure HANDLEBYEQUIVALENCE is called and in either cases we execute HANDLEBYNOEQUIVALENCE. However, we always copy the Mealy Machine before any changes are performed.

```
  /** Split both states and checks if some of the resulting states are
    equivalent. If so, they are merged. Otherwise, the split is
    reverted.
   */

  private static Pair handleByNoEquivalence(DisablingRelation
    disablingRelation, Alphabet<String> inputs, FastMealy<String,
    String> tempM, Map<FastMealyState<String>, FastMealyState<String>>
    mToTempMMap, FastMealyState<String> tempQ1, FastMealyState<String>
    tempQ2) {


    FastMealy<String, String> splitted = new FastMealy<>(inputs);

    Map<FastMealyState<String>, FastMealyState<String>> splitMapping =
    MealyHandler.copy(tempM, inputs, splitted);

    // splitState() gives us a new automaton
    Pair splitFirst =
    MealyHandler.splitState(splitMapping.get(tempQ1), splitted);
    ArrayList<FastMealyState> resultingStatesTempQ1 =
    (ArrayList<FastMealyState>) splitFirst.getFirst();
```

```
    if ((boolean) splitFirst.getSecond()) {
      splitted.removeState(splitMapping.get(tempQ1));
    }

    Pair splitSecond =
  MealyHandler.splitState(splitMapping.get(tempQ2), splitted);
    ArrayList<FastMealyState> resultingStatesTempQ2 =
  (ArrayList<FastMealyState>) splitSecond.getFirst();

    if ((boolean) splitSecond.getSecond()) {
      splitted.removeState(splitMapping.get(tempQ2));
    }

    //calculate all combinations between sets of resulting states from
  splitting
    List<Pair> crossPr = crossProduct(resultingStatesTempQ1,
  resultingStatesTempQ2);
    boolean splitSuccessful = false;

    for (Pair pairToCheck : crossPr) {

      if (HistoryEquivalence.checkHistoryEquivalence(splitted,
  (FastMealyState) pairToCheck.getFirst(), (FastMealyState)
  pairToCheck.getSecond(), new ArrayList<>(), disablingRelation)) {

        Triple equivalenceCheckResult =
  handleByEquivalence(disablingRelation, inputs, splitted,
  composeMap(mToTempMMap, splitMapping),
            (FastMealyState) pairToCheck.getFirst(), (FastMealyState)
  pairToCheck.getSecond());

        splitMapping = (Map<FastMealyState<String>,
  FastMealyState<String>>) equivalenceCheckResult.getThird();
        boolean mergePossible = (boolean)
  equivalenceCheckResult.getSecond();
        splitted = (FastMealy<String, String>)
  equivalenceCheckResult.getFirst();

        if (mergePossible) {
          splitSuccessful = true;
          break;
        }

      }
    }
    //update the mapping in splitSuccess only if something has been
  merged, i.e. the splitting was successful
    return splitSuccess(tempM, mToTempMMap, splitted, splitMapping,
  splitSuccessful);

  }
```

Listing A.8: Mealy Machine Handler when History-Based Output Equivalence is not given.

Because HANDLEBYEQUIVALENCE must be called before the obtained modifications from the splitting are saved, we need to use a composed mapping as shown in the listing below.

```
private static <K, V1, V2> Map<K, V2> composeMap(Map<K, V1> src,
 Map<V1, V2> map) {
  final Map<K, V2> result =
 Maps.newHashMapWithExpectedSize(src.size());

  for (Map.Entry<K, V1> e : src.entrySet()) {
    result.put(e.getKey(), map.get(e.getValue()));
  }

  return result;
}
```

```
/** Merge both states and update the mapping.
 */
private static Triple handleByEquivalence(DisablingRelation
 disablingRelation, Alphabet<String> inputs, FastMealy<String,
 String> tempM, Map<FastMealyState<String>, FastMealyState<String>>
 mToTempMMap, FastMealyState<String> tempQ1, FastMealyState<String>
 tempQ2) {

  final FastMealy<String, String> merged = new FastMealy(inputs);

  final Map<FastMealyState<String>, FastMealyState<String>>
 mergeMapping =
      MealyHandler.copy(tempM, inputs, merged);

  // where the equivalent state tempQ1 is removed and transitions of
 tempQ2 are updated
  MealyHandler.mergeStates(merged, mergeMapping.get(tempQ1),
 mergeMapping.get(tempQ2), disablingRelation);
  merged.removeState(mergeMapping.get(tempQ1));

  if (mergePossible(merged, tempM, disablingRelation)) {

    // update mapping, to map the just-removed copy of tempQ1 to the
 copy of the (now) equivalent tempQ2
    mergeMapping.put(tempQ1, mergeMapping.get(tempQ2));

    // update (global) mapping
    for (FastMealyState<String> q : mToTempMMap.keySet()) {
      mToTempMMap.put(q, mergeMapping.get(mToTempMMap.get(q)));
    }

    // update tempM
    tempM = merged;

    return Triple.of(tempM, true, mToTempMMap);
  }

  return Triple.of(tempM, false, mToTempMMap);
}
```

Listing A.9: Mealy Machine Handler by History-Based Output Equivalence.

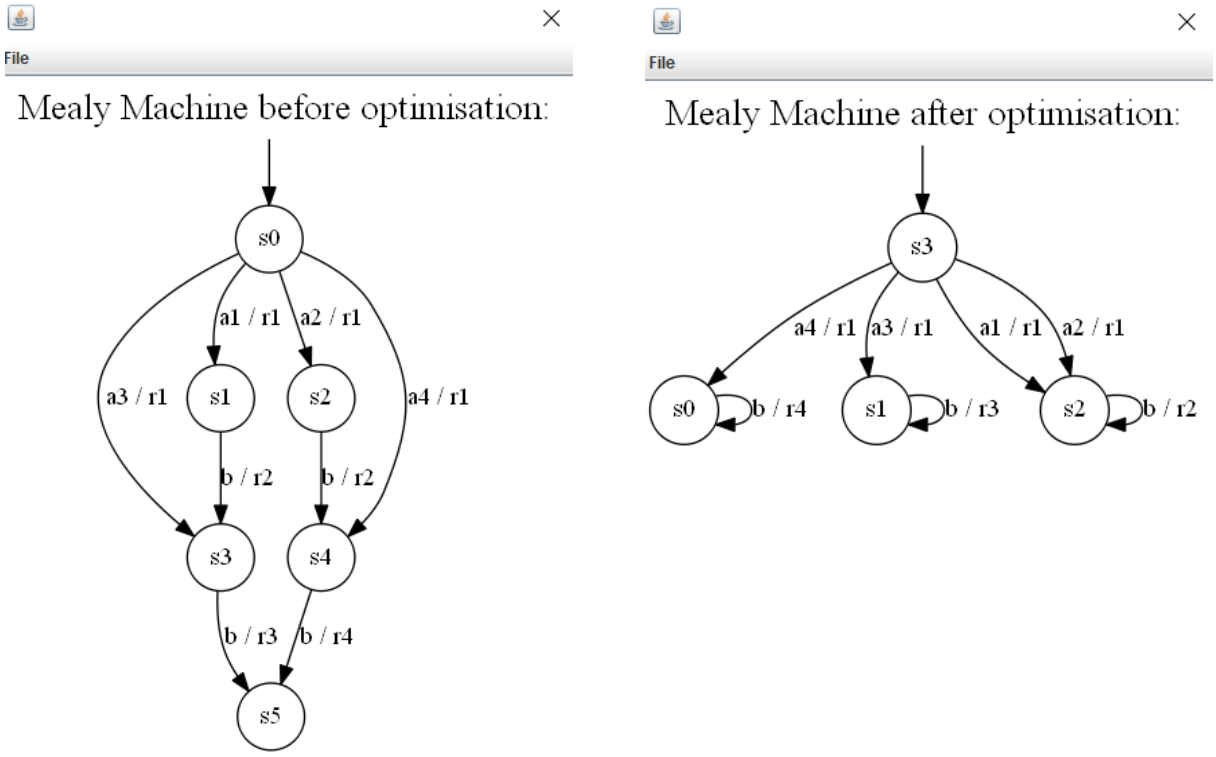


Figure A.1: Visualization of a Mealy Machine before and after the execution of Algorithm 7.

Figure A.1 shows the results of our program for an example Mealy Machine from a DOT-based file and a reflexive disabling relation. Such dialog windows serve as an entry and endpoint of the developed application.

Bibliography

Aufenkamp, D. D. (1958). Analysis of Sequential Machines II. *IRE Transactions on Electronic Computers*, EC-7(4):299–306.

Aufenkamp, D. D. and Hohn, F. E. (1957). Analysis of Sequential Machines. *IRE Transactions on Electronic Computers*, EC-6(4):276–285.

Cummings, C. E. (2002). The Fundamentals of Efficient Synthesizable Finite State Machine Design using NC-Verilog and BuildGates. In *Proceedings of International Cadence Usergroup Conference*, pages 1–27.

Dortmund University of Technology, G. (2015). AutomataLib. `https://learnlib.de/projects/automatalib/`.

Ellson, J. and Gansner, E. (2016). Graphviz - Graph Visualization Software. `https://graphviz.gitlab.io/`.

Gören, S. and Ferguson, F. J. (2007). On State Reduction of Incompletely Specified Finite State Machines. *Comput. Electr. Eng.*, 33(1):58–69.

Higuchi, H. and Matsunaga, Y. (1996). A Fast State Reduction Algorithm for Incompletely Specified Finite State Machine. In *Proceedings of the 33rd Annual Design Automation Conference*, DAC '96, pages 463–466, New York, NY, USA. ACM.

Hopcroft, J. E. (1971). An N Log N Algorithm for Minimizing States in a Finite Automaton. Technical report, Stanford, CA, USA.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Klimovich, A. S. and Solov'ev, V. V. (2010). Transformation of a Mealy Finite–State Machine into a Moore Finite–State Machine by Splitting Internal States. *Journal of Computer and Systems Sciences International*, 49(6):900–908.

Merz, A. (2016). JPGD - Java-based Parser for Graphviz Documents. `http://www.alexander-merz.com/graphviz/`.

Noll, T., Gerndt, A., and Müller, S. (2017). Synthesizing FDIR Recovery Strategies From Non-Deterministic Dynamic Fault Trees.