Exponential Blow-up in Asynchronous Automata

A Thesis

submitted to
Indian Institute of Science Education and Research Pune
in partial fulfillment of the requirements for the
BS-MS Dual Degree Programme

by

Joshi Varad

Indian Institute of Science Education and Research Pune
Dr. Homi Bhabha Road,
Pashan, Pune 411008, INDIA.

April, 2025

Supervisor: Prakash Saivasan
Co-Supervisor: Anantha Padmanabha

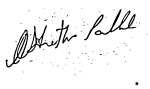
Certificate

This is to certify that this dissertation entitled Exponential Blow-up in Asynchronous Automata towards the partial fulfilment of the BS-MS dual degree programme at the Indian Institute of Science Education and Research, Pune represents study/work carried out by Joshi Varad at Institute of Mathematical Sciences (Chennai) under the supervision of Prakash Saivasan, Assistant Professor, Department of Theoretical Computer Science , Institute of Mathematical Sciences and Anantha Padmanabha, Assistant Professor, Department of Computer Science and Engineering, Indian Institute of Technology Madras, during the academic year 2024-2025.

Prakash Saivasan

Committee:

Prakash Saivasan 

Anantha Padmanabha 

Moumanti Podder 

Declaration

I hereby declare that the matter embodied in the report entitled Exponential Blow-up in Asynchronous Automata are the results of the work carried out by me at the Department of Theoretical Computer Science, Institute of Mathematical Sciences (Chennai), under the supervision of Prakash Saivasan and Anantha Padmanabha and the same has not been submitted elsewhere for any other degree.

Joshi Varad

Acknowledgments

I would like to thank my thesis committee- Prof. Prakash Saivasan, Prof. Anantha Padmanabha and Prof. Moumanti Podder. They have always helped me during my thesis. I thank them for their time and energy.

I thank Prof. Saivasan and Prof. Padmanabha for guiding me for this thesis. It has truly been a pleasurable experience.

I thank Prof. Prafullkumar Tale for guiding my initial studies in mathematical logic and automata theory. I thank Prof. Padmanabha for guiding projects in modal logic before the thesis.

I would like to thank Prof. Anisa Chorwadwala for her timely help during thesis submission. I would like to thank the thesis checking committee for spending their valuable time to grade the thesis.

I would like to thank Indian Institute of Science Education and Research Pune. I thank the Institute of Mathematical Sciences for a nice time at Chennai.

Abstract

Zielonka introduced the concept of asynchronous automata to model some distributed systems [1]. Asynchronous automata are characterized by the languages they accept- regular trace languages [2][3][4]. A minimal DFA accepting a regular trace language is called a Diamond DFA. The characterization of asynchronous automata follows from algorithms converting Diamond DFAs to asynchronous automata accepting the same language [2][3][4]. A natural subject of interest is the lower and upper bounds on the size of minimal asynchronous automata accepting regular trace languages with respect to size of corresponding Diamond DFA and number of processes. The thesis studies upper bounds presented in [2][3][4]. A lower bound on locally rejecting asynchronous automata is studied from [4]. The lower bound result involves a set of regular trace languages- $Path_n$ ($n \in \mathbb{N}$).

The thesis contributes by providing lower and upper bounds on minimal ϵ-non-deterministic asynchronous automata accepting $Path_n$ and by providing lower and upper bounds on minimal register asynchronous automata accepting $Path_n$. The thesis also contains an algorithm converting any asynchronous automata accepting $Path_n$ to a locally rejecting asynchronous automata accepting $Path_n$.

Contents

Chapter 1

Introduction

Distributed systems are systems of computers which can share information by communicating with each other. Distributed systems are used in many applications to compute faster and handle sensitive information. Many distributed systems correspond to a set of DFAs which can communicate with each other when given some input.

Consider an example. There is a building which has 2 groups of elevators- one group which serves only the ground floor and even numbered floors and another group which serves only the ground floor and odd numbered floors. The elevators are super fast and can go from one floor to another instantaneously. The elevator request given by the customer must specify the floor to which he or she wishes to go. Please note that a customer can request a service from second floor to third floor. An 'even floor' elevator can provide the service, but will not accept a customer from the third floor. Since the elevators are 'super fast', the service of one request is executed before another input request is given to the system.

The letters of input alphabet can be thought of as a tuple $(floor_a, floor_b)$ where $floor_a$ is the floor from which the customer requests elevator and $floor_b$ is the floor to which the customer wishes to go.

One way of coding for this elevator system is by creating a 'central computer' which can be represented by a DFA. The central computer will have location information about all elevators. When this computer gets some input $(floor_a, floor_b)$, it will

1. Check whether $floor_a$ is even, odd or ground floor.

2. If even, it will check which 'even floor' elevator is closest to $floor_a$ and send it to fulfil the request. Since the elevators are super fast, it will directly reach $floor_b$. (The central DFA will change its state accordingly).

3. If odd, it will check which 'odd floor' elevator is closest to $floor_a$ and send it to fulfil the request. Since the elevators are super fast, it will directly reach $floor_b$. (The central DFA will change its state accordingly).

4. If ground floor, it will check which elevator is closest to the ground floor and send it to fulfil the request. Since the elevators are super fast, it will directly reach $floor_b$. (The central DFA will change its state accordingly).

Computers execute computations based on inputs received.

If the central computer gets damaged, all elevators will stop. It will be desirable if each elevator has its own computer such that, through communication, the computers will control the elevator system just like the central computer. If there is no restriction on communication and no restriction on what input a computer can read, then one can easily encode the computers for the elevators. However, if there are some restrictions on communication and reading access of input, the encoding is not so easy and sometimes even impossible.

The restrictions for the example are mentioned in this paragraph. 'Even floor' elevators can only read inputs from ground floor and even floors. This is because 'even floor' elevators can not receive requests from odd floors. 'Odd floor' elevators read inputs only from ground floor and odd floors for similar reasons. Two or more computers 'communicate' only when an input readable by the computers is inputted to the system.

The solution to encoding this multi-computer system is discussed now. The computer of an elevator will store its current position. Since the number of floors has to be finite, the possible positions (states) are finite. Hence, the elevator computer can be thought of as a DFA. Given an input, the computers which can read the input will share their information and decide which elevator amongst themselves is closer to the customer. The closer elevator will service the request. This will ensure that the elevator system controlled by

multiple computers behaves exactly the same as the central computer controlled elevator system.

If the system of elevators gets a request $(groundfloor, thirdfloor)$, then all elevators read the input. The elevators share their locations. The closest elevator is sent to serve the customer. Since the elevator is super fast, it will instantaneously reach third floor. (The closest elevator's DFA will change its state accordingly)

If the system of elevators gets a request $(thirdfloor, groundfloor)$, then 'even floor' elevators do not receive the input at all as they do not have reading access for that input. 'Odd floor' elevators receive the input. They share their current location and decide which elevator is closest to the third floor. The closest elevator serves the customer. Since the elevator is super fast, it will instantaneously reach third floor. (The closest elevator's DFA will change its state accordingly)

This 'decentralization' of the central DFA computer was possible in this case. Zielonka proved that this decentralization can be done for DFAs accepting a class of languages called 'regular trace languages'. The idea of a regular trace language is dependent on the restrictions on reading access of inputs and communication of the proposed multi-computer system.

Asynchronous automata mimic the idea of DFAs communicating on some input. An asynchronous automata has a set of processes, which are DFAs. Like most automata, an asynchronous automaton has an input alphabet Σ. Processes have reading access to some letters in the alphabet. The alphabet, along with the reading access information θ of all processes, is termed as a distributed alphabet or distribution denoted by (Σ, θ). An important characteristic of any set of automata is the accepted language. Zielonka proved an important result- all asynchronous automata accept Regular (Σ, θ)-Trace Languages (defined later) and all Regular (Σ, θ)-trace languages can be accepted by an asynchronous automaton. A minimal DFA accepting a Regular (Σ, θ)-trace language is called (Σ, θ)-Diamond Automaton. While proving the Zielonka Theorem, an algorithm was made which converts a (Σ, θ)-Diamond Automaton to an asynchronous automaton with distribution (Σ, θ) accepting the same language. In practical sense, the algorithm converts a single computer program to a distributed programme which has same output as the single computer programme for any input.

Several such algorithms are found [2][3][4]. Since, such algorithms might be practically important, computer scientists have been trying to decrease the storage needed for the resulting asynchronous automaton. The issue with these algorithms is that the output asynchronous automaton is at least exponential in the number of processes [2][3][4].

The algorithms in [2][3][4] output asynchronous automata which solve the 'gossiping problem'. The gossiping problem asks 'which of the processes reading the new input has latest information regarding other processes (which may or may not be reading the input)?'.

Either one of these statements must hold if the exponential blow-up is to be avoided.

1. the gossiping problem is not necessary in general
2. the gossiping problem can be solved without exponential blow-up

Both these statements are unlikely to be true.

An attempt of proving the conjecture uses a set of languages $\{Path_n\}_{n\in\mathbb{N}}$ [4]. The language $Path_n$ for some n considers n many processes. For all pairs of processes p_a and p_b, the alphabet of $Path_n$ has a letter (p_a, p_b) which only p_a and p_b can read. A string $u = u_1...u_k$ of this alphabet is a string in $Path_n$ if for all $i \in [1...k-1]$, there exists a process p_i such that p_i reads both u_i and u_{i+1}.

The size of the minimal DFA accepting $Path_n$ is bounded above by n^2+2. If it is proved that the lower bound of asynchronous automata accepting $Path_n$ is exponential in the number of processes, then the conjecture would be proved.

[4] proves that the lower bound for locally rejecting asynchronous automata for $Path_n$ is exponential in the number of processes- $2^{\frac{n}{4}}$ to be exact.

The thesis discusses the algorithms mentioned in [2][3][4]. The lower bound result [4] is also discussed. New results are-

1. lower and upper bounds for ϵ-non-deterministic asynchronous automata accepting $Path_n$.
2. lower and upper bounds for register asynchronous automata accepting $Path_n$.

3. an algorithm which converts any asynchronous automaton accepting $Path_n$ to a locally rejecting asynchronous automata accepting $Path_n$.

Chapter 2

Preliminaries

Some preliminary results and definitions will be defined in this chapter. Concepts like Asynchronous automata and Mazurkiewicz traces are introduced

2.1 Asynchronous Automaton

Definition 2.1.1. *Asynchronous Automaton [2]*

Given an alphabet Σ*,* $(\Sigma, \theta = \{\Sigma_1 ... \Sigma_n\})$ *is said to be a distribution of* Σ *if* $\bigcup_{m \in \{1...n\}} \Sigma_m = \Sigma$*.*

$loc : \Sigma \to 2^{\{1,2,...,n\}}$ *such that* $loc(a) = \{i \in \{1, 2, ..., n\} | a \in \Sigma_i\}$

Let $(\Sigma, \theta = (\Sigma_1, \Sigma_2 ..., \Sigma_n))$ *be a distribution. For each* $i \in \{1, 2, ..., n\}$*, let* Q_i *be a finite set of states. For each* $a \in \Sigma$*, let* $\Delta_a \subseteq Q_a \times Q_a$ *be the transition relation for* a*, where* $Q_a = \prod_{i \in loc(a)} Q_i$*. The asynchronous automaton defined by this data is the following*

. $Q = Q_1 \times Q_2 \times ... Q_k$

. *Let* $(q_1, q_2, ..., q_n), (q'_1, q'_2, ..., q'_n) \in Q$*. Then* $((q_1, q_2, ..., q_n), (q'_1, q'_2, ..., q'_n)) \in \Delta$ *if*

- *For* $loc(a) = \{i_1, i_2 ... i_k\}, ((q_{i_1}, q_{i_2}, ..., q_{i_k}), (q'_{i_1}, q'_{i_2}, ..., q_{i_k})) \in \Delta_a$
- *For each* $i \notin loc(a), q_i = q'_i$

. $Q_{in} = Q_{in}^1 \times Q_{in}^2 \times ... \times Q_{in}^n$

. $F \subseteq Q_1 \times Q_2 \times ... \times Q_n$

Formally, the asynchronous automaton $\mathcal{A} = (\{Q_i\}_{i \in n}, \Delta, Q_{in}, F, (\Sigma, \theta))$

$\mathcal{A}$ accepts a string u if $\Delta(Q_{in}, u) \cap F \neq \emptyset$

The set $\{1, 2, ..., n\}$ is called the set of processes. It is denoted by $\mathcal{P}$. The motivation for doing so is that each process can be thought to represent a DFA (computer). Unless specifically mentioned otherwise, we consider deterministic asynchronous automata- asynchronous automata where Δ_a is deterministic for all $a \in \Sigma$.

The size of an asynchronous automaton is usually defined as $|\mathcal{A}| = \sum_{i \in [1..n]} |S_{p_i}|$.

Close inspection of the definition shows that each process $p \in \mathcal{P}$ is a DFA which can synchronize with other processes (DFAs).

1. Q_p is the set of states
2. Q_{in}^p are the initial state
3. Δ is like the transition function, but the function output depends on the live state of other processes
4. Local p-states of F is like the accepting set, but the other processes must also agree.

With close observation of the definition of asynchronous automaton, one can see that for two alphabets $a, b \in \Sigma$ such that $loc(a) \cap loc(b) = \emptyset$, an asynchronous automaton will transition to same final state on input ab and ba. This motivates the definition of independence relation.

Definition 2.1.2. *$(\Sigma : \Sigma_1, \Sigma_2, ...\Sigma_n)$ is a distribution. $\mathcal{I}_{loc} \subseteq \Sigma \times \Sigma$ such that $(a, b) \in \mathcal{I}_{loc}$ (said as a and b are independent) if $loc(a) \cap loc(b) = \emptyset$.*

$(\Sigma, \mathcal{I}_{loc})$ is called a concurrent alphabet

The dependence relation $\mathcal{D}_{loc} = (\Sigma \times \Sigma) \backslash \mathcal{I}_{loc}$

The independence relation invokes an equivalence relation on strings. As discussed in a previous paragraph, if $loc(a) \cap loc(b) = \emptyset$, then the final state of input ab and ba will be the same. Shuffling adjacent independent letters causes no change to the final state of input. The following equivalence relation captures the idea.

Definition 2.1.3. *Let $(\Sigma, \mathcal{I})$ be a concurrent alphabet. $\sim \subseteq \Sigma^* \times \Sigma^*$ is the Mazurkiewicz trace relation. For $u, w \in \Sigma^*$, $u \sim w$ if there exists a sequence of strings of Σ such that $u = v_1, v_2, ..., v_k = w$ such that for all $i \in \{1, 2, ..., k-1\}$ there exists $v_i', v_i'' \in \Sigma^*$ and a_i, b_i satisfying*

$$v_i = v_i'.a_ib_i.v_i'', v_{i+1} = v'.b_ia_i.v_i'' \text{ and } (a_i, b_i) \in \mathcal{I}$$

In short, for $u, w \in \Sigma^*$, $u \sim w$ if w can be obtained by a sequence of position swapping of adjacent independent letters.

For a distribution (Σ, θ), if two strings u and w are such that $u \sim w$, then the final global state after transition on u equals the final global state after transition on w. Particularly, the transition on u ends in an accepting state if and only if the transition on w ends in an accepting state. So, asynchronous automata accept languages closed under this relation.

Definition 2.1.4. *A language $L \subseteq \Sigma^*$ is said to be a Regular (Σ, θ)-trace language if*

1. *$u \in L$ if and only if $w \in L$ for all $w \sim u$.*
2. *L is regular.*

Since regular languages have an accepting DFA, the following definitions are natural.

Definition 2.1.5. *If $\mathcal{A}$ is the minimal DFA accepting a Regular (Σ, θ)-trace Language, $\mathcal{A}$ is called a (Σ, θ)-Diamond Automaton.*

If $\mathcal{A}$ is a (Σ, θ)-Diamond Automaton, and the independence relation corresponding to (Σ, θ) is $\mathcal{I}$, then $\mathcal{A}$ is a $(\Sigma, \mathcal{I})$-Diamond automaton (or $\mathcal{I}$-Diamond Automaton if Σ is clear from the context.)

Minimality of (Σ, θ) and $(\Sigma, \mathcal{I})$ Diamond automata gives the following lemma.

Lemma 2.1.1. *Let (Σ, θ) be a distribution and $\mathcal{D} = (S, \delta, q_0, F, \Sigma)$ be a (Σ, θ)-Diamond automaton.*

If $u \sim w$ be words in Σ^*, then $\delta(q, u) = \delta(q, w)$ for all $q \in S$.

Proof. This is a corollary of the following lemma. □

Lemma 2.1.2. *Let (Σ, θ) be a distribution and $\mathcal{D} = (S, \delta, q_0, F, \Sigma)$ be a (Σ, θ)-Diamond automaton.*

If $a, b \in \Sigma$ are independent, then $\delta(q, ab) = \delta(q, ba)$ for all $q \in S$

Proof. This comes from the minimality of $\mathcal{D}$ and the Myhill-Nerode Theorem. The proof is mentioned in [2]. □

The example of letters a and b such that $loc(a) \cap loc(b) = \emptyset$ is referred again. The local states of processes in $loc(a)$ after transition of a, ab and ba are going to be same. Similarly, the local states of processes in $loc(b)$ after transition of b, ab and ba are going to be same. This is because processes in $loc(a)$ are oblivious to b. Similarly, processes in $loc(b)$ are oblivious to a. Assume there was another letter c such that $loc(a) \cap loc(c) \neq \emptyset$ and $loc(b) \cap loc(c) \neq \emptyset$. The local states of processes in $loc(a)$ after transition on bca and ca can be different. Previously, it was as if processes in $loc(a)$ were oblivious to the b. But now, with the addition of c, the occurrence of b can cause a difference.

For a given (Σ, θ) and $u \in \Sigma^*$, a structure can be defined to keep track of which occurrences of letters affect which processes. The structure is called the Mazurkiewicz trace of u for (Σ, θ).

First, the notion of Mazurkiewicz trace will be defined. Then the notion of trace of a word will be defined. Mazurkiewicz traces are partial orders.

Definition 2.1.6. *[2] A Mazurkiewicz trace over $(\Sigma, \mathcal{I})$ is a labelled partial order $T = (\mathcal{E}, \leq, \lambda)$ such that*

1. *$\mathcal{E}$ is called set of events*
2. *$\lambda : \mathcal{E} \to \Sigma$ is a function*

3. *$\leq$ is a partial order on $\mathcal{E}$ satisfying ($e, f \in \mathcal{E}$)*

   (a) *$(\lambda(e), \lambda(f)) \in \mathcal{D}$ implies $e \leq f$ or $f \leq e$*

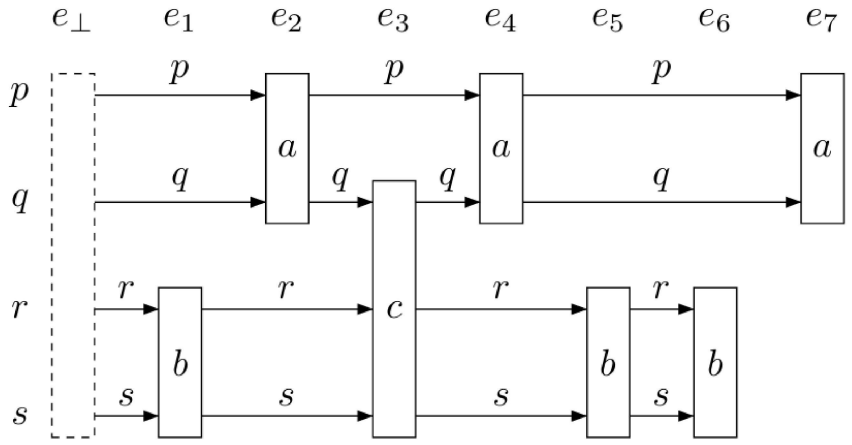
   (b) *$e \lessdot f$ implies $(\lambda(e), \lambda(f)) \in \mathcal{D}$, where $\lessdot$ is the immediate successor relation of $\leq$.*

Now for Mazurkiewicz trace of a string

Definition 2.1.7. *Let $v = v_1 ... v_k$ be a linearization of a Mazurkiewicz trace $T = (\mathcal{E}, \leq, \lambda)$. If $u = \lambda(v_1) ... \lambda(v_k)$, then T is the Mazurkiewicz trace of u.*

Note: The thesis assumes that all processes synchronize before execution of any transition. This is like the initiation of the programme. This zeroth event is called $e_\perp$. $e_\perp \leq e$ for all events of the trace.

An example of a Mazurkiewicz trace is [2]



Example 1.13. Let $\mathcal{P} = \{p, q, r, s\}$ and $\Sigma = \{a, b, c\}$ where $a = \{p, q\}$, $b = \{r, s\}$ and $c = \{q, r, s\}$. Figure 1.3 has another picture of the trace $t = (\mathcal{E}, \leq, \lambda)$ corresponding to the word *bacabba*. The dashed box corresponds to the event $e_\perp$, which we insert at the beginning for convenience.

In the figure, the labelled arrows between the events denote the relations $\lessdot_p$, $\lessdot_q$, $\lessdot_r$ and $\lessdot_s$. From these, we can compute $<$ and $\leq$. Thus, for example, we have $e_1 \leq e_4$ since $e_1 \lessdot_r e_3 \lessdot_q e_4$, and the events e_5 and e_7 are unordered.

The following definitions make future discussion convenient.

Definition 2.1.8. *[2] Let e be an event, p be a process and P be a set of processes*

1. *An event e is a p-event for some $p \in \mathcal{P}$ if $p \in loc(\lambda(e))$*
2. *For $P \subseteq \mathcal{P}$, e is a P-event if $P = loc(\lambda(e))$.*
3. *If e is a P-event, we say the processes in P 'meet', 'participate' or 'synchronize' at e. All processes in P are participants in e.*

For two strings $s_1 \sim s_2$. Let their respective traces be t_1 and t_2. t_1 and t_2 are graph theoretically isomorphic. There are much more similarities between t_1 and t_2. However, understanding the similarities will require more definitions introduced later.

In the thesis, the motivation to define the Mazurkiewicz trace of a string was to answer the question- Given a string u, 'which occurrences of letters of u affect which process'. The following ideas of the trace give us this information.

Definition 2.1.9. *Given a Mazurkiewicz trace T*

1. *The maximum p-event is called $max_p(T)$.*
2. $\partial_p(T) = \{e | e \leq max_p(T)\}$

$\partial_p(T)$ is the set of all events (occurrences of letters in string u) which affect the final local state of p. If $e' \notin \partial_p(T)$, then removing that occurrence of $\lambda(e')$ will not change the final local state of p.

The following conventions are followed

Definition 2.1.10. *Given a Mazurkiewicz trace T*

1. $\partial_\emptyset(T) = \emptyset$
2. *For any $P \subseteq \mathcal{P}, \partial_P(T) = \bigcup\limits_{p \in P} \partial_p(T)$*

The main theorem is the Zielonka's theorem. It is stated.

Theorem 2.1.3. *(Zielonka)[2]*

1. *An asynchronous automaton on (Σ, θ) accepts a regular (Σ, θ)-trace language.*
2. *Let L be a recognizable trace language over a concurrent alphabet $(\Sigma, \mathcal{I})$. Then, for every distributed alphabet (Σ, θ) such that $\mathcal{I}_{loc_\theta} = \mathcal{I}$, we can construct an asynchronous automaton $\mathcal{A}$ over (Σ, θ) with language of $\mathcal{A}$, $\mathcal{L}(\mathcal{A}) = L$*

Proof. of part 1

Proving the first part of Zielonka's theorem does not require many lemmas. Let $\mathcal{A} = (\{S_p\}_{p\in\mathcal{P}}, \Delta, Q_{in}, F, (\Sigma, \theta))$ be a asynchronous automaton.

First, we prove that the language accepted by $\mathcal{A}$ is regular. We prove this by constructing a DFA which accepts same languages. Consider the DFA with set of states $\prod_{p\in\mathcal{P}} S_p$ and the transition function is Δ and accepting states F and initial state Q_{in}. We can prove that this DFA accepts the same language of $\mathcal{A}$. Hence, the language accepted by $\mathcal{A}$ is regular.

Induction on length of u will show that if $u \in \mathcal{L}(\mathcal{A})$, then for all $u' \sim u$, $u' \in \mathcal{L}(\mathcal{A})$

The proof of part 2 is difficult and discussed in 3.1.1. □

2.2 Locally Rejecting Asynchronous Automata

All algorithms which will be mentioned in the thesis create asynchronous automata which are 'locally rejecting'.

Definition 2.2.1. *For a word u and a language $\mathcal{L}$ (it may be that $u \notin \mathcal{L}$)*

1. *$pref_p(u)$ is a linearization of $\partial_p(T_u)$*
2. *$pref(\mathcal{L})$ is the set of all prefixes of all words in $\mathcal{L}$*

Definition 2.2.2. *A deterministic asynchronous automaton $\mathcal{B}$ is called locally rejecting if for every process p, there is a set of states $R_p \subseteq S_p$ such that for every word u:*

$pref_p(u) \notin pref(\mathcal{L}(\mathcal{B}))$ if and only if the p-local state reached by $\mathcal{B}$ on u is in R_p

[4] proves that for regular trace languages $Path_n$ (n stands for number of processes), a locally rejecting asynchronous automaton must have at least $2^{\frac{n}{4}}$ local states. It also proves that for all $n \in \mathbb{N}$, $Path_n$ has a minimal DFA of order n^2. This means that the smallest asynchronous automaton accepting $Path_n$ is exponentially larger than the minimal DFA. Hence, for every algorithm creating locally rejecting automata, the resulting locally rejecting asynchronous automaton will be exponential larger (in number of processes) than the minimal DFA for $Path_n$. We now define $Path_n$

Definition 2.2.3. *Let* $n \in \mathbb{N}, n > 1$

1. $\Sigma = \binom{n}{2}$
2. $\Sigma_{i \in [1...n]} = \{(p_k, p_j) | i = k \vee i = j\}$
3. $u = (p_{k_1}, p_{j_1})(p_{k_2}, p_{j_2})...(p_{k_N}, p_{j_N}) \in \Sigma_n^*$ *is an element of* $Path_n$ *if for every* $i \in [1...N-1]$, $\{p_{k_i}, p_{j_i}\} \cap \{p_{k_{i+1}}, p_{j_{i+1}}\} \neq \emptyset$

Now the proof of the lower bound of locally rejecting asynchronous automata accepting $Path_n$ is presented.

Theorem 2.2.1. *The lower bound of locally rejecting asynchronous automata accepting* $Path_n$ *is* 2^k *where* $k = \lfloor \frac{n}{4} \rfloor$.

Proof. The following proof is for the case where n is a multiple of 4- $4k$. The general case follows closely.

Assume that for some locally rejecting asynchronous automaton accepting $Path_n$, $\mathcal{A}$, there exists some process p which has less than $2^{\frac{n}{4}}$ local states. Without loss of generality, $p = p_n$. (Simple renumbering).

Definition 2.2.4. *For* $m \in [0...k-1]$

1. $a_m = (4m, 4m+1)(4m+1, 4m+2)(4m+2, 4m+4)$
2. $b_m = (4m, 4m+1)(4m, 4m+3)(4m+3, 4m+4)$

The idea is that a_m skips $m+3$ whereas b_m skips and $m+2$

Definition 2.2.5. $P_n = (a_0 + b_0)(a_1 + b_1)...(a_{k-1} + b_{k-1})$ *is the set of strings* $u = c_0...c_{k-1}$ *where* $c_i = a_i$ *or* $c_i = b_i$

There are $2^{\frac{n}{4}}$ many strings in P_n. Hence, by the famous pigeon-hole principle, there must be $u_1 = (c_0^1 c_2^1 ... c_{k-1}^1), u_2 = (c_0^2 c_2^2 ... c_{k-1}^2) \in P_n$ such that the local state of p_n after computation on both strings is same. Call this state s.

Let $d \in [0...k-1]$ be the first moment of difference between u_1 and u_2. To be more formal, let $d \in [0...k-1]$ be such that $c_i^1 = c_i^2$ for all $i < d$ and $c_d^1 \neq c_d^2$

Without loss of generality, $c_d^1 = a_d$ and $c_d^2 = b_d$.

Proposition 2.2.2. *1.* $p_d, p_{d+1}, p_{d+2}, p_{d+4}$ *participate in* u_1 *and not* p_{d+3}

2. $p_d, p_{d+1}, p_{d+3}, p_{d+4}$ *participate in* u_2 *and not* p_{d+2}.

$u_1(p_d, p_{d+3})$ and u_2 will have same p_d view and p_n view. This means that the local state of p_d after computing $u_1(p_d, p_{d+3})$ and u_2 will be same. Same for p_n.

This means that the local state of p_d after computation of $u_1(p_d, p_{d+3})(p_d, p_n)$ will be same as local state after computation of $u_2(p_d, p_n)$. Similarly, this means that the local state of p_n after computation of $u_1(p_d, p_{d+3})(p_d, p_n)$ will be same as local state after computation of $u_2(p_d, p_n)$.

But, $pref_{p_n}(T_{u_1(p_d,p_{d+3})(p_d,p_n)}) \notin pref(Path_n)$. But, $pref_{p_n}(T_{u_2(p_d,p_n)}) \in pref(Path_n)$. Since, $\mathcal{A}$ is locally rejecting, the local state of p_n after $\mathcal{A}$ reads $u_1(p_d, p_{d+3})(p_d, p_n)$ must be in R_n. The local state of p_n after $\mathcal{A}$ reads $u_2(p_d, p_n)$ must not be in R_n. But the local state of p_n in both cases is same. This is a contradiction.

Hence, p_n has more than or equal to $2^{\frac{n}{4}}$ states.

The conjecture is proved for the locally rejecting automata. □

Chapter 3

Gossiping Automata

Most proofs of the Zielonka's theorem require an asynchronous automata solving the gossiping problem. In short, the gossiping problem among participating processes is 'which of the participants has latest information about a process (participating or otherwise)?'. It is formally defined in the next section.

3.1 The Gossiping Problem

Definition 3.1.1. *[2] Let $\mathcal{P} = \{p_1, p_2, ..., p_N\}$ be a set of processes which synchronize with each other from time to time and exchange information about themselves and others. Whenever $P \subseteq \mathcal{P}$ synchronizes, the processes in P must decide amongst themselves which of them has the latest information, direct or indirect, about each process $p \in \mathcal{P}$. We call this the Gossip Problem.*

Given a trace $T = (\mathcal{E}, \leq, \lambda)$, the latest information p has about q is the $\leq$-largest q-event e_q such that $e_q \leq e_p$ where e_p is the $\leq$-largest p-event.

One can create an algorithm solving the gossiping problem. Each process will associate every other process with a timestamp (number). When a set of processes P meet, each process will select a timestamp greater than every timestamp stored by the participants. Each participating process will associate this timestamp with the participants. All

participants also update timestamp associated with non-participants. The timestamp now associated with non-participant q will be the greatest associated timestamp for q among all participants.

For any set of processes P, the process p with largest timestamp associated with q has latest information about q.

Algorithm 1 Algorithm for one of the participating processes p

```
1: Every participating processes fixes m, a number greater than any timestamp remembered by participating processes.
2: for q ∈ 𝒫 do
3:     if q participates then
4:         Timestamp for q set to m
5:     else
6:         Let m′ be greatest timestamp of q any of the participants have
7:         Time stamp of q is updated by all participants to m′
8:     end if
9: end for
```

The issue is the size of the associated timestamps. The size of these numbers can become large and cannot be bounded. Formally, $\forall n \in \mathbb{N}$, there exists input string s such that timestamps used for s are greater than n.

DFAs cannot store unbounded information as the number of states is finite. Similarly, asynchronous automata can't store the infinite information either. Hence, we need a way to be able to use finitely many timestamps. The idea is to re-use timestamps

Having a finite amount of timestamps will work. The remaining part of this section is dedicated to proving this.

In any trace T, for any $p \in \mathcal{P}$, the set of all p-events is totally ordered.

As discussed in 1, $\partial_p(T)$, also called the 'view of p' in T, is the set of events which decide the final local state of p. Absence or presence of any event outside of $\partial_p(T)$ will not affect the final local state of p. One could use $\partial_p(T)$ to timestamp, but just like numbers, the size of information stored will be finite, but unbounded.

The whole $\partial_p(T)$ is not needed to determine which process has latest information about another process. The following information of $\partial_p(T)$ is all that is required.

Definition 3.1.2. *Given a trace T, for all processes $p, q \in \mathcal{P}$, $latest_{p\to q}(T) = max_q(\partial_p(T))$*

This is the latest information process p has about other processes. Assume all the events are timestamped chronologically. The process p with highest timestamped $latest_{p\to q}$ has the latest information about q.

Hence, the idea is that all processes p will store $latest_{p\to q}$ for all q. This will allow processes to select which process has latest information about another process.

Definition 3.1.3. *Primary information for p in T is*

$primary_p(T) = \{(p, q, latest_{p\to q}) | q \in \mathcal{P}\}$

For convenience, sometimes we will ignore the (p, q) and simply consider $primary_p(T) = \{latest_{p\to q} | q \in \mathcal{P}\}$

Definition 3.1.4. *For $P \subseteq \mathcal{P}, primary_P(T) = \bigcup_{p\in P} primary_p(T)$*

Assume that the events are chronologically timestamped and every process caries their finite primary information, then the gossiping problem will be solved.

Note that asynchronous automata can be designed to handle primary information if the timestamp set is finite. The remainder of this chapter will discuss why. Till now we looked at the infinite timestamp set (natural numbers). We introduce primary graphs and secondary information to create and use a viable finite timestamp set.

Definition 3.1.5. *The primary graph of $p \in \mathcal{P}$ in trace $T = (\mathcal{E}, \leq, \lambda)$ is $(V = primary_p(T), E = \{(e, e') | e' \lessdot e\})$*

The primary graph uses E to account for the relation of events in the main trace. To be particular, if event e_1 is descendent of event e_2, then $(e_2, e_1) \in E$

Definition 3.1.6. *For all processes $p, q, r \in \mathcal{P}$, $latest_{p\to q\to r} = max_r(\partial_q(\partial_p(T)))$*

This can be thought of as the latest information of r that q had during the event $latest_{p\to q}(T)$.

Definition 3.1.7. *Secondary information of p in T is*

$$secondary_p(T) = \{(p, q, r, latest_{p\to q\to r})|q, r \in \mathcal{P}\}.$$

For $P \subseteq \mathcal{P}, secondary_P(T) = \bigcup_{p\in P} secondary_p(T)$

The idea is to create a finite re-usable timestamp. The secondary information helps the participating processes decide which of the timestamps is being used in primary information of other processes. This ensures that for a given moment, a label in primary graphs of all processes is unique to a certain event in the trace. The primary graph can then decides which of the participating processes have latest information.

For example [2]

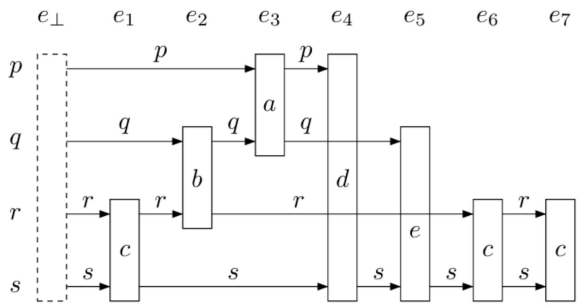


Fig. 1.5. Keeping track of active labels

Example 1.22. Let $\mathcal{P} = \{p, q, r, s\}$ and $\Sigma = \{a, b, c, d, e\}$ where $a = \{p, q\}$, $b = \{q, r\}$, $c = \{r, s\}$, $d = \{p, s\}$ and $e = \{q, s\}$. Figure 1.5 shows the trace $t = (\mathcal{E}, \leq, \lambda)$ corresponding to the word *cbadecc*.

At the end of this word, $e_2 = latest_{p\to r}(\mathcal{E})$, but $e_2 \notin primary_r(\mathcal{E})$, since $primary_r(\mathcal{E}) = \{(r, p, e_4), (r, q, e_5), (r, r, e_7), (r, s, e_7)\}$.

In the example, e_2 is in the secondary information of r. More precisely, $e_2 = latest_{r\to p\to r}(\mathcal{E})$

Surprisingly, this is all the information needed from $\partial_p(T)$. If timestamps are finite, the information is finite. Due to finiteness of this information, asynchronous automaton can store this information. However, designing automaton which can handle this information requires some theorems and definitions.

Definition 3.1.8. *For a trace $(\mathcal{E}, \leq, \lambda)$ an ideal is a set of events $I \subseteq \mathcal{E}$ such that if $e \in I$,*

then for all $f \leq e$, $f \in I$. The maximal elements of an ideal are called its generators.

Lemma 3.1.1. *[2] The maximal events of $\partial_p(I) \cap \partial_q(I)$ are events in $primary_p(I) \cap primary_q(I)$*

Proof. The proof can be found in [2] □

Lemma 3.1.2. *Let I be an ideal and $p, q, r \in \mathcal{P}$. Let $e = latest_{p \to r}(I)$ and $f = latest_{q \to r}(I)$. Then $e \leq f$ if and only if there exists $g \in primary_p(I) \cap primary_q(I)$ such that $e \leq g$. .*

Proof. The proof follows from 3.1. It is also found in [2]

□

Note that 3.1 says that q has latest information of r than p. e will be in primary graph of p and f in the primary graph of q. g will be in both the primary graphs. The $\leq$ relation restricted to primary information is in primary graph. Hence, primary graphs of participating processes can give the information. This is where the primary graph comes in handy.

As discussed before, primary information and chronological timestamping is enough to decide which participant p had latest information about q. The primary graph has the required chronological data about the timestamped events.

Primary graph has more information about $\partial_p(T)$. Let P be set of participating processes. If for every event e, every process p either labels e with fixed timestamp or does not label e, then primary graph of participating process can decide which process has latest information about a process. Please note that the timestamp set need not be an ordered set where greater number means later event. The primary graph will give all required chronology information. The formal proof will be presented later.

Lemma 3.1.3. *Let $\{p_1 ... p_m\}$ be participating processes in the last event of $u.a$. Let p_1 and p_2 be two of the participating processes. If $latest_{p_1 \to q}(u) \leq latest_{p_2 \to q}(u)$, then there exists v in the primary graph of p_1 and p_2 such that $latest_{p_1 \to q}(u) \leq v$ and $latest_{p_2 \to q}(u) \not\leq v$*

Proof. The proof follows from 3.1 □

This shows that two processes can decide which process has latest information. For the general asynchronous automaton, more processes can participate. In that case, the idea is to tell the processes to decide 'latest information' pairwise.

As discussed earlier, it is enough to ensure that events in the primary graph have unique timestamp at a time. The timestamps will look like $(P, L), P \subseteq \mathcal{P}, L \in \mathcal{L}$ where P is the set of participating processes in the event and $\mathcal{L}$ is a set of labels (discussed later).

Secondary information is used to ensure that for each process p, all p-events in $primary_{\mathcal{P}}(I)$ have unique timestamps. The following theorem allows this

Theorem 3.1.4. *Let I be an ideal, $p \in \mathcal{P}$ and e be a p-event in I. If $e \notin secondary_p(I)$, then $e \notin primary_{\mathcal{P}}(I)$.*

$secondary_p(I)$ is like a storage of all 'relevant' events labelled by p. The theorem is a corollary of the following lemma.

Lemma 3.1.5. *Let I be an ideal and $p \in \mathcal{P}$. If $e \in primary_p(I)$ then for every $q \in \mathcal{P}$ such that e is a $q - event$, $e \in secondary_q(I)$.*

Proof. Proof is in [2] □

The set $secondary_{\mathcal{P}}(T) = \{latest_{p \to q \to r}(T) | p, q, r \in \mathcal{P}\}$ has n^3 many elements where $n = |\mathcal{P}|$. Hence, at most n^3 many timestamps can be in the secondary information of processes. Hence, $n^3 + 1$ many timestamps are enough timestamps. [2] also shows that $n^2 + 1$ many timestamps work.

Now the gossiping automaton will be constructed.

3.1.1 Gossiping Automaton

(Note that the automaton constructed here for a concurrent alphabet $(\Sigma, \mathcal{I})$ is not the most efficient gossiping automaton possible. We are considering a distribution (Σ, θ) such that $\mathcal{I} = \mathcal{I}_\theta$)

Definition 3.1.9.

1. $\mathbb{T} = [1...n^3 + 1]$ *is a set of pre-timestamps. ($\mathbb{T}$ is not ordered)*
2. $\mathbb{L} = \mathbb{P}(\mathcal{P}) \times \mathbb{T}$ *where $\mathbb{P}$ denotes power set. This set is the set of timestamps. The set of processes mentioned in the timestamp are the processes which were participating in the event.*
3. $Pr = \{f : \mathcal{P} \to \mathbb{L}\}$
4. $Sec = \{f : \mathcal{P} \times \mathcal{P} \to \mathbb{L}\}$
5. $\mathcal{G} = \{(V, E) | V \subseteq \mathbb{L}, E \subseteq V \times V\}$

Definition 3.1.10. *The Gossiping automaton for a distribution (Σ, θ) is $\mathbb{G} = (\{S_p\}, \Delta, (\Sigma, \theta), q_0)$ such that*

1. $S_p = Pr \times Sec \times \mathcal{G}$ *for all $p \in \mathcal{P}$*
2. *Let $(\mathcal{P}, l_0)$ be the timestamp for $e_\perp$. For process $p \in \mathcal{P}, p_0 = \{(pr : \mathcal{P} \to \{(\mathcal{P}, l_0)\}$ for some $l_0 \in \mathbb{T}, sec : \mathcal{P} \times \mathcal{P} \to \{(\mathcal{P}, l_0)\}, G = (V = \{(\mathcal{P}, l_0)\}, E = \emptyset))\}$*
3. $q_0 = \prod_{p \in \mathcal{P}} p_0$
4. Δ *will basically do the following*
   (a) *If $globalstate = \Delta(Q_0, u)$ for some $u \in \Sigma^*$, $\Delta(globalstate, a) = globalstate'$ where the primary graph, primary information and secondary information are calculated based on previous theorems*
   (b) *If $globalstate \neq \Delta(Q_0, u)$ for any $u \in \Sigma^*$, then $\Delta(globalstate, a)$ doesn't matter to us and can be whatever.*

More specifically, during $\Delta(globalstate, a)$, the following happens-

(Note that subscript $_p$ means that the value belongs to process p and $'$ means updated value. For example sec_p is the sec function of p before transition and sec'_p is the sec function of p after transition)

1. *Participating processes decide a $t \in \mathbb{T}$ such that $(\{p \in loc(a)\}, t)$ is not in any participating process's secondary information. The timestamp of the new event is $(\{p \in loc(a)\}, t)$.*

For every participating process p, the primary information $pr_p(q)$ is changed to $(\{p \in loc(a)\}, t)$ for all participating process q.

2. *Participating processes pairwise decide who has the latest information about a non-participating process. How do a pair of processes p_1, p_2 decide which has latest information about non-participant q?*
   (a) *Check the primary graph*
   (b) *If $pr_{p_1}(q) \geq pr_{p_2}(q)$, then $\exists g \in G_1 \cap G_2$ such that $pr_{p_1} \not< g$ and $pr_{p_2} \leq g$*
   (c) *(Similarly for $pr_{p_2}(q) \geq pr_{p_1}(q)$)*
   (d) *Decide who has latest information about q*
3. *Consider all the latest information now. The updated graph is $G' = (V', E')$ (same for every process) such that*
   (a) *V' is the updated primary information*
   (b) *$E' = (\bigcup_{p \in loc(a)} E_p|_{V' \cap V_p}) \cup \{((\{p \in loc(a)\}, t), m) | m$ is maximal without this edge$\}$*
4. *For every participant p and non-participant q, and process r, $sec'_p(q, r) = sec_{p_{latest}}(q, r)$ where p_{latest} is the process which had the latest information of q before the primary information was updated.*
5. *For every participant p, q, and every process r, $sec'_p(q, r) = pr'_q(r)$*

Chapter 4

Proofs of Zielonka's theorem

The paper discusses 3 proofs of Zielonka's theorem-

1. Residue Method [2]
2. Zone Method [3]
3. Updated Zone Method [4]

All these methods need the gossiping automaton. Formally, all these methods update the primary information, primary graph and secondary information. Each proof outputs a locally rejecting asynchronous automaton accepting the same language as input DFA. Roughly, as one goes from first algorithm to last algorithm, the resulting asynchronous automaton will be smaller.

4.1 Residue Method

Given $B = (S_B, q_{B_0}, F_B, \delta_B)$ be a DFA accepting regular $(\Sigma, \mathcal{I})$-trace language l and (Σ, θ) such that $\mathcal{I}_\theta = \mathcal{I}$. The plan is to create an asynchronous automaton A which accept the same language as B. For every $u \in \Sigma^*$, A keeps track of $\delta_b(q_{B_0}, u')$ for some $u' \sim u$. $\delta_b(q_{B_0}, u') = \delta_b(q_{B_0}, u)$

[2] paper is discussed in this section. First, we define the notion of projection

Definition 4.1.1. *A string $u \in \Sigma^*$ can be thought of as a function $u : [1...n] \to \Sigma$ where n is the length of u. The trace of u is $[u] = (\mathcal{E}, \leq, \lambda)$. Without loss of generality, $\mathcal{E} = \{e_\perp, e_1, e_2, ..., e_n\}$ such that $\lambda(e_i) = u(i)$. Let $X \subseteq \mathcal{E}$ where $X \backslash \{e_\perp\} = \{e_{i_1}, e_{i_2}, ..., e_{i_k}\}$ with $i_1 < i_2 < ... < i_k$. Then $u[X] = u(i_1)u(i_2)...u(i_k)$. If $X \backslash \{e_\perp\} = \emptyset$, then $u[X] = \epsilon$, the empty string.*

Let $|\mathcal{P}| = n$. Without loss of generality, $\mathcal{P} = \{p_1, p_2, ..., p_N\}$.

The definition of ideals is extended to ensure that the empty set $\emptyset$ is also an ideal.

[2] tries to ensure that for each trace T, p_m stores the information of the transition function $f_m : S_B \to S_B$ such that $f_m(s) = \delta_B(s, u[\partial_{p_m}(T) \backslash \partial_{\{p_1, p_2, ..., p_{m-1}\}}(T)])$ for all $s \in S_B$. Then the transition is accepted by A if $f_n \circ f_{n-1} \circ ... \circ f_1(q_{B_0}) \in F_B$.

The reason why this works is the following theorem.

Theorem 4.1.1. *Let u be a word $[u] = (\mathcal{E}, \leq, \lambda)$ and $I_1 \subseteq I_2 \subseteq ... \subseteq I_k \subseteq \mathcal{E}$ a sequence of ideals. Then $u[I_k] \sim u[I_1]u[I_2 \backslash I_1]...u[I_k \backslash I_{k-1}]$.*

This theorem says that

$$u = u[\mathcal{E}] \sim u[\partial p_1(\mathcal{E})]u[\partial\{p_2, p_2\}(\mathcal{E}) \backslash \partial_{p_1}(\mathcal{E})]... \\ u[\partial\{p_1, p_2, ..., p_n\}(\mathcal{E}) \backslash \partial_{\{p_1, p_2, ..., p_{n-1}\}}(\mathcal{E})]$$

Due to definition of B, this means

$\delta_B(q_0, u) = \delta_B(u[\partial p_1(\mathcal{E})]u[\partial\{p_2, p_2\}(\mathcal{E}) \backslash \partial_{p_1}(\mathcal{E})]...u[\partial\{p_1, p_2, ..., p_n\}(\mathcal{E}) \backslash \partial_{\{p_1, p_2, ..., p_{n-1}\}}(\mathcal{E})]) = f_n \circ f_{n-1} \circ ... \circ f_1(q_{B_0})$

The theorem is a corollary to the lemma

Lemma 4.1.2. *Let u be a word, $[u] = (\mathcal{E}, \leq, \lambda)$, the corresponding trace, and $I, J \subseteq \mathcal{E}$ be ideals such that $I \subseteq J$. Then $u[J] \sim u[I]u[J \backslash I]$*

Proof. The proof is in [2] □

Some calculation gives us

$u[\partial_{\{p_1...p_j\}}(\mathcal{E})\backslash\partial_{\{p_1..p_{j-1}\}}] = u[\partial_{p_j}(\mathcal{E})\backslash\partial_{\{p_1...p_{j-1}\}}(\mathcal{E})]$

Intuitively, $u[\partial_{\{p_1...p_j\}}(\mathcal{E})\backslash\partial_{\{p_1..p_{j-1}\}}] = u[\partial_{p_j}(\mathcal{E})\backslash\partial_{\{p_1...p_{j-1}\}}(\mathcal{E})$ is the set of events p_j can view, but $\{p_1...p_{j-1}\}$ cannot view.

$\partial_{\{p_1...p_{j-1}\}}(\mathcal{E})$ is an ideal. The thesis will use the notion of 'what a process p can see barring an ideal' many times. This motivates the following definition.

Definition 4.1.2. *Let $u \in \Sigma^*$ be a word whose associated trace is $[u] = (\mathcal{E}, \leq, \lambda)$, $I \subseteq \mathcal{E}$ an ideal and $p \in \mathcal{P}$. $\mathcal{R}(u, p, I) = u[\partial_p(\mathcal{E})\backslash I]$. This is called the residue of u at p with respect to I.*

Keen observation uncovers that $u[\partial_{p_n}(T)\backslash\partial_{\{p_1,p_2,...,p_{n-1}\}}(T)] = \mathcal{R}(u, p_n, \partial_{\{p_1,p_2,...,p_{n-1}\}}(T)$

Definition 4.1.3. *A residue $\mathcal{R}(u, p, I)$ is called a process residue if $\mathcal{R}(u, p, I) = (u, p, \partial_P(\mathcal{E}))$ for some $P \subseteq \mathcal{P}$*

If observed carefully, $u[\partial_{\{p_1,p_2,...,p_m\}}(\mathcal{E})\backslash\partial_{\{p_1,p_2,...,p_{m-1}\}}(\mathcal{E})] = \mathcal{R}(u, p_m, \partial_{\{p_1,p_2,...,p_{m-1}\}}(\mathcal{E}))$. But there is an issue.

Say an automaton can store for all process residues R, the function $f_R : S_B \to S_B$ such that $f_R(s) = \delta_B(s, R)$. The accepting states can be the global states where $f_n \circ ... f_1(q_0) \in F_B$ where $f_i = \delta(s, \mathcal{R}(u, p_i, \partial_{\{p_1...p_{i-1}\}}))$.

This will work as $u \sim u[\partial p_1(\mathcal{E})]u[\partial\{p_2, p_2\}(\mathcal{E})\backslash\partial_{p_1}(\mathcal{E})]...$
$u[\partial\{p_1, p_2, ..., p_N\}(\mathcal{E})\backslash\partial_{\{p_1,p_2,...,p_{N-1}\}}(\mathcal{E})]$, and hence $\delta(q_0, u) \in F_B$ if and only if $f_n \circ ... f_1(q_0) \in F_B$.

For a string $u \in \Sigma^*$ and $a \in \Sigma$, let $[u] = (\mathcal{E}_u, \leq_u, \lambda_u)$ and $[ua] = (\mathcal{E}_{ua}, \leq_{ua}, \lambda_{ua})$ be the traces of u and ua. Even if $p \in \mathcal{P}$ doesn't participate in the new event of $[ua]$, it can happen that for some $P \subseteq \mathcal{P}$, $\mathcal{R}(u, p, \partial_P(\mathcal{E}_u)) \neq \mathcal{R}(ua, p, \partial_P(\mathcal{E}_{ua}))$. This intuitively means that p might not know all the instantaneous processes residues for a given trace. This means at a given moment, some processes might not have correct instantaneous process residues.

So, we introduce a concept which only changes for a process p if p participates in an event- the primary residue.

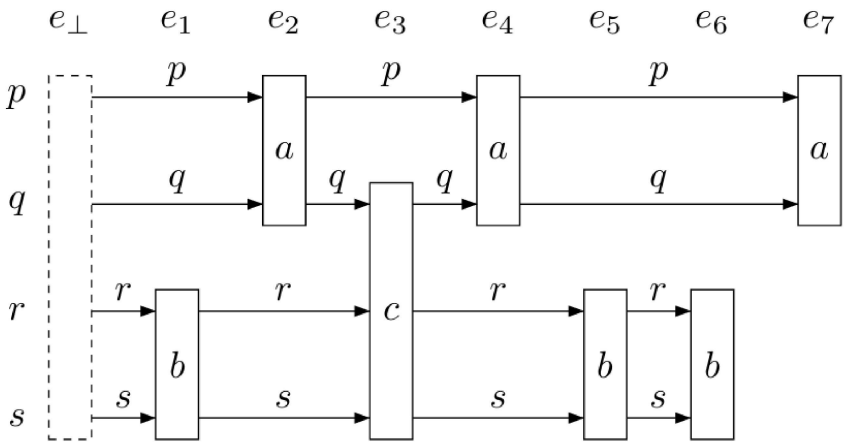
Definition 4.1.4. *$\mathcal{R}(u, p, I)$ is a primary residue if I is generated by a subset of $primary_p(T_u)$.*

For each primary residue R of a process p, p stores the function $f_R : S_B \to S_B$ such that $f_R(s) = \delta_B(s, R)$

If an asynchronous automaton is designed properly, every process p can update their data to store the functions associated with instantaneous primary residues. One can prove that all process residues are primary residues. Note that process residues of p can change without participation of p, but primary residues cannot. The next paragraph explains why.

$\mathcal{R}(u, p, \partial_P(\mathcal{E})) = \mathcal{R}(u, p, E))$ where E is generated by $\pi \subseteq primary_p(\mathcal{E})$. The issue is that when the new event does not involve p, p can't update which subset π generates E. So, the E changes and p might not be aware of it unless and until p participates in some event. But, the primary residues stay the same.

For example [2]



In the example, $\mathcal{R}(u, s, \partial_p(\mathcal{E}_u))$, a process residue is equal to primary residue $\mathcal{R}(u, s, \downarrow \{latest_{s \to q}(\mathcal{E}_u)\})$.

Unless p participates, its primary residues do not change- and hence the functions of the primary residues stay same. Once it participates, it can update its primary residue functions.

We would like to update primary information functions. This requires some lemmas.

Lemma 4.1.3. *[2]Let $u \in \Sigma^*$ and $p \in \mathcal{P}$*

1. *For ideals $I, J \subseteq \mathcal{E}_u$, let $\mathcal{R}(u,p,I)$ and $\mathcal{R}(u,p,J)$ be primary residues such that $\mathcal{R}(u,p,I) = \mathcal{R}(u,p,\downarrow E_I)$ and $\mathcal{R}(u,p,J) = \mathcal{R}(u,p,\downarrow E_J)$ for $E_I, E_J \subseteq primary_p(\mathcal{E}_u)$. Then $\mathcal{R}(u,p,I \cup J)$ is also a primary residue and $\mathcal{R}(u,p,I \cup J) = \mathcal{R}(u,p,\downarrow (E_I \cup E_J))$.*

2. *Let $Q \subseteq \mathcal{P}$. Then $\mathcal{R}(u,p,\partial_Q(\mathcal{E}_u))$ is a primary residue $\mathcal{R}(u,p,\downarrow E)$ for p. Further, p can effectively compute the set $E \subseteq primary_p(\mathcal{E}_u)$ from the information in $primary_{\{p\}\cup Q}(\mathcal{E}_u)$*

3. *Let $q, r \in \mathcal{P}$ such that $latest_{p\to r}(\mathcal{E}_u) \leq latest_{q\to r}$. Then $\mathcal{R}(u,p,\partial_r(\partial_q(\mathcal{E}_u)))$ is a primary residue $\mathcal{R}, p, \downarrow E)$ for p. Further p can effectively compute the set $E \subseteq primary_p(\mathcal{E}_u)$ from the information in $primary_p(\mathcal{E}_u)$ and $secondary_q(\mathcal{E}_u)$.*

Proof. [2]

1. We can rewrite $\mathcal{R}(u,p,I \cup J)$ as $\mathcal{R}(u,p,\partial_p(\mathcal{E}_u) \cap (I \cup J))$. But $\partial_p(\mathcal{E}_u) \cap (I \cup J) = (\partial_p(\mathcal{E}_u) \cap I) \cup (\partial_p(\mathcal{E}_u) \cap J)$. Since $\mathcal{R}(u,p,I) = \mathcal{R}(u,p,\partial_p(\mathcal{E}_u) \cap I)$, we know that $\partial_p(\mathcal{E}_u) \cap I$ is generated by E_I. Similarly, $\partial_p(\mathcal{E}_u) \cap J$ is generated by E_J. So, $\downarrow (E_I \cup E_J) = (\partial_p(\mathcal{E}_u) \cap I) \cup (\partial_p(\mathcal{E}_u) \cap J)$. Therefore, $E_I \cup E_J$ generates $\partial_p(\mathcal{E}_u) \cap (I \cup J)$ and so the residue $\mathcal{R}(u,p,I \cup J) = \mathcal{R}(u,p,\downarrow (E_I \cup E_J))$

2. Let $Q = \{q_1, q_2, ..., q_k\}$. We can rewrite $\mathcal{R}(u,p,\partial_Q(\mathcal{E}_u))$ asn $\mathcal{R}(u,p,\bigcup_{i\in[1...k]} \partial_{q_i}(\mathcal{E}_u))$. From previous lemma, it follows that for each $i \in [1...k]$, p can compute a set $E_i \subseteq primary_p(\mathcal{E}_u)$ from the information in $primary_{p,q_i}(\mathcal{E}_u)$ such that $\mathcal{R}(u,p,\partial_{q_i}(\mathcal{E}_u)) = \mathcal{R}(u,p,\downarrow E_i)$. From part 1 of this corollary, it then follow that $\mathcal{R}(u,p,\partial_Q(\mathcal{E}_u)) = \mathcal{R}(u,p,\bigcup_{i\in[1...k]} \partial_{q_i}(\mathcal{E}_u)) = \mathcal{R}(u,p,\downarrow E)$ where $E = \bigcup_{i\in[1...k]} E_i$.

3. Let $J = \partial_p(\mathcal{E}_u) \cup \partial_r(\partial_q(\mathcal{E}_u))$. J is an ideal. By the construction of J, $max_p(J) = max_p(\mathcal{E}_u)$. From the assumption that $latest_{p\to r}(\mathcal{E}_u) \leq latest_{q\to r}(\mathcal{E}_u)$, we have $max_r(J) = latest_{q\to r}(\mathcal{E}_u)$. So, $\partial_p(J) = \partial_p(\mathcal{E}_u)$ and $\partial_r(J) = \partial_r(\partial_q(\mathcal{E}_u))$.
Since $\mathcal{R}(u,p,\partial_r(\partial_q(\mathcal{E}_u)) = \mathcal{R}(u,p,\partial_p(\mathcal{E}_u) \cap (\partial_r(\partial_q(\mathcal{E}_u))) = \mathcal{R}(u,p,\partial_p(J) \cap \partial_r(J))$, it suffices to find a subset $E \subseteq primary_p(\mathcal{E}_u)$ which generates $\partial_p(J) \cap \partial_r(J)$. By previous lemma, $\partial_p(J) \cap \partial_r(J)$ is generated by $primary_p(J) \cap primary_r(J)$. Since $max_p(J) = max_p(\mathcal{E}_u)$, $primary_p(J) = primary_p(\mathcal{E}_u)$.

On the other hand, $primary_r(J) = primary_r(\downarrow latest_{q \to r}(\mathcal{E}_u))$. By definition, this is the set $\{latest_{q \to r \to s}(\mathcal{E}_u)\}_{s \in \mathcal{P}}$. So the set $E \subseteq primary_p(\mathcal{E}_u)$ generating $\partial_p(J) \cap \partial_r(J)$ is given by $E = primary_p(J) \cap primary_r(J) = primary_p(\mathcal{E})u) \cap \{latest_{q \to r \to s}(\mathcal{E}_u)\}_{s \in \mathcal{P}}$ and can be computed from $primary_p(\mathcal{E}_u)$ and $secondary_q(\mathcal{E}_u)$

□

Using this information, it is proved that primary residue function can be updated locally.

The essence is as follows

1. Initially, every primary residue of every process is the empty string ϵ. So the primary residue function for every residue of every process is $Id : S_B \to S_B$ because $\delta(s \in S_B, \epsilon) = s$ for all states. For all processes p, q, r, $latest_{p \to q} = e_\perp$ and $latest_{p \to q \to r} = e_\perp$.
2. When the asynchronous automaton reads a letter $a \in \Sigma$, the participating processes update their primary residue functions based on the primary information, secondary information and currently stored primary residue functions. (based on upcoming lemma)

If $p \notin loc(a)$, then p's primary residue- and hence primary residue functions-, primary information and secondary information remain unchanged. So, p doesn't need to update anything.

The asynchronous automaton must be able to do all the updates. From the gossip automaton proof, it is evident that updating the primary and secondary information is possible. Now it must be proved that primary residue functions can be updated.

Lemma 4.1.4. *Let $p \in loc(a)$ and $E \subseteq primary_p(\mathcal{E}_{ua})$. The function f_w corresponding to the primary residue $w = \mathcal{R}(ua, p, \downarrow E)$ can be computed from the primary residues of processes in $loc(a)$ at u using the information in $primary_{loc(a)}(\mathcal{E}_u)$ and $secondary_{loc(a)}(\mathcal{E}_u)$.*

Proof. e_a is the event in $\mathcal{E}_{ua}$ associated with the last a.

Case 1- $e_a \in E$

Hence $\mathcal{R}(ua, p, \downarrow E) = \mathcal{R}(ua, p, \downarrow e_a)$ is empty. Hence, $f_w = Id$.

Case 2- $e_a \notin E$

The aim is to compute f_w for $w = \mathcal{R}(ua, p, \downarrow E)$. $f_w = f'_w$ for some $w' \sim w$.

$ua[\partial_p(\mathcal{E}_{ua})] \sim ua[\downarrow E]ua[\partial_p(\mathcal{E}_{ua})\backslash \downarrow E] \sim u[\downarrow E]ua[\partial_p(\mathcal{E}_u)\backslash \downarrow E]$ as $ua[\downarrow E] = u[\downarrow E]$

Also,

$ua[\partial_p(\mathcal{E}_{ua})] \sim u[\partial_{loc(a)}(\mathcal{E}_u)]a \sim u[\downarrow E]u[\partial_{loc(a)}(\mathcal{E}_u)\backslash \downarrow E]a$

This yields $u[\downarrow E]ua[\partial_p(\mathcal{E}_u)\backslash \downarrow E] \sim u[\downarrow E]u[\partial_{loc(a)}(\mathcal{E}_u)\backslash \downarrow E]a$. This in turn means that $ua[\partial_p(\mathcal{E}_u)\backslash \downarrow E] \sim u[\partial_{loc(a)}(\mathcal{E}_u)\backslash \downarrow E]a$

The transition function of B corresponding to $u[\downarrow E]$ is already known as it is previous primary residue. It is enough to find the transition function for $u[\partial_{loc(a)}(\mathcal{E}_u)\backslash \downarrow E]a$.

In short, the w' chosen is $u[\partial_{loc(a)}(\mathcal{E}_u \backslash \downarrow E]a$

Let $loc(a) = \{p_1, p_2, ..., p_k\}$ Define $Q_i = \{p_1, p_2, ..., p_i\}$ for $i \in [1...k]$. Define $I_0 =\downarrow E$. For $i \in [1...k]$, $I_i = I_0 \cup \partial_{Q_i}(\mathcal{E}_u)$

Since $E \subseteq \partial_{loc(a)}(u)$, $u[I_k] = u[\partial_{loc(a)}(\mathcal{E}_u)]$. By previous lemma, $u[\partial_{loc(a)}(\mathcal{E}_u)] = u[I_k] = u[I_0]u[I_1\backslash I_0]...I_k\backslash I_{k-1}]$. This means that
$ua[\partial_p(\mathcal{E}_u)\backslash \downarrow E] \sim u[\partial_p(\mathcal{E}_u)\backslash \downarrow E]a \sim u[I_1\backslash I_0]...I_k\backslash I_{k-1}]$.

It is enough to prove that $u[I_i\backslash I_{i-1}]$ can be computed by using the primary and secondary information of processes in $loc(a)$ along with the primary residues. This is what the rest of the proof does.

To make the statement more precise, it is shown that $u[I_i\backslash I_{i-1}]$ is a primary residue $\mathcal{R}(u, p_i, \downarrow F_i)$ and F_i can be calculated using primary, secondary and residue information of participants.

Initially, the statement is proved for $i = 1$. Later it is generalized to all $i \in [1...k]$.

$E_i = E \cap primary_{p_i}(\mathcal{E}_u)$. Please note that it is $\mathcal{E}_u$ and not $\mathcal{E}_{ua}$.

This means that $E \backslash E_1$ is $E \cap \bigcup_{k \in [2...k]} primary_{p_k}(\mathcal{E}_u)$ (because $e_a \notin E$). If $(p', q', e) \in E \backslash E_i$ means that $latest_{p \to q'}(\mathcal{E}_u) < latest_{p' \to q'}(\mathcal{E}_u)$.

$E' = E \backslash E_1$

$u[I_1 \backslash I_0] = u[\partial_p(\mathcal{E}_u) \backslash \downarrow E] = u[\partial_p(\mathcal{E}_u) \backslash (\downarrow E' \cup \downarrow E_1)]$

Let $e \in E'$. $e = latest_{p_i \to r}(\mathcal{E}_u)$ for some $i \in [2...k]$. $\mathcal{R}(u, p, \downarrow e) = \mathcal{R}(u, p, \partial_r(\partial_{p_i}(\mathcal{E}_u))$. By previous lemma, $\mathcal{R}(u, p, \partial_r(\partial_{p_i}(\mathcal{E}_u))) = \mathcal{R}(u, p, \downarrow G_e)$ for some $G_e \subseteq primary_p(\mathcal{E}_u)$. G_e can be computed from $primary_p(\mathcal{E}_u)$ and $secondary_{p_i}(\mathcal{E}_u)$.

$u[I_1 \backslash I_0] = u[\partial_{p_1}(\mathcal{E}_u) \backslash \downarrow E] = u[\partial_p(\mathcal{E}_u) \backslash (\downarrow E' \cup \downarrow E_1)] = \mathcal{R}(u, p_1, \downarrow E' \cup \downarrow E_1) = \mathcal{R}(u, p_1, \downarrow E_1 \cup \bigcup_{e \in E'} \downarrow G_e) = \mathcal{R}(u, p_1, \downarrow (E_1 \cup \bigcup_{e \in E'} G_e))$.

Now it is proved for general $v \in [1...k]$. The idea is the same.

$E' = E \backslash E_v$

$u[I_v \backslash I_{v-1}] = u[\partial_{p_v}(\mathcal{E}_u) \backslash (\downarrow E \cup \bigcup_{q \in [p_1, ..., p_{v-1}]} \downarrow \partial_q(\mathcal{E}_u))] = u[\partial_{p_v}(\mathcal{E}_u) \backslash (\downarrow E' \cup \downarrow E_v \cup \bigcup_{q \in [p_1, ..., p_{v-1}]} \downarrow \partial_q(\mathcal{E}_u))]$

Let $e \in E'$. $e = latest_{p_i \to r}(\mathcal{E}_u)$ for some $i \neq v$. $\mathcal{R}(u, p_v, \downarrow e) = \mathcal{R}(u, p, \partial_r(\partial_{p_i}(\mathcal{E}_u)))$. By previous lemma, $\mathcal{R}(u, p_v, \partial_r(\partial_{p_i}(\mathcal{E}_u))) = \mathcal{R}(u, p_v, \downarrow G_e)$ for some $G_e \subseteq primary_{p_v}(\mathcal{E}_u)$. G_e can be computed from $primary_{p_v}(\mathcal{E}_u)$ and $secondary_{p_i}(\mathcal{E}_u)$

$\mathcal{R}(u, p_v, \downarrow \partial_q(\mathcal{E}_u))$. By the previous lemma, the generator can be computed from $primary_{\{q, p_v\}}(\mathcal{E}_u)$. Call the generator F_q

$u[I_v \backslash I_{v-1}] = u[\partial_{p_v}(\mathcal{E}_u) \backslash \downarrow I_{v-1}] = u[\partial_{p_v}(\mathcal{E}_u) \backslash \downarrow E' \cup \downarrow E_v \bigcup_{i \in [v...k]} \partial_{p_i}(\mathcal{E}_u)] = \mathcal{R}(u, p_v, \downarrow E' \cup \downarrow E_v \bigcup_{p \in [p_{v+1} ... p_k]} \partial_p(\mathcal{E}_u)) = \mathcal{R}(u, p_v, \downarrow E_1 \bigcup_{e \in E'} \downarrow G_e \bigcup_{q \in [p_{v+1} ... p_k]} \downarrow F_q)$ □

The Accepting Asynchronous Automata

Given a minimal DFA accepting $\mathcal{L}$. $\mathcal{B} = (\mathcal{S}, q_0, \mathcal{F}, \delta)$. $\mathcal{A} = (\{S_p\}_{p \in \mathcal{P}}, \Delta, Q_0, F, (\Sigma, \theta))$

1. The set of states of p is $\{(a, b, c, \bigcup_{P \in \mathbb{P}(\mathcal{P})} \{f_P\}\}$ where

(a) a is primary information stored in the form $\{(p,q,l)|latest_{p\to q}$ event is labelled with $l\}$

(b) b is the secondary information stored in the form $\{p,q,r,l|latest_{p\to q\to r}$ event is labelled with $l\}$

(c) c is primary graph corresponding to a

(d) $f_P : \mathcal{S} \to \mathcal{S}$ such that $f_P(s) = \delta(s, \mathcal{R}(u,p,\partial_P(\mathcal{E}_u)))$

2. The initial state $Q_0 = (\{(p,q,e_\perp)|q \in \mathcal{P}\}, \{(p,q,r,e_\perp)|q,r \in \mathcal{P}\}, \underset{P\in\mathbb{P}(\mathcal{P})}{\cup}\{f_P = Id\})$

3. $F = \{\prod\limits_{i\in[1\ldots n]}(a^{p_i}, b^{p_i}, \underset{P\in\mathbb{P}(\mathcal{P})}{\cup}\{f_P^{p_i}\})|f^{p_n}_{\{p_1\ldots p_{n-1}\}} \circ f^{p_{n-1}}_{\{p_1\ldots p_{n-2}\}}\ldots f^{p_1}_{\emptyset}(q_0) \in \mathcal{F}$

4. $\Delta(((a^{p_1}, b^{p_1}, \underset{P\in\mathbb{P}(\mathcal{P})}{\cup}\{f_P^{p_1}\})\ldots(a^{p_n}, b^{p_n}, \underset{P\in\mathbb{P}(\mathcal{P})}{\cup}\{f_P^{p_n}\})), l \in \Sigma) =$
$((a'^{p_1}, b'^{p_1}, \underset{P\in\mathbb{P}(\mathcal{P})}{\cup}\{f_P'^{p_1}\})\ldots(a'^{p_n}, b'^{p_n}, \underset{P\in\mathbb{P}(\mathcal{P})}{\cup}\{f_P'^{p_n}\}))$ where

(a) If $i \notin loc(l), ((a'^{p_i}, b'^{p_i}, \underset{P\in\mathbb{P}(\mathcal{P})}{\cup}\{f_P'^{p_i}\}) = (a^{p_i}, b^{p_i}, \underset{P\in\mathbb{P}(\mathcal{P})}{\cup}\{f_P^{p_i}\}))$

(b) For $i \in loc(l)$,

i. a'^{p_i} is the updated primary graph of p_i

ii. b'^{p_i} is the updated secondary information of p_i

iii. $f_P'^{p_i}$ are the updated functions

The set F is chosen such that $f^{p_n}_{\{p_1\ldots p_{n-1}\}} \circ f^{p_{n-1}}_{\{p_1\ldots p_{n-2}\}}\ldots f^{p_1}(q_0) \in \mathcal{F}$. By a theorem proved before, for an input $u[\mathcal{E}_u] \sim u[\partial_{p_1}(\mathcal{E}_u)]u[\partial_{p_2}(\mathcal{E}_u)\backslash\partial_{p_1}(\mathcal{E}_u)\ldots u[\partial_{p_n}(\mathcal{E}_u)\backslash\partial_{\{p_1\ldots p_{n-1}\}}]$.

$\delta(q_0,u) \sim \delta(q_0, u[\partial_{p_1}(\mathcal{E}_u)]u[\partial_{p_2}(\mathcal{E}_u)\backslash\partial_{p_1}(\mathcal{E}_u)\ldots u[\partial_{p_n}(\mathcal{E}_u)\backslash\partial_{\{p_1\ldots p_{n-1}\}}]) =$
$f^{p_n}_{\{p_1\ldots p_{n-1}\}} \circ f^{p_{n-1}}_{\{p_1\ldots p_{n-2}\}}\ldots f^{p_1}(q_0)$

And hence, $\delta(q_0,u) \in \mathcal{F}$ if and only if $f^{p_n}_{\{p_1\ldots p_{n-1}\}} \circ f^{p_{n-1}}_{\{p_1\ldots p_{n-2}\}}\ldots f^{p_1}(q_0) \in \mathcal{F}$

4.1.1 The Locally Rejecting States

Let $\mathcal{D}$ be the input DFA as mentioned above and $\mathcal{A}$ be the corresponding output asynchronous automaton.

Lemma 4.1.5. $\mathcal{R}(u,p,\emptyset) \sim pref_p(u)$

For all processes p, consider the local p-states where $f_{\mathcal{R}(u,p,\emptyset))}(q_0) = q$ such that $\delta(q,u) \notin F$ for any $u \in \Sigma^*$. These states will be in R_p (locally rejecting states).

4.1.2 Size of Resulting Asynchronous Automaton

After doing the optimization as suggested in [2], the size of the resulting asynchronous automaton is $2^{O(2^n)}$. [4] gives a more accurate size of the resulting asynchronous automaton of this construction- $n^{n^2}.|\mathcal{D}|^{|\mathcal{D}|.2^n}$ where m is the size of

4.2 Zone Method

[3] tries to make the algorithm faster.

[2] uses factors called residues. [3] uses factors called zones. Just like how in [2] the resulting asynchronous automaton tries to understand the transition function on process residues, the resulting asynchronous automaton of [3] tries to understand transition function on zones. To understand transition functions on process residues, the resulting asynchronous automaton in [2] keeps track of transition functions on primary residues. The asynchronous automaton of [3] keeps track of transition functions of zones. There are less zones than there are primary residues. This decreases the number of states of the asynchronous automaton.

Definition 4.2.1. *For a trace $(\mathcal{E}, \leq, \lambda)$, $last(T) = \{max_p(T) | p \in \mathcal{P}\}$. $last(T)$ will also be referred to as global primary information.*

Now zones will be defined.

Definition 4.2.2. *For events e_1 and e_2 in a trace $T = (\mathcal{E}, \leq, \lambda)$*

1. *$S : \mathcal{E} \to \mathbb{P}(last(T))$ such that $S(e_1) = \{f \in last(T) | e_1 \leq f\}$ where $\mathbb{P}(last(T))$ denotes the power set of $last(T)$. Similarly for e_2*
2. *If $S(e_1) = S(e_2)$ then $e_1 \equiv e_2$.*

3. *Equivalence class of $\equiv$ relation is called a zone.*

The automaton in the paper now stores transition functions for linearizations of zones.

Definition 4.2.3. *If Z_1 and Z_2 be two zones of $(\mathcal{E}_u, \leq_u, \lambda_u)$. Let $e_1 \in Z_1$ and $e_2 \in Z_2$ be two elements. If $e_1 \leq_u e_2, Z_1 \leq_Z Z_2$. $\leq_Z$ is called zone order.*

In the last method, $u[\mathcal{E}_u] \sim u[\partial_{p_1}(\mathcal{E}_u)]u[\partial_{p_2}(\mathcal{E}_u)\backslash\partial_{p_1}(\mathcal{E}_u)...u[\partial_{p_n}(\mathcal{E}_u)\backslash\partial_{\{p_1...p_{n-1}\}}]$ was used. Here, the fact used is $u \sim u[Z_1]u[Z_2]...u[Z_v]$ where Z_i are zones of T_u and $Z_i \not\geq_Z Z_{i+1}$.

Zones have quite an interesting property. The property is written in the next lemma

Lemma 4.2.1. *[3][4]For a trace T*

1. *A zone of T is a factor of T.*
2. *A zone of T contains at most one event of global primary information.*
3. *The set of zones partitions the set of events of T*
4. *The relation $<$ on zones is acyclic. It induces a partial order such that $S(Z') \subsetneq S(Z)$ and $dom(Z) \cap dom(Z') \neq \emptyset$ implies $Z < Z'$*
5. *For every zone $Z, S(Z) \neq \emptyset$*
6. *There are at most $|\mathcal{P}|^2$ many zones. [3] says $|\mathcal{P}|^2 + |\mathcal{P}|$, but [4] gives this better lower bound*

Proof. We prove each point

1. The proof follows from definition.
2. Assume e_1, e_2 are two distinct global primary information events.
   - Case 1 $e_1 < e_2$. This means $e_1 \in S(e_1)$. But, $e_1 \notin S(e_2)$. Hence, e_1 and e_2 can't be in the same zone
   - Case 2 $e_1 > e_2$. This means $e_2 \in S(e_2)$. But $e_2 \notin S(e_1)$. Hence, e_1 and e_2 can't be in the same zone.

– Case 3 e_1 and e_2 are incomparable. $e_1 \in S(e_1)$. But $e_1 \notin S(e_2)$. Hence, e_1 and e_2 can't be in the same zone.

3. First proving every event is in a zone. For every event e, there exists $p \in \mathcal{P}$ such that e is a p-event. Hence, $max_p(T) \geq e$ and $max_p(T) \in S(e)$. Hence, for every event $e, S(e) \neq \emptyset$. Hence, for every event e, there is a zone Z_e such that $e \in Z_e$.

   To prove that no two zones intersect. Let Z_1, Z_2 be two distinct zones. Assume $Z_1 \cap Z_2 \neq \emptyset$. Let $e \in Z_1 \cap Z_2$. By definition of zone, $S(e) = S(Z_1)$ and $S(e) = S(Z_2)$. Since $Z_1 \neq Z_2, S(Z_1) \neq S(Z_2)$. Hence, $S(e) \neq S(e)$. This is a contradiction.

4. Proving that the relation is acyclic. Assume note. Assume $Z_1 < Z_2 < ... < Z_m = Z_1$. By definition, if $Z_i < Z_j$, $e_i < e_j$ for some $e_i \in Z_i, e_j \in Z_j$ ($e_i \notin Z_j, e_j \notin Z_i$ (by partition property)). Since $e_i < e_j, S(e_j) \subseteq S(e_i)$ (any event $e \geq e_j \implies e \geq e_i$). Since $Z_i \neq Z_j, S(e_j) \subsetneq S(e_i)$. This means $Z_1 \subsetneq Z_2 \subsetneq ... \subsetneq Z_m = Z_1$. Which means $Z_1 \subsetneq Z_1$. This is a contradiction.

   Proving that if $S(Z_1) \subsetneq S(Z_2)$ and $dom(Z_1) \cap dom(Z_2) \neq \emptyset$, then $Z_2 \leq Z_1$. Since $dom(Z_1) \cap dom(Z_2) \neq \emptyset$, there exists $e_1 \in Z_1$ and $e_2 \in Z_2$ such that both are p-events for some p. This means either $e_1 < e_2$ or $e_2 < e_1$ (can't be equal as zones are partitions). It is known that $S(Z_1) \subsetneq S(Z_2)$. Hence $S(e_1) \subsetneq S(e_2)$. Hence, it can't be the case that $e_1 < e_2$. Hence, $e_2 < e_1$. By definition of the zone relations, $Z_2 < Z_1$

5. This is proved above. Refer point 3.

6. $|\mathcal{P}|^2 = |\mathcal{P}|.|\mathcal{P}|$. Assume that the number of zones is more than $|\mathcal{P}||\mathcal{P}|$. This means that there is a process p such that there are at least $|\mathcal{P}| + 1$ many zones with domain containing p. Let the set of the zones be $\mathbb{Z}_p = \{Z_1, Z_2, ..., Z_{p+m}\}$ for some natural $m \geq 0$. Since domains of zones in $\mathbb{Z}_p$ intersect, $(\mathbb{Z}, \leq)$ is a totally ordered set. Without loss of generality, $Z_1 < Z_2 < ... < Z_{|\mathcal{P}|+m}$. Hence, $S(Z_1) \subsetneq S(Z_2) \subsetneq ... \subsetneq S(Z_{|\mathcal{P}|+m})$. Since $S(Z_{|\mathcal{P}|+m}) \neq \emptyset, |S(Z_1)| > |\mathcal{P}| + m + 1$. This is a contradiction as there are at most $|\mathcal{P}|$ many events in $last(T)$ and $S(Z_1) \subseteq last(T)$.

□

Another lemma is needed for the update of the zones when new letter is read.

Lemma 4.2.2. *Let $P \subsetneq \mathcal{P}$ and $q \notin P$, T is a trace and e, f be two events in $pref_P(T)$.*

$S(e) = \{e' \in last(pref_P(T)) | e \leq e'\}$. *Similarly for* f

$S'(e) = \{e' \in last(pref_{P \cup \{q\}}) | e \leq e'\}$. *Similarly for* f.

Then if $S(e) = S(f)$*, then* $S'(e) = S'(f)$.

Proof. Assume not. Without loss of generality, there exists $e' \in S'(e)$ such that $e' \notin S'(f)$.

– Case 1- if $e' \in pref_P(T)$, it is a contradiction to the fact that $S(e) = S(f)$

– Case 2- if $e' \notin pref_P(T)$

Let $H = \{h \in pref_P(T) | e \leq h < e'\}$. H must have maximal element due to finiteness and partial order. Let h' be such a maximal event. $h' \in pref_P(T)$ and is $max_{p'}(pref_P(T)) = h'$ for some $p' \in dom(h')$. Hence, $h' \in last(pref_P(T))$ and hence $h' \in S(f)$. This means $f \leq h' < e'$. Hence, $e' \in S'(f)$. This is a contradiction.

□

Asynchronous automata for regular trace language $\mathcal{L}$

Let $\mathcal{D} = \{S, \Sigma, q_0, F, \rightarrow\}$ be a DFA accepting $\mathcal{L}$.

The asynchronous automaton constructed is $\mathcal{A} = \{\{S_p\}_{p \in \mathcal{P}}, \delta, Q_0, \mathcal{F}, (\Sigma, \theta)\}$ such that elements of S_p look like $(TS, S_1, \langle dom(Z_i), S(Z_i), \Delta(Z_i)_{i \in I} \rangle)$ where

1. TS is a state of the gossiping automaton.
2. S_1 is a set of primary information of p (with labels from the gossiping automaton).
3. Z_i are zones in $\partial_p(T_u)$.
4. $dom(Z_i)$ and $S(Z_i)$ are as usual.
5. $\Delta(Z_i) : S \rightarrow S$ such that for all $s \in S, \Delta(Z_i)(s) = \delta(s, L_i)$ where L_i is a linearization of Z_i
6. $I = \{1...m\}$ where m is the number of zones.

The idea is that whenever p participates in the current event, the state of p will transition to a state with correct information. The S_1 and TS part of the state will be updated like the gossiping automaton.

Let u be a string $(S_1^u, TS^u, \langle dom(Z_i^u), S(Z_i^u), \Delta(Z_i^u)\rangle_{i=0...m^u})$ be the state of p after the automaton transitioned on u.

The state reached by the process after automaton reads $u.a$ is $\delta(\delta(Q_0, u), a)$. It is

$(S_1^{u.a}, TS^{u.a}, \langle dom(Z_i^{u.a}), S(Z_i^{u,a}), \Delta(Z_i^{u.a})\rangle_{i=0...m^{u.a}})$

This local state of processes in $dom(a)$ needs some steps to get computed properly. Without loss of generality, $dom(a) = \{p_1...p_l\}$ and the event corresponding to the new a in T_{ua} is e_a

- Step 1- For each process p, we update the zone information for zones of $T_{u.a}$ restricted to $\partial_p(T_u) \cup \{e_a\}$
- Step 2- For p_1, p_2, we update the zone information for zones of $T_{u.a}$ restricted to $\partial_{\{p_1,p_2\}}(T_u) \cup \{e_a\}$
- Step 3- For p_1, p_2, p_3 we update the zone information for zones of $T_{u.a}$ restricted to $\partial_{\{p_1,p_2,p_3\}}(T_u) \cup \{e_a\}$

  .

  .

  .

- Step l- For $p_1...p_l$ we update the zone information for zones of $T_{u.a}$ restricted to $\partial_{\{p_1,...p_l\}}(T_u) \cup \{e_a\} = \partial_{dom(a)}(T_{u.a})$

This is enough because zones not in $\partial_{dom(a)}(T_{u.a})$ will not change.

For Step 1

The new state of p will be $(S_1', TS', \langle dom(Z_i'), S(Z_i').\Delta(Z_i')\rangle_{i=0...m+1})$ constructed as follows

1. S'_1 is the primary information of p in $\partial_p(T_{u.a})$
2. TS' is the state of the gossiping automaton in $\partial_p(T_{u.a})$
3. $dom(Z'_{m+1}) = dom(a)$
4. $S(Z'_{m+1}) = \{e_a\}$
5. $\Delta(Z'_{m+1}) = \xrightarrow{a}$
6. $\langle dom(Z'_j), S(Z'_j), \Delta(Z'_j)\rangle_{j=1...m} = \langle dom(Z'_j), S(Z_j), \Delta(Z_j)\rangle_{j=1...m}$
7. For all Z'_j, $S(Z'_j) \leftarrow S(Z_j) \cup \{dom(a)\}$
8. If $S(Z'_i) = S(Z'_j)$ with $Z'_j \not< Z'_i$, then we merge Z'_i, Z'_j and let $\Delta(Z'_i) = \Delta(Z'_j) \circ \Delta(Z'_i)$ and $dom(Z'_i) = dom(Z'_i) \cup dom(Z'_j)$ and we delete Z'_j.
9. Repeat 8 till it can't be used.

In the construction, 3,4 and 5 ensure that there is a zone Z such that $S(Z) = \{e_a\}$. It is assumed for the time being that $Z = \{e_a\}$.

6 enforces that for the time being, the other zones are considered unchanged.

Since $e_a > g$ and $g \in \partial_p(T_u) \cup \{e_a\}$, all zones for the process must have e_a in the S image. This update is done by 7.

Some zones stored as distinct zones by p might have same S image and are not less than another. This would means that the two zones are part of same zone in $\partial_p(T_u) \cup \{e_a\}$. So they are merged in 8.

Constructions for Step 2 and onwards

The steps mentioned here convert the zone information about zones restricted to $\partial_P(T_u) \cup \{e_a\}$ to zone information about zones restricted to $\partial_{P\cup\{q\}}(T_u) \cup \{e_a\}$ for new process q. On step $N \in \{2...l\}$, use the conversion such that $P = \{p_1...p_{N-1}\}, q = p_N$.

The state of q (after undergoing 'step 1') is $(S^q_1, TS^q, \langle dom(Z^q_i), S(Z^q_i), \Delta(Z^q_i)\rangle_{i=1...M})$. For some state $p \in P$, the state is $(S_1, TS, \langle dom(Z_i), S(Z_i), \Delta(Z_i)\rangle_{i=1...m})$. (All process in P have the same state by construction.)

The updated states of p is constructed as follows. All states of $P \cup \{q\}$ will have the same state.

1. S'_1 is the primary information in $\partial_{P\cup\{q\}}(T_u)$
2. TS' is the state of the gossiping automaton after reading $\partial_{P\cup\{q\}}(T_u)$
3. Let $J = \{i \in [1...n] | TS^q(S(Z^q_i)) \cap TS(S_1) = \emptyset\}$. $v = m + |J|$
4. $\langle dom(Z'_j), S(Z'_j), \Delta(Z'_j)\rangle_{j\in[1...m]} = \langle dom(Z_j), S(Z_j), \Delta(Z_j)\rangle_{j\in[1...m]}$
5. $\langle dom(Z'_j), S(Z'_j), \Delta(Z'_j)\rangle_{j\in[m+1...m+|J|]} = \langle dom(Z_j), S(Z_j), \Delta(Z_j)\rangle_{j\in J}$
6. The partial order $<'$ on the new zones is given by the transitive closure of the relation $< \cup <^q \cup \{(Z'_i, Z'_j) | i \leq m < j, dom(Z'_i) \cap dom(Z'_j) \neq \emptyset\}$
7. For all $Z'_i <' Z'_j, S(Z'_i) \leftarrow S'(Z'_i) \cup S'(Z'_j)$
8. If $S(Z'_i) = S(Z'_j)$ and $Z_j \not< Z_i$, then we merge Z'_i and Z'_j and set $dom(Z'_i) = dom(Z'_j) \cup dom(Z'_i)$ and $\Delta(Z'_i) = \Delta(Z'_j) \circ \Delta(Z'_i)$
9. Repeat 8 till it can't be repeated.

It must be the case that $\bigcup_{i\in[1...v]} Z'_i = \partial_{P\cup\{q\}}(T_u.a)$. This is done by 3, 4 and 5. Taking Z^q_i for $i \notin J$ is not necessary. This is because of a previous lemma. The lemma implies that a zone is either completely in $\partial_P(T_{u.a})$ or completely out. For $Z^q_i, i \notin J$, the zone was already counted as a zone from the view of P. (It is possible that a zone in view of P can have maximum q-event in its 'S' image)

6 incorporates new zones with the zone order appropriately. 7 updates the value of $S(Z'_i)$. This is needed as some zones in $\partial_P(T_{u.a})$ can have $max_q(T_{u.a})$ in its S image.

As discussed earlier, it can happen that two distinctly stored 'zones' have same S image and one is not less than another. This actually means that they are subsets of a zone in $\partial_{P\cup\{q\}}(T_{u.a})$. So they are merged in 8.

For a process p, let S^0_1 be the primary information at the beginning (like discussed in the gossiping automaton). TS^0 be the timestamping information (gossiping automaton state)

at the beginning. The number of zones is 1. Call the zone Z_0. Z_0 is the zone containing $e_\perp$. $dom(Z_0) = \mathcal{P}, S(Z_0) = \{e_\perp\}, \Delta(Z_0) = Id$.

$Q_0 = (S_1^0, TS^0\langle dom(Z_0), S(Z_0), \Delta(Z_0)\rangle)$

A state $s = (S_1, TS, \langle dom(Z_i), S(Z_i), \Delta(Z_i)\rangle_{i\in[1...n]}) \in \mathcal{F}$ if $\Delta(Z_{i_1})\circ\Delta(Z_{i_2})\circ...\circ\Delta(Z_{i_n})(q_0) \in F$ where $Z_{i_1}, Z_{i_2}, ..., Z_{i_n}$ is a linearization of $(\{Z_i | i \in [1...n]\}, \leq_Z)$

The automaton thus created accepts the regular trace language $\mathcal{L}$.

4.2.1 The Locally Rejecting States

Let $\mathcal{D}$ be the input DFA as mentioned above and $\mathcal{A}$ be the resulting asynchronous automaton.

Lemma 4.2.3. *Let $\{Z_i\}_{i\in I}, ZO$ be the zone order of p for string u. $lin(\{Z_i\}_{i\in I}) \sim pref_p(u)$ where $lin(\{Z_i\}_{i\in I}$ is the linearization of Z_i with ZO.*

For every process p, consider the local p-states where $f_{lin(\{Z_i\}_{i\in I})}(q_0) = q$ such that $\delta(q, u) \notin F$ for all $u \in \Sigma^*$. This is in R_p

4.2.2 Size of Resulting Asynchronous Automaton

[4] gives the upper bound on size of the resulting asynchronous automaton- $2^{3.n^3}.|\mathcal{D}|^{|\mathcal{D}|n^2}$.

4.3 Updated Zone Method

The third algorithm is like the zone algorithm. However, instead of storing the complete transition function, $\Delta(Z_i)$ completely, the processes simply store the image of q_0 for the transition function of$\downarrow \Delta(Z_i)$. This requires much less storage. Hence, the algorithm is quicker. As expected, this algorithm creates asynchronous automaton exponentially larger (in number of processes) than the input DFA.

Like usual, let $\mathcal{D} = \{S, F, q_0, \delta, \Sigma\}$ be the DFA accepting the regular trace language $\mathcal{L}$.

The resulting output asynchronous automaton is $\mathcal{A} = \{\{S_p\}_{p\in\mathcal{P}}\}, (\Sigma, \theta), \mathcal{F}, Q_0, \Delta\}$.

The set of states of a process look like $(TS, ZO, \hat{\Delta}(Z_i)_{i\in[1...m]})$ where

1. TS is the states of process p in the gossiping automaton.
2. $ZO = (\{Z_i | i \in [1...m]\}, \leq_Z)$ where Z_is are the zones in p-view and $\leq_Z$ is the relation of the zones
3. $\hat{\Delta}(Z_i) = \delta(q_0, \downarrow \bar{Z}_i)$ where $\downarrow \bar{Z}_i$ is a linearization of $\downarrow Z_i$.

The following lemmas is needed to update the state information.

Lemma 4.3.1. *For any subtraces $\mathcal{T}_1, \mathcal{T}_2$ of T, there exists subtraces T_0, T_1 and T_2 of T such that $dom(T_1) \cap dom(T_2) = \emptyset$ and $\mathcal{T}_1 = T_0T_1$ and $\mathcal{T}_2 = T_0T_2$*

Proof. The proof can be found in [5]. □

Lemma 4.3.2. *Let T_0, T_1, T_2 be traces such that $dom(T_1) \cap dom(T_2) = \emptyset$. Then, for an I-diamond automaton, $\delta(q_0, T_0T_1T_2)$ can be calculated from $\delta(q_0, T_0), \delta(q_0, T_0T_1)$ and $\delta(q_0, T_0T_2)$ where T_iT_j is the trace of $u_i.u_j$ where u_i and u_j are linearizations of T_i and T_j.*

Proof. The statement is claimed in [4]. □

It is necessary to update the information in the state.

1. TS- Is updated like in the Gossiping automaton.
2. ZO- Needs some work
3. $\hat{\Delta}$- Needs some work

The following lemma helps in calculating $S_u(Z)$ given ZO_u.

Lemma 4.3.3. *For trace T_u and a zone Z, $max_r(T) \in S_u(Z)$ if and only if $Z \leq_{Z_u} Z_r$ where Z_r is the maximum zone with r in the domain.*

Proof. ($\Rightarrow$)

If $max_r(T) \in S_u(Z)$ then for some $e \in Z, e \leq_u max_r(T_u)$. This means that $Z \leq_{Z_u} Z^r$ where Z^r is the zone containing $max_r(T_u)$. $Z^r = Z_r$ otherwise $max_r(T_u) \leq_u e_r$ for some r-event. This is a contradiction.

($\Leftarrow$)

If $Z \leq_{Z_u} Z_r$, there exists $e \in Z$ and $e' \in Z_r$ such that $e \leq e'$. Since $e', max_r(T) \in Z_r, max_r(T) \in S_u(e')$. Hence, $e' \leq_u max_r(T)$. Hence, $e \leq_u max_r(T)$. Hence, $max_r(T) \in S_u(Z)$

□

Lemma 4.3.4. *If zone order ZO_u is given, $dom_u(Z)$ and $S_u(Z)$ can be computed.*

Proof. $dom_u(Z)$ is already captured in ZO_u. Hence trivial.

By the previous lemma, $S_u(Z)$ can be computed. □

Hence, the S function and domain information can be updated just like the zone algorithm automaton.

If the zone update in the zone algorithm automaton is observed carefully, the new zones are unions of zones and $\{e_a\}$. Let the current trace be of $u \in \Sigma^*$. We append new input a to it. e_a is the event associated with it. For new zone Z_{new}, let $\{C_1, C_2, ..., C_m\} \subseteq ZO_u \cup \{e_a\}$ such that $\bigcup_{i \in [1...m]} C_i = Z_{new}$.

Lemma 4.3.5. *For $i, j \in [1...m]$, there exists k such that $C_i \leq_{ua} C_k$ and $C_j \leq_{ua} C_k$.*

Proof. If $C_i \leq_{ua} C_j$ or $C_i \leq_{ua} C_j$ then trivial.

If $C_i = \{e_a\}$, then the zone contains $max_{\bar{p}}(T_{ua})$ for some participant $\bar{p}$. Hence $C_j \leq_{ua} C_i$ is necessary. If not, then Z_{new} will have more than one event of global primary information, which is a contradiction. Similarly if $C_j = \{e_a\}$.

Assume C_i and C_j are incomparable.

Hence, there exists $q \in \mathcal{P}$ such that $q \in dom(C_i)$ but $q \notin dom(C_j)$.

If $q \in dom(a)$, then $max_q(T_{ua}) \in S_{ua}(Z_{new})$. Hence, $max_q(T_{ua}) \in S_{ua}(C_j)$. Hence, $C_j \leq_{ua} \{e_a\}$. Hence, $C_k = \{e_a\}$.

If $q \notin dom(a)$ assume C_k doesn't exist. $max_q(T_u) \in S_u(C_i)$. $max_q(T_u) = max_q(T_{ua})$. Since $C_i, C_j \subseteq Z_{new}, max_q(T_{ua}) \in S_{ua}(C_j)$. Since $q \notin dom(a), max_q(T_{ua}) = max_q(T_u)$ and hence $max_q(T_u) \in S_{ua}(C_j)$. Since $q \notin dom(a)$, $max_q(T_u) \in S_{ua}(C_j)$ only if $max_q(T_u) \in S_u(C_j)$. Let e_q be the maximal q event in Z_{new}. Let C_k be the set containing e_q.

□

Corollary 4.3.6. *For a zone Z_{new} in T_{ua}, there exists set C_{max} which is a zone in T_u or $C_{max} = \{e_a\}$ such that $\downarrow_{ua} Z_{new} = \downarrow_{ua} C_{max}$.*

This corollary gives the following lemma which can be used to update $\hat{\Delta}$.

Lemma 4.3.7. *An updated zone Z_{new} in ZO_{ua} is such that $\downarrow Z_{new} = \downarrow C_{max}$ where $C_{max} \in ZO_u$ or C_{max} is a zone containing e_a. And C_{max} can be computed effectively*

Proof. The existence of C_{max} is a corollary of the previous lemma.

C_{max} can be computed while updating zone order. Remember the greatest zone or $\{e_a\}$ merged. □

Definition 4.3.1. *Let ZO have linerarization of zones $\{Z_1, Z_2, ..., Z_m\}$.*

For $k \geq i, B_{i,k} = \{Z_1 ... Z_i\} \cup \{Z_j | j \geq i, Z_j \leq Z_k\}$

Lemma 4.3.8. *For a trace T_u, if ZO_u and $\hat{\Delta}(Z \in ZO_u)$ is known, then $\delta(q_0, B_{i,k})$ can be calculated ($B_{i,k}$ is a linearization of $B_{i,k}$)*

This implies that $\delta(q_0, T_u)$ can be calculated as $T_u = B_{m,m}$

Proof. The proof is based on induction on i.

$\delta(q_0, B_{0,k}) = \hat{\Delta}(Z_k)$

Hence true for $i = 0$.

Assume true for i. Prove true for $i + 1$

By induction hypothesis, $\delta(q_0, B_{i,k})$ is known for all $k \geq i$. Consider $B_{i+1,k}$.

– Case 1- $Z_{i+1} \leq Z_k$. $B_{i+1,k} = B_{i,k}$. Hence by induction hypothesis, we know $\delta(q_0, B_{i+1,k})$

– Case 2- $Z_{i+1} \not\leq Z_k$. Consider $T_0 = B_{i,i}, T_0T_1 = B_{i+1,i+1}, T_0T_2 = B_{i,k}$. $B_{i+1,k} = T_0T_1T_2$. Calculate $\delta(q_0, T_0T_1T_2)$ ($dom(T_2) \cap dom(T_1) = \emptyset$ otherwise, $Z_{i+1} \leq Z_k$

□

Hence, $\delta(q_0, u)$ can be computed for every $u \in \Sigma$.

The catch is that this is possible only if $\hat{\Delta}$ can be updated properly. The following lemma proves that proper update is possible.

Lemma 4.3.9. *$\hat{\Delta}$ can be updated when new letter is appended to input word*

Proof. Let initial word be u and new letter a. Z_{new} be a new zone. Let $\mathbb{Z}$ be the maximal set of old zones such that $Z_{new} \backslash \{e_a\} = \cup\mathbb{Z}$. The old zone order is $\mathbb{Z}\mathbb{O}$. Based on previous lemma, we can calculate $\delta(q_0, \bar{\mathbb{Z}})$ where $\bar{\mathbb{Z}}$ is a linearization of $\mathbb{Z}$. $\hat{\Delta}_{new}(Z_{new}) = \delta(\delta(q_0, \bar{\mathbb{Z}}), a)$.

Done □

The Asynchronous Automaton

$\mathcal{A} = (\{S_p\}_{p \in \mathcal{P}}, \Delta, Q_0, \mathcal{F}, (\Sigma, \theta))$ where $\mathcal{P} = \{p_1 ... p_n\}$

1. $S_p = \{TS, ZO, \hat{\Delta}\}$ such that

   (a) TS is a local state of p in the gossiping automaton of the distribution

(b) ZO is all possible zone orders.

(c) $\hat{\Delta}$ is the function mapping every zone in ZO to a state of $\mathcal{D}$.

2. $Q_0 = \{TS_0, ZO_0, \hat{\Delta}_0\}$ where $TS_0, ZO_0, \hat{\Delta}_0$ are the initial timestamp, zone order and $\hat{\Delta}$

3. Δ updates all the three parameters as discussed.

Deciding the states in $\mathcal{F}$ needs the knowledge of $\hat{\Delta}$ stored by all processes. Let G be a global state Let $s \in S$ be the state of the DFA corresponding to $B_{m,m}$ where m is the number of zones in G. If $s \in F$, $G \in \mathcal{F}$.

$\mathcal{F} = \{\text{global state} | \delta(q_0, B_{m,m}) \in F, \text{where } m \text{ is the number of zones in global state}\}$

4.3.1 The Locally Rejecting States

Like before, let $\mathcal{D}$ be the input DFA and $\mathcal{A}$ be the resulting asynchronous automaton.

Lemma 4.3.10. *For every process p, for every p-local state, there exists Z_p such that it is unique maximal element of $\{Z_i\}_{i \in I}$ with given zone order. $p \in dom(Z_p)$*

For every process p, consider the set of local states with $\hat{\Delta}(Z_p) = q$ such that $\delta(q, u) \notin F$ for any $u \in \Sigma^*$. This is the locally rejecting state set.

As observed, all these algorithms output asynchronous automata exponentially larger than the corresponding Diamond DFA. But that is expected as the resulting asynchronous automata are locally rejecting. It is proved that the exponential blow-up must happen for $Path_n$ by these algorithms.

4.3.2 Size of Resulting Asynchronous Automaton

[4] gives the upper bound on size of resulting asynchronous automaton- $4^{n^4}|\mathcal{D}|^{n^2}$

Chapter 5

Lower and Upper Bounds of Automata accepting $Path_n$

Some of the attempts to prove the conjecture try to find an exponential lower bound for size of asynchronous automaton accepting $Path_n$. Since $Path_n$ has minimal DFA of size less than equal $n^2 + 2$, the lower bound would imply the conjecture.

5.1 ϵ-Non-Deterministic Asynchronous Automata

A asynchronous automata which is non-deterministic and has ϵ moves is a non-deterministic asynchronous automata

5.1.1 Upper Bound for Minimal ϵ-Non-Deterministic Asynchronous Automaton for $Path_n$

This section will prove that for every $n \in \mathbb{N}$, there exists a non-deterministic asynchronous automaton of linear size which can accept $Path_n$.

For all $n \in \mathbb{N}$ the non-deterministic asynchronous automaton accepting $Path_n$ is called

$\mathcal{N}_n = \{\{S_p\}_{p\in\mathcal{P}}, \Delta, Q_0, F, (\Sigma, \theta)\}$

1. $S_p = \{on, off\}$
2. (Σ, θ) is as usual for $Path_n$
3. $Q_0 = (off, off, ...off)$
4. F are the global states with only one process in the on state
5. Defining Δ. For an input letter (p_a, p_b),
   (a) $\Delta_{(p_a,p_b)}((off, off), (p_a, p_b)) = \{(on, on)\}$
   (b) $\Delta_{(p_a,p_b)}((on, on), (p_a, p_b)) = \{(on, on)\}$
   (c) $\Delta_{(p_a,p_b)}((on, off), (p_a, p_b)) = \{(on, off), (off, on)\}$
   (d) $\Delta_{(p_a,p_b)}((off, on), (p_a, p_b)) = \{(on, off), (off, on)\}$
   (e) (ϵ transition) At any time, a process may transition from off state to on state.

It is clear that the size of the non-deterministic asynchronous automaton is $2n$

Some lemmas are required to prove $\mathcal{L}(\mathcal{N}_n) = Path_n$

Lemma 5.1.1. *If a global state g of $\mathcal{N}_n$ has v many processes in on local state, then for every $u \in \Sigma^*$, a global state in $\Delta(g, u)$ has at least v many states in on local state.*

Proof. Proof by induction on length of u.

True if $len(u) = 1$.

Assume true if $len(u) = k$. Prove true for $len(u) = k + 1$.

Let $u = v.a$ for some $v \in \Sigma^*, a \in \Sigma$. By the induction hypothesis, $\Delta(g, v)$ is a subset of the set of global states with at least v many processes in on local states. Invoking the fact that the lemma is true for $len(u) = 1$, it can be shown that $\Delta(g'' \in \Delta(g, v), a)$ has at least v many processes in on local states. This in turn means that all states in $\Delta(\Delta(g, v), a) = \Delta(g, u)$ have at least v many processes in on local state. □

Corollary 5.1.2. *If a global state g of $\mathcal{N}_n$ has 2 or more processes in on local state, then g is such that for all $u \in \Sigma^*$, $\Delta(g, u) \not\subseteq F$.*

Theorem 5.1.3. $\mathcal{L}(\mathcal{N}_n) = Path_n$

Proof. ($Path_n \subseteq \mathcal{L}(\mathcal{N}_n)$)

This is easier than the other part. Let $u = (p_{a_1}, p_{b_1})...(p_{a_m}, p_{b_m}) \in Path_n$. $u \in \mathcal{L}(\mathcal{N}_n)$ by following the following steps.

1. At the beginning, use ϵ-move to turn p_{a_1} on
2. At any $z \in [1...m]$,
   - Case 1- if state of $p_{a_z} = on$ and state of $p_{b_z} = off$, then transition
     i. $(on, off) \to (on, off)$ if $a_{z+1} \in \{a_z, b_z\}$
     ii. $(on, off) \to (off, on)$ if $a_{z+1} \notin \{a_z, b_z\}$ and $b_{z+1} \in \{a_z, b_z\}$
   - Case 2- if state of $p_{a_z} = off$ and state of $p_{b_z} = on$, then transition
     i. $(off, on) \to (on, off)$ if $a_{z+1} \in \{a_z, b_z\}$
     ii. $(off, on) \to (off, on)$ if $a_{z+1} \notin \{a_z, b_z\}$ and $b_{z+1} \in \{a_z, b_z\}$
   - Case 3- Another case won't happen due to design

It can be proved that following these rules will lead to an accepting state. The idea is to do an ϵ move to turn on the process participating in first letter of string. Then during every transition, the participating process which is involved in the letter must be turned on. If both are involved in next letter, then a tie-breaker is used to decide which one to turn on and which one to turn off.

($\mathcal{L}(\mathcal{N}_n) \subseteq Path_n$)

Let $u = u_1...u_k \in \Sigma^*$, but not in $Path_n$.

Let $D \subseteq [2...k]$ be the set of indices such that $dom(u_{i-1}) \cap dom(u_i) = \emptyset$. Let h be the infimum of the set.

There are 3 cases about $g \in \Delta(Q_0, u_1...u_{h-1})$

- Case 1- If $g \in \Delta(Q_0, u_1...u_{h-1})$ has 2 or more processes in on local state. By 5.1.1, $\Delta(g, u_h...u_k)$ will have more than 1 processes in on local state. Hence, $\Delta(g, u_h...u_k) \notin F$
- Case 2- If $g \in \Delta(Q_0, u_1...u_{h-1})$ has only 1 process in on local state.
  - SubCase 1- The process in on local state in global state g is in $dom(u_{h-1})$. This means that both processes in $dom(u_h)$ are in off state in g. This means that for all $g' \in \Delta(g, u_h)$ at least 2 processes are in on local state. (if ϵ move is used before u_h move, at least 2 processes will be in on local state and if ϵ move is not used before u_h move, at least 3 processes will be in on local state). By previous lemma, this means that $\Delta(\Delta(Q_0, u_1...u_h), u_{h+1}...u_k) = \Delta(Q_0, u) \notin F$
  - Subcase 2- The process in on local state in global state g is in $dom(u_h)$. This means that both processes in $dom(u_{h-1})$ are in off local state. This will cause atleast 2 on processes to be in on state.
- Case 3- If $g \in \Delta(Q_0, u_1...u_{h-1})$ has 0 processes in on local state. This is not possible as u_1 transition has already happened. At least 1 process must be on.

Hence, $\mathcal{L}(\mathcal{N}_n) \subseteq Path_n$. Hence $\mathcal{L}(\mathcal{N}_n) = Path_n$

□

5.1.2 Lower Bound for ϵ-Non-Deterministic Asynchronous Automaton

Theorem 5.1.4. *A lower bound for ϵ-Non-Deterministic Asynchronous Automaton accepting $Path_n$ is $2n-1$ (for $n \geq 4$)*

Proof. Assume there exists a ϵ-Non-Deterministic Asynchronous Automaton $N = \{\{S_p\}_{p \in \mathcal{P}}, (\Sigma, \theta), Q_0, F, \Delta\}$ accepting $Path_n$ with size less than $2n-1$. This means that at least 2 processes, say p_i and p_j, have at most 1 states.

Let $u = u_1...u_k \in Path_n$ such that $dom(u_k) \cap \{p_i, p_j\} = \emptyset$. Such a u exists as $n \geq 4$.

$\Delta(Q_0, u) = \Delta(Q_0, u(p_i, p_j))$ because p_i and p_j cannot change state. Hence, $\Delta(Q_0, u(p_i, p_j))$ is accepted by N. This is a contradiction to that fact that $\mathcal{L}(N) = Path_n$ as $u(p_i, p_j) \notin Path_n$. Hence lower bound. □

5.2 Register Asynchronous Automata

A register asynchronous automaton can be thought of as an asynchronous automaton with a global register. Whenever a set of processes participate in an event, they can change what is written in the register.

The following text will show that there exists a register asynchronous automaton of size polynomial in number of processes for $Path_n$. If one can show that there exists an algorithm converting register asynchronous automaton of size N to asynchronous automaton of size $P(N)$ for some polynomial P, then there exists an asynchronous automaton accepting $Path_n$ without exponential blowup by number of processes.

Definition 5.2.1. *A register asynchronous automaton is a tuple* $\{\{S_p\}_{p\in\mathcal{P}}, \Delta : (\prod_{p\in\mathcal{P}} S_p \times \Sigma \times \Sigma') \to (\prod_{p\in\mathcal{P}} S_p \times \Sigma'), Q_0, F_0, (\Sigma, \theta)\}$ *where*

1. S_p *is the set of local states of process* p
2. (Σ, θ) *is input distribution*
3. Σ' *is the register alphabet.* $B \in \Sigma'$.
4. $Q_0 \in \prod_{p\in\mathcal{P}} S_p$ *is the initial global state*
5. $F \subseteq \prod_{p\in\mathcal{P}} S_p$ *are the final global states*
6. *Initially, register is filled with empty symbol* B.

Δ *is defined like the asynchronous case.* $\Delta_a \subseteq (\prod_{p\in P} S_p \times \Sigma \times \Sigma') \times (\prod_{p\in P} S_p \times \Sigma')$ *where* $P = loc(a)$.

$((q_1...q_n, f \in \Sigma'), a, (q'_1...q'_n, f' \in \Sigma')) \in \Delta$ *if*

1. *for* $loc(a) = \{i_1...i_k\}, ((q_{i_1}...q_{i_k}, f), (q'_{i_1}...q'_{i_k}, f')) \in \Delta_a$
2. *for* $i \notin loc(a), q_i = q'_i$

Unless mentioned otherwise, we consider deterministic register asynchronous automaton. Size of register asynchronous automaton is sum of local processes.

5.2.1 Upper Bound for Minimal Register Asynchronous Automaton for $Path_n$

1. $S_p = \{on, off\}$ for all processes p
2. Σ is the same as for $Path_n$
3. $\Sigma' = \Sigma \cup \{B\}$
4. $Q_0 = (on, on...on)$
5. $F = (on, on...on)$

From this information, it is clear that the asynchronous automaton is has size $2n$ where n is the number of processes.

Now $\Delta_{(p_i,p_j)}$ will be defined.

1. $\Delta_{(p_i,p_j)}(on, on, f = (p_a, p_b))$ is equal to
   (a) $(on, on, f = (p_i, p_j))$ if $\{a, b\} \cap \{i, j\} \neq \emptyset$
   (b) $(off, off, f = (p_i, p_j))$ if $\{a, b\} \cap \{i, j\} = \emptyset$
2. $\Delta_{(p_i,p_j)}(switch_1, switch_2, f = B) = (on, on, f = (p_i, p_j))$ for any values of $switch_1$ and $switch_2$- on or off.
3. $\Delta_{(p_i,p_j)}(switch_1, switch_2, f = s \in \Sigma') = (off, off, f = (p_i, p_j))$ (where $switch_1$ or $switch_2$ is off.

This register asynchronous automaton accepts $Path_n$.

5.2.2 Lower Bound for Register Asynchronous Automaton

Theorem 5.2.1. *$2n-1$ is a lower bound for size of register asynchronous automata accepting $Path_n$. (for $n \geq 4$)*

Proof. Assume not. There exists a register asynchronous automaton $\mathcal{A}$ accepting $Path_n$ with size $2n-2$ or less. Let q_0 be the initial state of $\mathcal{A}$. This means that at least 2 processes, say p_i and p_j have 1 local state. Consider a word $u = u_1...u_k \in Path_n$ such that $p_i, p_j \notin loc(u_k)$. Such a u exists for $n \geq 4$.

Consider the word $u' = u.(p_i, p_j)$. $u' \notin Path_n$. But, since p_i and p_j have only 1 local state each, $\Delta(q_0, u) = \Delta(q_0, u')$. Hence, $u \in Path_n$ if and only if $u' \in Path_n$. This is a contradiction.

Hence, the lower bound for register asynchronous automata accepting $Path_n$ is $2n-1$ (for $n \geq 4$).

□

Chapter 6

Asynchronous Automaton Accepting to Locally Rejecting One (for $Path_n$)

In this chapter we prove that there exists an algorithm converting any asynchronous automaton accepting $Path_n$ to a locally rejecting asynchronous automaton accepting $Path_n$.

If there exists a polynomial space algorithm converting any asynchronous automaton accepting $Path_n$ to a locally rejecting asynchronous automaton accepting $Path_n$, the conjecture will be proved to be false.

A rough proof for the above statement is given. Assume there exists a polynomial space algorithm converting any asynchronous automaton accepting $Path_n$ to a locally rejecting asynchronous automaton. Let P be the polynomial of the algorithm such that if input asynchronous automaton is of size x, then size of resulting locally rejecting asynchronous automaton is less than $P(x)$. We know that $P(x)$ has exponential blow-up. Hence, x must also have exponential blow-up.

6.1 Exploiting Properties of Primary Graphs

This idea exploits the fact that for every process $p \in \mathcal{P}$, if $pref_p(u) \in pref(Path_n)$, then during the transition of gossiping automaton on $pref_p(u)$, the primary graph of p is always

non-branched- i.e. the graph is a total order. The moment the primary graph is branched, issues arise.

Definition 6.1.1. *A ordered set $(S, \leq)$ is said to be a linear ordered set if*

1. $\forall s, s' \in S, s \leq s'$
2. $s \leq s', s' \leq s''$ *implies* $s \leq s''$
3. $s \not< s$

Lemma 6.1.1. *$\forall u \in Path_n$, the trace of u, $(\mathcal{E}_u, \leq_u, \lambda_u)$ is a linear ordered set.*

Proof. Proof by induction on length of u.

For length=1, true

Induction Hypothesis- True for length=m

Prove true for length=$m+1$

Let $u.a \in Path_n$ such that $len(u) = m$. $len(u.a) = m+1$.

Consider traces $T_u = (\mathcal{E}_u, \leq_u, \lambda_u)$ and $T_{ua} = (\mathcal{E}_{ua}, \leq_{ua}, \lambda_{ua})$ of u and ua respectively. Event corresponding to last a in T_{ua} is e_a.

Let e be the maximum element in $(\mathcal{E}_u, \leq_u)$ (maximum element exists as $\mathcal{E}_u$ is finite and T_u is linear ordered set by induction hypothesis). $\lambda(e) = (p_x, p_y)$. By definition of $Path_n$, $a = (p_a, p_b)$ such that $\{x, y\} \cap \{a, b\} \neq \emptyset$.

Hence, since T_{ua} is a trace, $e \leq_{ua} e_a$. Hence, linear ordered set.

□

Consider the Gossiping automaton for $Path_n$'s distribution from [2].

Let $\mathbb{TS}_p$ be the set of states of process p in the Gossiping automaton.

$\mathcal{A}_n = ((s_p)_{p \in \mathcal{P}}, \delta, q_0 = (s_1^0, s_2^0, ..., s_n^0), f)$ is an asynchronous automaton accepting $Path_n$

Construct $\mathcal{B}_n = ((S_p)_{p\in\mathcal{P}}, \Delta, Q_0, F)$ such that

1. $S_p = s_p \times \mathbb{TS}_p \times \{good, bad\}$. The idea is to ensure that if $pref_p(u)$ is not in $pref(Path_n)$, then p goes to a bad-state.
2. $Q_0 = ((s_1^0, \Phi_{p_1}, good), s_2^0, \Phi_{p_2}, good), ..., s_n^0, \Phi_{p_n}, good)$ where Φ_p is the initial state of process p in the Gossiping Automaton.
3. $F = \{((s_{p_1}, TS_{p_1}, good), (s_{p_2}, TS_{p_2}, good), ..., (s_{p_n}, TS_{p_n}, good))|(s_{p_1}, s_{p_2}, ..., s_{p_n}) \in f\}$
4. $R_p = s_p \times \mathbb{TS}_p \times \{bad\}$
5. $\Delta(Q = ((s_1, TS_1, l_1), (s_2, TS_2, l_2), ..., (s_n, TS_n, l_n)), a \in \Sigma) =$
   $((t_1, TS_1', o_1), (t_2, TS_2', o_2), ..., (t_n, TS_n', o_n))$ where
   (a) $(t_1, t_2, ..., t_n) = \delta((s_1, s_2, ..., s_n), a)$
   (b) TS_i' is the updated gossiping information from TS_i
   (c) o_i is
      i. bad if
         A. l_k is bad for some $k \in loc(a)$ OR
         B. primary graph of TS_k is not a linear ordered set for any $k \in loc(a)$
      ii. $good$ if
         A. l_k is $good$ for all $k \in loc(a)$ AND
         B. primary graph of TS_k are linear ordered sets for all $k \in loc(a)$

Definition 6.1.2. $OS(((s_1, TS_1, l_1), (s_2, TS_2, l_2), ..., (s_n, TS_n, l_n)) \in \prod_{p\in\mathcal{P}} S_p) = (s_1, s_2, ..., s_n)$

Lemma 6.1.2. $\forall u \in \Sigma^*, OS(\Delta(Q_0, u) = \delta(q_0, u)$

Proof. The proof follows from induction on length of u. □

To talk about 'locally rejecting' asynchronous automaton, 'locally rejecting' states of processes must be defined. The following definitions define a set of states called BAD. It will be proved that for each process, the set of states BAD will be the set of locally rejecting states.

Definition 6.1.3. .

1. *For a global state* $((s_1, TS_1, l_1), (s_2, TS_2, l_2), ..., (s_n, TS_n, l_n))$, l_i *is the active label of process* p_i.
2. BAD *is the set of global states where at least one process has active label as* bad

Lemma 6.1.3. .

1. *If active label of process* p *is* bad*, then the active label of* p *will always be* bad *in future.*
2. *For all* $u \in Path_n$ *such that* $u = v.w$*, then* $v, w \in Path_n$
3. *If* $u' \in pref(Path_n)$ *then* $u' \in Path_n$
4. *For all* $u \in Path_n, pref_p(u) \in Path_n$
5. *For every* $u \in Path_n$*, for every* $p \in \mathcal{P}$ *active primary graph in* $\Delta(Q_0, u)$ *is a linear ordered set*
6. $pref(Path_n) = Path_n$

Proof. We prove each point.

1. The proof follows from definition of transition function.
2. The proof follows from the definition of $Path_n$.
3. The proof follows from the definition of $Path_n$.
4. Since T_u is a linear ordered set, $pref_p(u)$ is a prefix of u. Hence, $pref_p(u) \in Path_n$.
5. Primary graph p is a restriction of of T_u. By previous lemma, T_u is a linear ordered set. hence, primary graph of is a linear ordered set.
6. By previous point in this theorem, $pref(Path_n) \subseteq Path_n$. $Path_n \subseteq pref(Path_n)$ as each string $u \in Path_n$ is its own prefix

□

Lemma 6.1.4. $\forall u \in Path_n, \Delta(Q_0, u) \notin BAD$

Proof. Assume $\Delta(Q_0, u) \in BAD$. There exists $u' = u'_1, u'_2, ..., u'_c$ prefix of u such that

1. $\Delta(Q_0, u') \in BAD$ AND
2. $\Delta(Q_0, u'') \notin BAD$ for all prefixes u'' of u'.

By definition of Δ, for some process $p_i \in loc(u'_c)$, g_i was not a linear ordered set. But, by previous lemma, $u' \in Path_n$. But, since u' is a prefix of u, $u' \in Path_n$ by the previous lemma. Hence by previous lemma, for all processes p_h, g_h is a linear ordered set in $\Delta(Q_0, u')$. This is a contradiction.

Hence, our assumption is wrong. □

Theorem 6.1.5. $\mathcal{L}(\mathcal{B}_n) = Path_n$

Proof. ($\mathcal{L}(\mathcal{B}_n) \subseteq Path_n$)

Say $u \notin Path_n$. This means that $OS(\Delta(Q_0, u)) \notin f$. Hence, by definition of $F, \Delta(Q_0, u) \notin F$. Hence $\mathcal{L}(\mathcal{B}_n) \subseteq Path_n$

($Path_n \subseteq \mathcal{L}(\mathcal{B}_n)$)

Say $u \in Path_n$. This means that $OS(\Delta(Q_0, u)) \in f$. By previous lemma, active labels of $\Delta(Q_0, u)$ are $good$. Hence, by definition of $F, \Delta(Q_0, u) \in F$. Hence, $Path_n \subseteq \mathcal{L}(\mathcal{B}_n)$.

This means $Path_n = \mathcal{L}(\mathcal{B}_n)$ □

Lemma 6.1.6. *Let $T = (\mathcal{E}, \leq, \lambda)$ be a trace. Let $e, e' \in \mathcal{E}$ such that $e \lessdot e'$. If active label of processes in $dom(e)$ are bad, then active label of processes in $dom(e')$ is bad.*

Proof. By the fact that $e \lessdot e'$, there exists $p \in dom(e), dom(e')$. Since active label of p is bad at event e, all processes in $dom(e')$ must mark their active label as bad at event e'. □

Corollary 6.1.7. *Let $T = (\mathcal{E}, \leq, \lambda)$ be a trace. Let $e, e' \in \mathcal{E}$ such that $e \leq e'$. If active label of processes in $dom(e)$ are bad, then active label of processes in $dom(e')$ is bad.*

Theorem 6.1.8. *$u \in \Sigma^*$ is such that active label of p in $\Delta(Q_0, u)$ is bad if and only if $pref_p(u) \notin pref(Path_n) = Path_n$*

Proof. $(\Rightarrow)$

Assume active label of p in $\Delta(Q_0, u)$ is bad but $pref_p(u) \in pref(Path_n) = Path_n$. But by previous lemma, if $pref_p(u) \in pref(Path_n)$, then $pref_p(u) \in Path_n$. By another lemma, $\Delta(q_0, pref_p(u)) \notin BAD$. This is a contradiction

$(\Leftarrow)$

Case 1-$len(pref_p(u)) = 0$. Trivial done as p starts out with $good$ label

Case 2-$len(pref_p(u)) > 0$.

Let $pref_p(u) \notin pref(Path_n) = Path_n$. Hence, $pref_p(T_u) = w.(p_x, p_y)(p_c, p_d).v$ such that $\{x, y\} \cap \{c, d\} = \emptyset$ and $w \in Path_n$ (This is the first instance of mistake in $pref_p(T_u)$. In T_u, the event corresponding to this particular (p_x, p_y) is e and to this particular (p_c, p_d) is e'.

Let h be the smallest event in $\partial_p(T_u)$ s.t $e, e' \leq h$. Let $dom(h) = \{q, q'\}$.

Proposition 6.1.9. *At least one of the following statements hold*

1. $latest_{q \to p_x}$ *and* $latest_{q \to p_c}$ *are incomparable in* T_u *in* $\downarrow h(T_u)$
2. $latest_{q \to p_x}$ *and* $latest_{q \to p_d}$ *are incomparable in* $\downarrow h(T_u)$
3. $latest_{q \to p_y}$ *and* $latest_{q \to p_c}$ *are incomparable in* $\downarrow h(T_u)$
4. $latest_{q \to p_y}$ *and* $latest_{q \to p_d}$ *are incomparable in* $\downarrow h(T_u)$

(proved later) ($\downarrow h(T_u)$ is the ideal generated by h in T_u)

This proposition implies primary graph of processes after transition of h is not a linear ordered set. This implies $\forall \pi \in dom(h)$, active label of π is going to be bad from now on.

By definition of $\partial_p(T_u)$, $max_p(T_u) \geq h$. From previous lemma, active label of p in $max_p(T_u) = bad$. Hence, states with active label 'bad' is locally rejecting state of every process in the locally rejecting asynchronous automaton.

□

Proof of the proposition

Proof. (of the proposition)

$\downarrow h(T_u)$ the ideal generated by h.

Assume that none of them hold

1. $latest_{q \to p_x}(\downarrow h(T_u))$ and $latest_{q \to p_c}(\downarrow h(T_u))$ are comparable.

   $e \leq latest_{q \to p_x}(\downarrow h(T_u)) \leq h$ and $e' \leq latest_{q \to p_c}(\downarrow h(T_u)) \leq h$.

   - Case 1: $latest_{q \to p_x}(\downarrow h(T_u)) > latest_{q \to p_c}(\downarrow h(T_u))$

     $e' \leq latest_{q \to p_c}(\downarrow h(T_u)) \leq latest_{q \to p_x}(\downarrow h(T_u)) \leq h$

     This means $e, e' \leq latest_{q \to p_x}(\downarrow h(T_u)) \leq h$. Due to minimality of h, $latest_{q \to p_x}(\downarrow h(T_u)) = h$. Hence $p_x \in dom(h)$

   - Case 2: $latest_{q \to p_c}(\downarrow h(T_u)) > latest_{q \to p_x}(\downarrow h(T_u))$

     $e \leq latest_{q \to p_x}(\downarrow h(T_u)) \leq latest_{q \to p_c}(\downarrow h(T_u)) \leq h$

     This means $e, e' \leq latest_{q \to p_c}(\downarrow h(T_u)) \leq h$. Due to minimality of h, $latest_{q \to p_c}(\downarrow h(T_u)) = h$. Hence $p_c \in dom(h)$

   - Case 3: $latest_{q \to p_x}(\downarrow h(T_u)) = latest_{q \to p_c}(\downarrow h(T_u))$ This means $e, e' \leq latest_{q \to p_x}(\downarrow h(T_u)) = latest_{q \to p_c}(\downarrow h(T_u)) \leq h$. Due to minimality of h, $latest_{q \to p_x}(\downarrow h(T_u)) = latest_{q \to p_c}(\downarrow h(T_u)) = h$. Hence, $p_x, p_c \in dom(h)$

2. $latest_{q \to p_x}(\downarrow h(T_u))$ and $latest_{q \to p_d}(\downarrow h(T_u))$ are comparable.

   Just like the previous point, this implies that either p_x or p_d or both are in $dom(h)$

3. $latest_{q \to p_y}(\downarrow h(T_u))$ and $latest_{q \to p_c}(\downarrow h(T_u))$ are comparable. Just like the previous point, this implies that either p_y or p_c or both are in $dom(h)$

4. $latest_{q \to p_y}(\downarrow h(T_u))$ and $latest_{q \to p_d}(\downarrow h(T_u))$ are comparable

   Just like previous point, this implies that either p_y or p_c or both are in $dom(h)$

So, for all to hold, one of the possible conditions must hold

- $p_x, p_y \in dom(f)$ OR
- $p_c, p_d \in dom(f)$ OR
- $p_x, p_y, p_c, p_d \in dom(f)$ OR
- $p_x, p_y, p_c \in dom(f)$ OR
- $p_x, p_y, p_d \in dom(f)$ OR
- $p_y, p_c, p_d \in dom(f)$ OR
- $p_x, p_c, p_d \in dom(f)$

$|dom(h)| \leq 2$ by definition of (Σ, θ)

Hence, last 5 options are not possible.

$p_x, p_y \in dom(h) \iff \lambda(h) = (p_x, p_y)$.

$p_c, p_d \in dom(h) \iff \lambda(h) = (p_c, p_d)$ by definition of (Σ, θ).

- Case 1:$\lambda(h) = (p_x, p_y)$

  Consider path ρ from e' to h. Since $dom(e') \cap dom(h) = \emptyset$, there is p_y-event or p_x-event apart from h in the ρ. Call this event η.

  $e' \leq \eta$ by definition of ρ. Since for all $p \in \mathcal{P}$, p-events are totally ordered, $e \leq \eta$. Hence h is not minimal. This is a contradiction

- Case 2:$\lambda(h) = (p_c, p_d)$

  Same reasoning ensures a contradiction

Hence, no possible conditions hold.

Hence, primary graph at h is not linear ordered set. Hence, active label of h-processes is bad. Hence, by previous lemma, active label of $max_p(T_u) = bad$ □

Chapter 7

Conclusion

The exponential blow-up in [2][3][4] is caused by the use of gossiping automaton. The thesis proved that register and ϵ-non-deterministic asynchronous automata can be constructed with polynomial size for $Path_n$. The register and ϵ-non-deterministic asynchronous automata did not 'avoid' solving the gossiping problem. In fact, the register and non-determinism was used to handle primary information in some sense. The register was used to keep track of the previous input letter. The non-determinism was used to guess the next input letter.

The necessity of gossiping problem to create asynchronous automata has not been proved. The necessity of the gossiping problem can arise when the accepting condition on a word is related to properties of the word's trace. A great example is $Path_n$. A word in $Path_n$ is accepted if and only if the trace is a path. If a language is defined based on properties of primary graphs of processes, then the accepting asynchronous automaton might need to solve gossiping problem. (Please note that every process has a primary graph during computation whether or not the process is designed to compute or store it)

It is not proved that a gossiping automaton can be constructed with size polynomial in number of processes. Believing that a gossiping automaton can be constructed with polynomial size in processes is difficult.

If there is a polynomial space algorithm changing a register asynchronous automaton into an asynchronous automaton accepting the same language, then the conjecture will be

proven false. However, this does not seem to be feasible.

Converting NFA to DFA has exponential blow-up. Hence, conversion of ϵ-non-deterministic asynchronous automata to asynchronous automata has exponential blow-up.

Converting an asynchronous automaton accepting $Path_n$ to locally rejecting asynchronous automaton accepting $Path_n$ in polynomial space is difficult. The culprit is the gossiping automaton. Each process must have good amount of information about its primary graph before locally rejecting. A process can only locally reject if its primary graph is not linear.

Creating a gossiping automaton with size linear in number of processes might prove the conjecture.

Bibliography

[1] Wieslaw Zielonka. Notes on finite asynchronous automata. *RAIRO-Theoretical Informatics and Applications*, 21(2):99–135, 1987.

[2] M Mukund and M Sohoni. Gossiping, asynchronous automata and zielonka's theorem. *Distributed Computing*, 10(3):117–127, 1997.

[3] Blaise Genest and Anca Muscholl. Constructing exponential-size deterministic zielonka automata. In *Automata, Languages and Programming: 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II 33*, pages 565–576. Springer, 2006.

[4] Blaise Genest, Hugo Gimbert, Anca Muscholl, and Igor Walukiewicz. Optimal zielonka-type construction of deterministic asynchronous automata. In *Automata, Languages and Programming: 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II 37*, pages 52–63. Springer, 2010.

[5] Robert Cori, Yves Métivier, and Wieslaw Zielonka. Asynchronous mappings and asynchronous cellular automata. *Information and Computation*, 106(2):159–202, 1993.