Chapter 1

Basic Definitions

The first chapter defines fundamentals of this thesis and notation used later on.

1.1 General Mathematical Terms

As our main focus is ω-words, we will require a small extension of natural numbers into the transfinite realm.

1.1.1 Sets and Functions

Definition 1.1.1. The *natural numbers* $\mathbb{N} = \{0, 1, 2, \dots\}$ are the set of all non-negative integers. We define $0 := \emptyset$, $1 := \{0\}$, $2 := \{0, 1\}$, and so forth.

The value ω denotes the "smallest" infinity, $\omega := \mathbb{N}$. For all natural numbers, we write $n < \omega$ and $\omega \not< \omega$. Also, we sometimes use the convention $n + \omega = \omega$.

We denote the set $\mathbb{N} \cup \{\omega\}$ by $\mathbb{N}_\omega$.

Definition 1.1.2. Let X and Y be two sets. We use the usual definition of union ($\cup$), intersection ($\cap$), and set difference ($\setminus$). If some domain ($X \subseteq D$) is clear in the context, we write $X^{\complement} = D \setminus X$.

We use the cartesian product $X \times Y = \{(x, y) \mid x \in X, y \in Y\}$.

We write X^Y for the set of all functions with domain Y and range X. If we have a function $f : D \to \{0, 1\}$, then we sometimes implicitly use it as a set $X \subseteq D$ with $x \in X$ iff $f(x) = 1$. In particular, 2^Y is the powerset of Y.

Definition 1.1.3. Let $f : D \to R$ be a function and let $X \subseteq D$ and $Y \subseteq R$. We describe by $f(X) = \{f(x) \in R \mid x \in X\}$ and $f^{-1}(Y) = \{x \in D \mid \exists y \in Y : f(x) = y\}$.

Definition 1.1.4. Let $X \subseteq D$ be a set. For $D' \subseteq D$, we define $X \restriction_{D'} = X \cap D$. In particular, we use this notation for relations, e.g. $R \subseteq \mathbb{N} \times \mathbb{N}$ and $R \restriction_{\{0\} \times \mathbb{N}}$.

For a function $f : D \to R$, we write $f \restriction_{D'}$ for the function $f' : D' \to R, x \mapsto f(x)$.

1.1.2 Relations and Orders

Definition 1.1.5. Let X be a set. We call a set $R \subseteq X \times X$ a *relation* over X. R is

- *reflexive*, if for all $x \in X$, $(x, x) \in R$.
- *irreflexive*, if for all $x \in X$, $(x, x) \notin R$.
- *symmetric*, if for all $(x, y) \in R$, also $(y, x) \in R$.
- *asymmetric*, if for all $(x, y) \in R$, $(y, x) \notin R$.
- *transitive*, if for all $(x, y), (y, z) \in R$, also $(x, z) \in R$.
- *total*, if for all $x, y \in X$, $(x, y) \in R$ or $(y, x) \in R$ is true.

We call R

- a *partial order*, if it is irreflexive, asymmetric, and transitive.
- a *total order*, if it is a partial order and total.
- a *preorder*, if it is reflexive and transitive.
- a *total preorder*, if it is a preorder and total.
- an *equivalence relation*, if it is a preorder and symmetric.

If R is a partial order or a preorder, we call an element $x \in X$ *minimal* (w.r.t. R), if for all $y \in X$, $(y, x) \in R$ implies $(x, y) \in R$. Similarly, we call it *maximal*, if for all $y \in X$, $(x, y) \in R$ implies $(y, x) \in R$.

We call x the *minimum* of R if for all $y \neq x$, $(y, x) \in R$. We write $x = \min_R X$.

Definition 1.1.6. Let R be a partial order over X. We call a set $S \subseteq Y$ an *extension of R to Y* if $X \subseteq Y$, $R \subseteq S$, and S is a partial order over Y. We use the same notation for total orders, preorders, and total preorders.

Definition 1.1.7. Let R be an equivalence relation over X. R implicitly forms a partition of X into *equivalence classes*. For an element $x \in X$, we call $[x]_R := \{y \in X \mid (x, y) \in R\}$ the equivalence class of x. We denote the set of equivalence classes by $\mathfrak{C}(R) = \{[x]_R \mid x \in R\}$.

1.2 Words and Languages

Definition 1.2.1. A non-empty set of symbols can be called an *alphabet*, which we will denote by a variable Σ most of the time. As symbols, we usually use lower case letters, i.e. a or b.

A *finite word*, usually denoted by u, v, or w, over an alphabet Σ is a function $w : n \to \Sigma$ for some n. We call n the *length* of w and write $|w| = n$. The unique word of length 0 is called *empty word* and is written as ε.

Given $\Sigma^n = \{w \mid w \text{ is a word of length } n \text{ over } \Sigma\}$, we define $\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n$ as the set of all finite words over Σ.

Definition 1.2.2. An *ω-word*, usually denoted by α or β, over an alphabet Σ is a function $\alpha : \omega \to \Sigma$. ω is the length of α and we write $|\alpha| = \omega$. The set Σ^ω then describes the set of all ω-words over Σ.

Definition 1.2.3. A *language* over an alphabet Σ is a set of words $L \subseteq \Sigma^* \cup \Sigma^\omega$. In the context we use it should always be clear whether we are using finite words or ω-words.

Definition 1.2.4. Let $v, w \in \Sigma^*$ and $w_i \in \Sigma^*$ for all $i \in \mathbb{N}$ be words over Σ and $\alpha \in \Sigma^\omega$ be an ω-word over Σ.

The *concatenation* of v and w (denoted by $v \cdot w$) is a word u such that:

$$u : |v| + |w| \to \Sigma, i \mapsto \begin{cases} v(i) & \text{if } i < |v| \\ w(i - |v|) & \text{else} \end{cases}$$

The *concatenation* of w and α (denoted by $w \cdot \alpha$) is an ω-word β such that:

$$\beta : \mathbb{N} \to \Sigma, i \mapsto \begin{cases} w(i) & \text{if } i < |w| \\ \alpha(i - |w|) & \text{else} \end{cases}$$

For some $n \in \mathbb{N}$, the *n-iteration* of w (denoted by w^n) is a word u such that:

$$u : |w|^n \to \Sigma, i \mapsto w(i \mod |w|)$$

The *ω-iteration* of w (denoted by w^ω) is an ω-word α such that:

$$\beta : \mathbb{N} \to \Sigma, i \mapsto w(i \mod |w|)$$

For the purpose of easier notation and readability, we write singular symbols as words, i.e. for an $a \in \Sigma$ we write a for the word $w_a : \{0\} \to \Sigma, i \mapsto a$.
We also abbreviate $v \cdot w$ to vw and $w \cdot \alpha$ to $w\alpha$. Further, we use $\alpha \cdot \varepsilon = \alpha$ for $\alpha \in \Sigma^\omega$.

Definition 1.2.5. Let $L, K \subseteq \Sigma^*$ be a language and $U \subseteq \Sigma^\omega$ be an ω-language.
The *concatenation* of L and K is $L \cdot K = \{u \in \Sigma^* \mid \text{There are } v \in L \text{ and } w \in K \text{ such that } u = v \cdot w\}$.
The *concatenation* of L and U is $L \cdot U = \{\alpha \in \Sigma^\omega \mid \text{There are } w \in L \text{ and } \beta \in U \text{ such that } \alpha = w \cdot \beta\}$.
For some $n \in \mathbb{N}$, the *n-iteration* of L is $L^n = \{w \in \Sigma^* \mid \text{There is } v \in L \text{ such that } w = v^n\}$.
The *Kleene closure* of L is $L^* = \bigcup_{n \in \mathbb{N}} L^n$.

Definition 1.2.6. Let $w \in \Sigma^* \cup \Sigma^\omega$ be a word. We define a substring or subword of w for some $n \leq m \leq |w|$ as $w[n, m] = w(n) \cdot w(n+1) \cdots w(m-1)$. In the case that $m = |w| = \omega$, it is simply $w[n, m] = w(n) \cdot w(n+1) \cdots$. Note that for $n = m$, we have $w[n, m] = \varepsilon$.

Definition 1.2.7. Let $v, w \in \Sigma^* \cup \Sigma^\omega$ be words. We call v

- a *prefix* of w, if there is an $n \in \mathbb{N}_\omega$ with $v = w[0, n]$.
- a *suffix* of w, if there is an $n \in \mathbb{N}_\omega$ with $v = w[n, |w|]$.
- an *infix* of w, if there are $n, m \in \mathbb{N}_\omega$ with $v = w[n, m]$.

Definition 1.2.8. The *occurrence set* of a word $w \in \Sigma^* \cup \Sigma^\omega$ is the set of symbols which occur at least once in w.

$$\mathrm{Occ}(w) = \{a \in \Sigma \mid \text{There is an } n \in |w| \text{ such that } w(n) = a.\}$$

The *infinity set* of a word $w \in \Sigma^\omega$ is the set of symbols which occur infinitely often in w.

$$\mathrm{Inf}(w) = \{a \in \Sigma \mid \text{For every } n \in \mathbb{N} \text{ there is a } m > n \text{ such that } w(m) = a.\}$$

1.3 Automata

Definition 1.3.1. Let Q be a set, Σ an alphabet, and $\delta : Q \times \Sigma \to Q$ a function. We call $\mathcal{S} = (Q, \Sigma, \delta)$ a *deterministic transition structure*. We call Q the states or state space.

For $q \in Q$ and a word $w \in \Sigma^* \cup \Sigma^\omega$, we call $\rho \in Q^{1+|w|}$ the *run* of $\mathcal{S}$ on w starting in q if $\rho(0) = q$ and for all i, $\rho(i+1) = \delta(\rho(i), w(i))$.

Definition 1.3.2. Let $\mathcal{S} = (Q, \Sigma, \delta)$ be a deterministic transition structure. For a set $\Omega \subseteq Q^* \cup Q^\omega$, we say that $\mathcal{S}$ has acceptance condition Ω.

We say that a run ρ of $\mathcal{A}$ on some $w \in \Sigma^*$ is *accepting*, if $\rho \in \Omega$; otherwise, the run is *rejecting*. In either case, we say that $\mathcal{A}$ accepts or rejects w.

The *language* of $\mathcal{A}$ with Ω from $q \in Q$ is the set of all words and ω-words that are accepted by $\mathcal{A}$ from q.

Definition 1.3.3. A *deterministic finite automaton* (or DFA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, F)$, where $F \subseteq Q$, such that (Q, Σ, δ) is a deterministic transition structure and has acceptance condition $\Omega = \{\rho \in Q^* \mid \rho(|\rho| + 1) \in F\}$. For the language of (Q, Σ, δ) with Ω from q, we write $L(\mathcal{A}, q)$.

Definition 1.3.4. A *deterministic parity automaton* (or DPA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, c)$, where $c : Q \to \mathbb{N}$, such that (Q, Σ, δ) is a deterministic transition structure and has acceptance condition $\Omega = \{\rho \in Q^* \mid \min \mathrm{Inf}(c(\rho)) \text{ is even}\}$. For the language of (Q, Σ, δ) with Ω from q, we write $L(\mathcal{A}, q)$.

We call the DPA a *Büchi automaton* (or DBA) if $c(Q) \subseteq \{0, 1\}$. In that case, we use F instead of c.

Definition 1.3.5. Let $\mathcal{S} = (Q, \Sigma, \delta)$ be a deterministic transition structure. We define $\delta^* : Q \times \Sigma^* \to Q$ as $\delta^*(q, \varepsilon) = q$ and $\delta^*(q, w \cdot a) = \delta(\delta^*(q, w), a)$.

Definition 1.3.6. Let $\mathcal{A} = (Q, \Sigma, \delta, c)$ be a DPA. We define $c^* : Q \times (\Sigma^* \cup \Sigma^\omega) \to (\mathbb{N}^* \cup \mathbb{N}^\omega)$ as $c^*(q, w) : 1 + |w| \to \mathbb{N}, i \mapsto c(\delta^*(q, w[0, i]))$.

Chapter 2

Theory

2.1 General Results

We first use this section to establish some general results that are used multiple times in the upcoming proofs.

2.1.1 Equivalence Relations

In general, we use the symbol $\equiv$ to denote equivalence relations, mostly between states of an automata. In general, we have automata $\mathcal{A}$ and $\mathcal{B}$ with states p and q from there respective state spaces. Our relations are then defined on $(\mathcal{A}, p) \equiv (\mathcal{B}, q)$.

Definition 2.1.1. Assuming that $\mathcal{A}$ is a fixed automaton that is obvious in context and p and q are both states in $\mathcal{A}$, we shorten $(\mathcal{A}, p) \equiv (\mathcal{A}, q)$ to $p \equiv q$.

Furthermore, we write $\mathcal{A} \equiv \mathcal{B}$ if for every p in $\mathcal{A}$ there is a q in $\mathcal{B}$ such that $(\mathcal{A}, p) \equiv (\mathcal{B}, q)$; and the same holds with $\mathcal{A}$ and $\mathcal{B}$ exchanged.

Definition 2.1.2. Let $\mathcal{A} = (Q_1, \Sigma, \delta_1)$ and $\mathcal{B} = (Q_2, \Sigma, \delta_2)$ be deterministic transition structures and let $\sim \subseteq (\{\mathcal{A}\} \times Q_1) \times (\{\mathcal{B}\} \times Q_2)$ be an equivalence relation. We call R a *congruence relation* if for all $(\mathcal{A}, p) \sim (\mathcal{B}, q)$ and all $a \in \Sigma$, also $(\mathcal{A}, \delta_1(p, a)) \sim (\mathcal{B}, \delta_2(q, a))$.

The following is a comprehensive list of all relevant equivalence relations that we use.

- Language equivalence, $\equiv_L$. Defined below.
- Moore equivalence, $\equiv_M$. Defined below.
- Priority almost equivalence, $\equiv_\dagger$. Defined below.
- Delayed simulation equivalence, $\equiv_{\mathrm{de}}$. Defined in
- Path refinement equivalence, $\equiv_{\mathrm{PR}}$. Defined in
- Threshold Moore equivalence, $\equiv_{\mathrm{TM}}$. Defined in
- Labeled SCC filter equivalence, $\equiv_{\mathrm{LSF}}$. Defined in

Immediately we define the three first of these relations and show that they are computable.

Language Equivalence

Definition 2.1.3. Let $\mathcal{A}$ and $\mathcal{B}$ be ω-automata. We define *language equivalence* as $(\mathcal{A}, p) \equiv_L (\mathcal{B}, q)$ if and only if for all words $\alpha \in \Sigma^\omega$, $\mathcal{A}$ accepts α from p iff $\mathcal{B}$ accepts α from q.

Lemma 2.1.1. *$\equiv_L$ is a congruence relation.*

Proof. It is obvious that $\equiv_L$ is an equivalence relation. For two states $(\mathcal{A}, p) \equiv_L (\mathcal{B}, q)$ and some successors $p' = \delta_1(p, a)$ and $q' = \delta_2(q, a)$, it must be true that $(\mathcal{A}, p') \equiv_L (\mathcal{B}, q')$. Otherwise there is a word $\alpha \in \Sigma^\omega$ that is accepted from p' and rejected from q' (or vice-versa). Then $a \cdot \alpha$ is rejected from p and accepted from q and thus $p \not\equiv_L q$. $\square$

Lemma 2.1.2. *Language equivalence of a given DPA can be computed in $\mathcal{O}(|Q|^2 \cdot |c(Q)|^2)$.*

Proof. The algorithm is based partially on [1].

Let $\mathcal{A} = (Q, \Sigma, \delta, c)$ be the DPA that we want to compute $\equiv_L$ on. We construct a labeled deterministic transition structure $\mathcal{B} = (Q \times Q, \Sigma, \delta', d)$ with $\delta'((p_1, p_2), a) = (\delta(p_1, a), \delta(p_2, a))$ and $d((p_1, p_2)) = (c(p_1), c(p_2)) \in \mathbb{N}^2$. Then, for every $i, j \in c(Q)$, let $\mathcal{B}_{i,j} = \mathcal{B} \restriction_{Q_{i,j}}$ with $Q_{i,j} = \{(p_1, p_2) \in Q \times Q \mid c(p_1) \geq i, c(p_2) \geq j\}$, i.e. remove all states which have first priority less than i or second priority less than j.

For each i and j, let $S_{i,j} \subseteq 2^{Q \times Q}$ be the set of all SCCs in $\mathcal{B}_{i,j}$ and let $S = \bigcup_{i,j} S_{i,j}$. From this set S, remove all SCCs $s \subseteq Q \times Q$ in which the parity of the smallest priority in the first component differs from the parity of the smallest priority in the second component. The "filtered" set we call S'. For any two states $p, q \in Q$, $p \not\equiv_L q$ iff there is a pair $(p', q') \in \bigcup S'$ that is reachable from (p, q) in $\mathcal{B}$.

We omit the correctness proof of the algorithm here. Regarding the runtime, observe that $\mathcal{B}$ has size $\mathcal{O}(|Q|^2)$ and we create $\mathcal{O}(|c(Q)|^2)$ copies of it. All other steps like computing the SCCs can then be done in linear time in the size of the automata, which brings the total to $\mathcal{O}(|Q|^2 \cdot |c(Q)|^2)$. $\square$

Priority Almost Equivalence

Definition 2.1.4. Let $\mathcal{A} = (Q_1, \Sigma, \delta_1, c_1)$ and $\mathcal{B} = (Q_2, \Sigma, \delta_2, c_2)$ be DPAs. We define *priority almost equivalence* as $(\mathcal{A}, p) \equiv_{\dagger} (\mathcal{B}, q)$ if and only if for all words $\alpha \in \Sigma^\omega$, $c_1^*(p, \alpha)$ and $c_2^*(q, \alpha)$ differ at only finitely many positions.

Lemma 2.1.3. *Priority almost equivalence is a congruence relation.*

Proof. It is obvious that $\equiv_{\dagger}$ is an equivalence relation. For two states $(\mathcal{A}, p) \equiv_{\dagger} (\mathcal{B}, q)$ and some successors $p' = \delta(p, a)$ and $q' = \delta(q, a)$, it must be true that $(\mathcal{A}, p') \equiv_{\dagger} (\mathcal{B}, q')$. Otherwise there is a word $\alpha \in \Sigma^\omega$ such that $c_1^*(p', \alpha)$ and $c_2^*(q', \alpha)$ differ at infinitely many positions. Then $c_1^*(p, a\alpha)$ and $c_2^*(q, a\alpha)$ also differ at infinitely many positions and thus $(\mathcal{A}, p) \not\equiv_{\dagger} (\mathcal{B}, q)$. $\square$

The following definition is used as an intermediate step on the way to computing $\equiv_{\dagger}$.

Definition 2.1.5. Let $\mathcal{A} = (Q_1, \Sigma, \delta_1, c_1)$ and $\mathcal{B} = (Q_2, \Sigma, \delta_2, c_2)$ be DPAs. We define the deterministic Büchi automaton $\mathcal{A} \top \mathcal{B} = (Q_1 \times Q_2, \Sigma, \delta_{\top}, F_{\top})$ with $\delta_{\top}((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$. The transition structure is a common product automaton.

The final states are $F_{\top} = \{(p, q) \in Q_1 \times Q_2 \mid c_1(p) \neq c_2(q)\}$, i.e. every pair of states at which the priorities differ.

Lemma 2.1.4. *$\mathcal{A} \top \mathcal{B}$ can be computed in time $\mathcal{O}(|\mathcal{A}| \cdot |\mathcal{B}|)$.*

Proof. The definition already provides a rather straightforward description of how to compute $\mathcal{A} \top \mathcal{B}$. Each state only requires constant time (assuming that δ and c can be evaluated in such) and has $|\mathcal{A}| \cdot |\mathcal{B}|$ many states. $\square$

Lemma 2.1.5. *Let $\mathcal{A} = (Q_1, \Sigma, \delta_1, c_1)$ and $\mathcal{B} = (Q_2, \Sigma, \delta_2, c_2)$ be DPAs. $(\mathcal{A}, p) \equiv_{\dagger} (\mathcal{B}, q)$ iff $L(\mathcal{A} \top \mathcal{B}, (p, q)) = \emptyset$.*

Proof. For the first direction of implication, let $L(\mathcal{A} \top \mathcal{B}, (p_0, q_0)) \neq \emptyset$, so there is a word α accepted by that automaton. Let $(p, q)(p_1, q_1)(p_2, q_2) \cdots$ be the accepting run on α. Then $pp_1 \cdots$ and $qq_1 \cdots$ are the runs of $\mathcal{A}$ and $\mathcal{B}$ on α respectively. Whenever $(p_i, q_i) \in F_{\top}$, p_i and q_i have different priorities.

As the run of the product automaton vists infinitely many accepting states, α is a witness for p and q being not priority almost-equivalent.

For the second direction, let p and q be not priority almost-equivalent, so there is a witness α at which infinitely many positions differ in priority. Analogously to the first direction, this means that the run of $\mathcal{A} \intercal \mathcal{B}$ on the same word is accepting and therefore the language is not empty. $\square$

Corollary 2.1.6. *Priority almost equivalence of a given DPA can be computed in quadratic time.*

Proof. By Lemma 2.1.4, we can compute $\mathcal{A} \intercal \mathcal{A}$ in quadratic time. The emptiness problem for deterministic Büchi automata is solvable in linear time by checking reachability of loops that contain a state in F. $\square$

Moore Equivalence

Definition 2.1.6. Let $\mathcal{A} = (Q_1, \Sigma, \delta_1, c_1)$ and $\mathcal{B} = (Q_2, \Sigma, \delta_2, c_2)$ be DPAs. We define *Moore equivalence* as $(\mathcal{A}, p) \equiv_M (\mathcal{B}, q)$ if and only if for all words $w \in \Sigma^*$, $c_1(\delta^*(p, w)) = c_2(\delta^*(q, w))$.

Lemma 2.1.7. $\equiv_M$ *is a congruence relation.*

Proof. It is obvious that $\equiv_M$ is an equivalence relation. For two states $(\mathcal{A}, p) \equiv_M (\mathcal{B}, q)$ and some successors $p' = \delta(p, a)$ and $q' = \delta(q, a)$, it must be true that $(\mathcal{A}, p') \equiv_M (\mathcal{B}, q')$. Otherwise there is a word $w \in \Sigma^*$ such that $c_1(\delta_1^*(p', w)) \neq c_2(\delta_2^*(q', w))$. Then $c_1(\delta_1^*(p, aw)) \neq c_2(\delta_2^*(q, aw))$ and thus $(\mathcal{A}, p) \not\equiv_M (\mathcal{B}, q)$. $\square$

Lemma 2.1.8. *Moore equivalence of a given DPA can be computed in log-linear time.*

Proof. We refer to [2]. The given algorithm can be adapted to Moore automata without changing the complexity. $\square$

Lemma 2.1.9. $\equiv_M \subseteq \equiv_{☥} \subseteq \equiv_L$

Proof. Let $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, q_0^{\mathcal{A}}, \delta_{\mathcal{A}}, c_{\mathcal{A}})$ and $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, q_0^{\mathcal{B}}, \delta_{\mathcal{B}}, c_{\mathcal{B}})$ be two DPA that are priority almost-equivalent and assume towards a contradiction that they are not language equivalent. Due to symmetry we can assume that there is a $w \in L(\mathcal{A}) \setminus L(\mathcal{B})$.

Consider $\alpha = \lambda_{\mathcal{A}}(q_0^{\mathcal{A}}, w)$ and $\beta = \lambda_{\mathcal{B}}(q_0^{\mathcal{B}}, w)$, the priority outputs of the automata on w. By choice of w, we know that $a := \max \mathrm{Inf}(\alpha)$ is even and $b := \max \mathrm{Inf}(\beta)$ is odd. Without loss of generality, assume $a > b$. That means a is seen only finitely often in β but infinitely often in a. Hence, α and β differ at infinitely many positions where a occurs in α. That would mean w is a witness that the two automata are not priority almost-equivalent, contradicting our assumption. $\square$

2.1.2 Representative Merge

Definition 2.1.7. Let $\mathcal{A} = (Q, \Sigma, \delta, c)$ be a DPA and let $\emptyset \neq C \subseteq M \subseteq Q$. Let $\mathcal{A}' = (Q', \Sigma, \delta', c')$ be another DPA. We call $\mathcal{A}'$ a *representative merge of* $\mathcal{A}$ *w.r.t.* M *by candidates* C if it satisfies the following:

- There is a state $r_M \in C$ such that $Q' = (Q \setminus M) \cup \{r_M\}$.
- $c' = c \restriction_{Q'}$.
- Let $p \in Q'$ and $\delta(p, a) = q$. If $q \in M$, then $\delta'(p, a) = r_M$. Otherwise, $\delta'(p, a) = q$.

We call r_M the *representative* of M in the merge. We might omit C and implicitly assume $C = M$.

Definition 2.1.8. Let $\mathcal{A} = (Q, \Sigma, \delta, c)$ be a DPA and let $\mu : D \to (2^Q \setminus \emptyset)$ be a function for some $D \subseteq 2^Q$. If all sets in D are pairwise disjoint and for all $X \in D$, $\mu(X) \subseteq X$, we call μ a *merger function*.

A DPA $\mathcal{A}'$ is a representative merge of $\mathcal{A}$ w.r.t. μ if there is an enumeration $X_1, \ldots, X_{|D|}$ of D and a sequence of automata $\mathcal{A}_0, \ldots, \mathcal{A}_{|D|}$ such that $\mathcal{A}_0 = \mathcal{A}$, $\mathcal{A}_{|D|} = \mathcal{A}'$ and every $\mathcal{A}_{i+1}$ is a representative merge of $\mathcal{A}_i$ w.r.t. X_{i+1} by candidates $\mu(X_{i+1})$.

A common special case of this are quotient automata that are often used in state space reduction. Given a congruence relation $\sim$, the quotient automaton w.r.t. $\sim$ is equivalent to a representative merge w.r.t. $\mu : \mathfrak{C}(\sim) \to 2^Q, \kappa \mapsto \kappa$.

Lemma 2.1.10. *Let $\mathcal{A} = (Q, \Sigma, \delta, c)$ be a DPA and let $\sim$ be an equivalence relation. A representative merge of $\mathcal{A}$ w.r.t. $\sim$ is a representative merge of $\mathcal{A}$ w.r.t. $\mu : \mathfrak{C}(\sim) \to 2^Q, \kappa \mapsto \kappa$.*

The following Lemma formally proofs that this definition actually makes sense, as building representative merges is commutative if the merge sets are disjoint.

Lemma 2.1.11. *Let $\mathcal{A} = (Q, \Sigma, \delta, c)$ be a DPA and let $M_1, M_2 \subseteq Q$. Let $\mathcal{A}_1$ be a representative merge of $\mathcal{A}$ w.r.t. M_1 by some candidates C_1. Let $\mathcal{A}_{12}$ be a representative merge of $\mathcal{A}_1$ w.r.t. M_2 by some candidates C_2. If M_1 and M_2 are disjoint, then there is a representative merge $\mathcal{A}_2$ of $\mathcal{A}$ w.r.t. M_2 by candidates C_2 such that $\mathcal{A}_{12}$ is a representative merge of $\mathcal{A}_2$ w.r.t M_1 by candidates C_1.*

Proof. By choosing the same representative r_{M_1} and r_{M_2} in the merges, this is a simple application of the definition. $\square$

The following Lemma, while simple to prove, is interesting and will find use in multiple proofs of correctness later on.

Lemma 2.1.12. *Let $\mathcal{A}$ be a DPA. Let $\sim$ be a congruence relation on Q and let $M \subseteq Q$ such that for all $x, y \in M$, $x \sim y$. Let $\mathcal{A}'$ be a representative merge of $\mathcal{A}$ w.r.t. M by candidates C. Let ρ and ρ' be runs of $\mathcal{A}$ and $\mathcal{A}'$ on some α. Then for all i, $\rho(i) \equiv \rho'(i)$.*

Proof. We use a proof by induction. For $i = 0$, we have $\rho(0) = q_0$ for some $q_0 \in Q$ and $\rho'(0) = r_{[q_0]_M}$. By choice of the representative, $q_0 \in M$ and $r_{[q_0]_M} \in M$ and thus $q_0 \sim r_{[q_0]_M}$.

Now consider some $i + 1 > 0$. Then $\rho'(i + 1) = r_{[q]_M}$ for $q = \delta(\rho'(i), \alpha(i))$. By induction we know that $\rho(i) \sim \rho'(i)$ and thus $\delta(\rho(i), \alpha(i)) = \rho(i + 1) \sim q$. Further, we know $q \sim r_{[q]_M}$ by the same argument as before. Together this lets us conclude in $\rho(i + 1) \sim q \sim \rho'(i + 1)$. $\square$

The following is a comprehensive list of all relevant merger functions that we use.

- Moore merger μ_M. Defined below.
- Skip merger, $\mu^{\sim}_{\text{skip}}$. Defined in section **??**.

Moore merger

Definition 2.1.9. Let $\mathcal{A}$ be a DPA. The Moore merger μ_M is defined as $\mu_M : \mathfrak{C}(\equiv_M) \to 2^Q, \kappa \mapsto \kappa$.

Lemma 2.1.13. *Let $\mathcal{A}$ be a DPA and let $\mathcal{A}'$ be a representative merge of $\mathcal{A}$ w.r.t. μ_M. Then $\mathcal{A}$ and $\mathcal{A}'$ are Moore equivalent.*

Proof. □

Corollary 2.1.14. *Let $\mathcal{A}$ be a DPA and let $\mathcal{A}'$ be a representative merge of $\mathcal{A}$ w.r.t. μ_M. Then $\mathcal{A}$ and $\mathcal{A}'$ are language equivalent.*

2.1.3 Reachability

Definition 2.1.10. Let $\mathcal{S} = (Q, \Sigma, \delta)$ be a deterministic transition structure. We define the *reachability order* $\preceq^{\mathcal{S}}_{\text{reach}}$ as $p \preceq^{\mathcal{S}}_{\text{reach}} q$ if and only if q is reachable from p.

We want to note here that we always assume for all automata to only have one connected component, i.e. for all states p and q, there is a state r such that p and q are both reachable from r. In practice, most automata have an predefined initial state and a simple depth first search can be used to eliminate all unreachable states.

Lemma 2.1.15. *$\preceq^{\mathcal{S}}_{\text{reach}}$ is a preorder.*

Definition 2.1.11. Let $\mathcal{S} = (Q, \Sigma, \delta)$ be a deterministic transition structure. We call a relation $\preceq$ a *total extension of reachability* if it is a minimal superset of $\preceq^{\mathcal{S}}_{\text{reach}}$ that is also a total preorder.

For $p \preceq q$ and $q \preceq p$, we write $p \simeq q$.

Lemma 2.1.16. *For a given deterministic transition structure $\mathcal{S}$, a total extension of reachability is computable in $\mathcal{O}(|\mathcal{S}|)$.*

Proof. Using e.g. Kosaraju's algorithm **??**, the SCCs of $\mathcal{A}$ can be computed in linear time. We can now build a DAG from $\mathcal{A}$ by merging all states in an SCC into a single state; iterate over all transitions (p, a, q) and add an a-transition from the merged representative of p to that of q. Assuming efficient data structures for the computed SCCs, this DAG can be computed in $O(|\mathcal{A}|)$ time.

To finish the computation of $\preceq$, we look for a topological order on that DAG. This is a total preorder on the SCCs that is compatible with reachability. All that is left to be done is to extend that order to all states. □

Bibliography

[1] Monika Rauch Henzinger and Jan Arne Telle. Faster algorithms for the nonemptiness of streett automata and for communication protocol pruning. In *Algorithm Theory — SWAT'96*, pages 16–27, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[2] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. *An N Log N Algorithm for Minimizing States in A Finite Automaton*, page 15, 01 1971.