Chapter 9

Automata on Distributed Alphabets

Madhavan Mukund

Chennai Mathematical Institute
H1 SIPCOT IT Park, Padur PO
Siruseri 603103, India
E-mail: madhavan@cmi.ac.in

Traditional automata theory is an extremely useful abstraction for reasoning about sequential computing devices. For distributed systems, however, there is no clear consensus on how best to incorporate various features such as spatial independence, concurrency and communication into a formal computational model. One appealing and elegant approach is to have a network of automata operating on a distributed alphabet of local actions. Components are assumed to synchronize on overlapping actions and move independently on disjoint actions. We describe two formulations of automata on distributed alphabets, synchronous products and asynchronous automata, that differ in the degree to which distributed choices can be coordinated. We compare the expressiveness of the two models and provide a proof of Zielonka's fundamental theorem connecting regular trace languages to asynchronous automata. Along the way, we describe a distributed time-stamping algorithm that is basic to many interesting constructions involving these automata.

9.1. Introduction

Automata theory provides an extremely useful abstract description of sequential computing devices. A typical computer program manipulates variables. The *state* of the program is the set of values currently assigned to these variables. A computation is a sequence of steps that transforms a program from an initial state to a desired final state.

When we model programs in automata theory, we typically hide the concrete structure of states in terms of the variables used and their values and, instead, assign abstract names to these states. In the same way, we hide the specific nature of the transformations from one state to another and replace them by abstract actions.

In this article, we shift focus from traditional sequential computation to distributed computation. Our aim is to model programs that run on multiple computing devices and have to interact with each other in order to achieve their objective. In this setting, we need to model how programs interact and the way in which they

exchange information during these interactions.

There is no clear consensus on how best to incorporate various features such as spatial independence, concurrency and communication into a formal computational model. One appealing and elegant approach is to have a network of automata operating on a distributed alphabet of local actions. The components are assumed to synchronize on overlapping actions and move independently on disjoint actions.

In the most simplistic model, synchronizations serve only to coordinate the actions of independent components and no information is exchanged between components. We call such networks *synchronized products*.

A more elaborate model, proposed by Zielonka [1], is one in which processes share the information in their local states when they synchronize. This facility greatly enhances the computational power of the model. These automata, called *asynchronous automata*, have close connections with Mazurkiewicz trace theory, a language-theoretic formalism for studying concurrent systems [2].

The article begins with a quick introduction to transition systems and automata. We then define direct product automata, which are the building blocks of synchronized products. After establishing some characterization and closure properties of these models, we move on to asynchronous automata and their connection to trace languages. We describe a distributed time-stamping algorithm for these automata that is fundamental for many automata-theoretic results in trace theory. Using this tool we prove Zielonka's theorem that every regular trace language is recognized by an asynchronous automaton whose structure reflects the independence structure of the trace language.

9.2. Transition systems, automata and languages

As a computation evolves, the internal state of the computing device is transformed through a sequence of actions. We model this using labelled transition systems, in which we abstractly represent the various possible states of the system and the moves that the system makes from one state to another, labelled by an appropriate action.

Labelled transition systems Let Σ be a set of *actions*.

- A transition system over Σ is a triple $TS = (Q, \rightarrow, Q_{\text{in}})$ where Q is a set of *states*, $\rightarrow \subseteq Q \times \Sigma \times Q$ is the *transition relation* and $Q_{\text{in}} \subseteq Q$ is the set of *initial states*. We usually write $q \xrightarrow{a} q'$ to denote that $(q, a, q') \in \rightarrow$. As usual, a transition system is *deterministic* if the transition relation $\rightarrow$ satisfies the property that whenever $q \xrightarrow{a} q'$ and $q \xrightarrow{a} q''$, $q' = q''$.
- A (finite-state) automaton over Σ is a quadruple $A = (Q, \rightarrow, Q_{\text{in}}, F)$ where $(Q, \rightarrow, Q_{\text{in}})$ is a transition system with a finite set of states over a finite set of actions Σ, and $F \subseteq Q$ is a set of final states. An automaton is deterministic if the underlying transition system is.

Runs Let $A = (Q, \rightarrow, Q_{in}, F)$ be an automaton over Σ and let $w = a_1 a_2 \ldots a_n$ be a word in Σ^*. A run of A on w is a sequence of states $q_0 q_1 \ldots q_n$ such that $q_0 \in Q_{in}$ and for each $i \in [1..n]$, $q_{i-1} \xrightarrow{a_i} q_i$. (For natural numbers $m \leq n$, we write $[m..n]$ to denote the set $\{m, m+1, \ldots, n\}$.) This run is said to be *accepting* if $q_n \in F$.

The automaton A *accepts* or *recognizes* w if it admits at least one accepting run on w. The language of A, $L(A)$ is the set of all words over Σ that A recognizes.

9.3. Direct product automata

A large class of distributed systems can be modelled as networks of local transition systems whose moves are globally synchronized through common actions. To formalize this, we begin with the notion of a distributed alphabet.

Distributed alphabets A distributed alphabet over Σ, or a *distribution* of Σ, is a tuple of nonempty sets $\theta = \langle \Sigma_1, \Sigma_2, \ldots, \Sigma_k \rangle$ such that $\bigcup_{1 \leq i \leq k} \Sigma_i = \Sigma$. For each action $a \in \Sigma$, the *locations* of a with respect to the distribution θ is the set $loc_\theta(a) = \{i \mid a \in \Sigma_i\}$. If θ is clear from the context, we write just $loc(a)$ instead of $loc_\theta(a)$.

Direct product automaton Let $\langle \Sigma_1, \Sigma_2, \ldots, \Sigma_k \rangle$ be a distribution of Σ. For each $i \in [1..k]$, let $A_i = (Q_i, \rightarrow_i, Q_{in}^i, F_i)$ be an automaton over Σ_i. The *direct product automaton* $(A_1 \parallel A_2 \parallel \cdots \parallel A_k)$ is the automaton $A = (Q, \rightarrow, Q_{in}, F)$ over $\Sigma = \bigcup_{1 \leq i \leq k} \Sigma_i$, where:

- $Q = Q_1 \times Q_2 \times \cdots \times Q_k$.
- Let $\langle q_1, q_2, \ldots, q_k \rangle, \langle q_1', q_2', \ldots, q_k' \rangle \in Q$.
  Then $\langle q_1, q_2, \ldots, q_k \rangle \xrightarrow{a} \langle q_1', q_2', \ldots, q_k' \rangle$ if
    - For each $j \in loc(a)$, $q_j \xrightarrow{a}_j q_j'$.
    - For each $j \notin loc(a)$, $q_j = q_j'$.
- $Q_{in} = Q_{in}^1 \times Q_{in}^2 \times \ldots \times Q_{in}^k$.
- $F = F_1 \times F_2 \times \ldots \times F_k$.

Direct product language Let $\langle \Sigma_1, \Sigma_2, \ldots, \Sigma_k \rangle$ be a distribution of Σ. $L \subseteq \Sigma^*$ is said to be a direct product language if there is a direct product automaton $A = (A_1 \parallel A_2 \parallel \cdots \parallel A_k)$ such that $L = L(A)$.

Direct product languages can be precisely characterized in terms of their projections onto the local components of the system.

Projections Let $\langle \Sigma_1, \Sigma_2, \ldots, \Sigma_k \rangle$ be a distribution of Σ. For $w \in \Sigma^*$ and $i \in [1..k]$, the projection of w onto Σ_i is denoted $w\downarrow_{\Sigma_i}$ and is defined inductively as follows:

- $\varepsilon\downarrow_{\Sigma_i} = \varepsilon$, where ε is the empty string.
- $wa\downarrow_{\Sigma_i} = \begin{cases} (w\downarrow_{\Sigma_i})a & \text{if } a \in \Sigma_i \\ (w\downarrow_{\Sigma_i}) & \text{otherwise} \end{cases}$

Shuffle closure The shuffle closure of L with respect to $\langle \Sigma_1, \Sigma_2, \ldots, \Sigma_k \rangle$, $shuffle(L, \langle \Sigma_1, \Sigma_2, \ldots, \Sigma_k \rangle)$, is the set

$$\{w \in \Sigma^* \mid \forall i \in [1..k], \exists u_i \in L, w\downarrow_{\Sigma_i} = u_i\downarrow_{\Sigma_i}\}$$

As usual, we write just $shuffle(L)$ if $\langle \Sigma_1, \Sigma_2, \ldots, \Sigma_k \rangle$ is clear from the context.

Proposition 9.1. *Let $\langle \Sigma_1, \Sigma_2, \ldots, \Sigma_k \rangle$ be a distribution of Σ and let $L \subseteq \Sigma^*$ be a regular language. L is a direct product language iff $L = shuffle(L)$.*

Proof Sketch: $(\Rightarrow)$ Suppose that L is a direct product language. It is easy to see that $L \subseteq shuffle(L)$, so we show that $shuffle(L) \subseteq L$. Since L is a direct product language, there exists a direct product automaton $A = (A_1 \parallel A_2 \parallel \cdots \parallel A_k)$ such that $L = L(A)$.

Let $w \in shuffle(L)$. For each $i \in [1..k]$, there is a witness $u_i \in L$ such that $w\downarrow_{\Sigma_i} = u_i\downarrow_{\Sigma_i}$. Since $u_i \in L$, there is an accepting run $q \in Q_{in}^i \xrightarrow{u\downarrow_{\Sigma_i}}_i q_f \in F_i$ in A_i. Since this is true for every i, we can "glue" these runs together and construct an accepting run for A on w, so $w \in L(A) = L$.

$(\Leftarrow)$ Suppose that $L = shuffle(L)$. We prove that L is a direct product language. For $i \in [1..k]$, $L_i = L\downarrow_{\Sigma_i}$ is a regular language, since homomorphic images of regular languages are regular. For each $i \in [1..k]$, there exists a deterministic automaton A_i such that $L_i = L(A_i)$. It is then easy to see that $L = L(A_1 \parallel A_2 \parallel \cdots \parallel A_k)$. $\square$

Proposition 9.2. *Direct product languages are not closed under boolean operations.*

Example 9.3.
Let $\theta = \langle \{a\}, \{b\} \rangle$ and let $L = \{ab, ba, aabb, abab, abba, baab, baba, bbaa\}$. Then L is clearly the union of $\{ab, ba\}$ and $\{aabb, abab, abba, baab, baba, bbaa\}$, both of which are direct product languages. However, L is not itself a direct product language because $L \neq shuffle(L)$. For instance, $abb \in shuffle(L) \setminus L$.

9.4. Synchronized products

We can increase the expressiveness of product automata by removing the restriction that the final states are just the product of the local final states of each component. Instead, we permit an arbitrary subset of global states to be final states.

Synchronized product automaton Let $\langle \Sigma_1, \Sigma_2, \ldots, \Sigma_k \rangle$ be a distribution of Σ. For each $i \in [1..k]$, let $TS_i = (Q_i, \to_i, Q_{\text{in}}^i)$ be a transition system over Σ_i. The *synchronized product automaton* of $(TS_1, TS_2, \ldots, TS_k)$ is an automaton $A = (Q, \to, Q_{\text{in}}, F)$ over $\Sigma = \bigcup_{1 \le i \le k} \Sigma_i$, where:

- $Q = Q_1 \times Q_2 \times \cdots \times Q_k$
- Let $\langle q_1, q_2, \ldots, q_k \rangle, \langle q_1', q_2', \ldots, q_k' \rangle \in Q$.
  Then $\langle q_1, q_2, \ldots, q_k \rangle \xrightarrow{a} \langle q_1', q_2', \ldots, q_k' \rangle$ if
    - For each $j \in loc(a)$, $q_j \xrightarrow{a}_j q_j'$.
    - For each $j \notin loc(a)$, $q_j = q_j'$.
- $Q_{\text{in}} = Q_{\text{in}}^1 \times Q_{\text{in}}^2 \times \ldots \times Q_{\text{in}}^k$.
- $F \subseteq Q_1 \times Q_2 \times \ldots \times Q_k$.

Synchronized product language Let $\langle \Sigma_1, \Sigma_2, \ldots, \Sigma_k \rangle$ be a distribution of Σ. $L \subseteq \Sigma^*$ is said to be a synchronized product language if there is a synchronized product automaton A such that $L = L(A)$.

Example 9.4. The language defined in Example 9.3 is a synchronized product language. The synchronized product automaton for this language is shown in Figure 9.1. The set of global final states F is $\{\langle q_1, q_1' \rangle, \langle q_2, q_2' \rangle\}$.

Fig. 9.1. A synchronized product automaton for Example 9.3

Proposition 9.5. *A language is a synchronized product language if and only if it can be written as a finite union of direct product languages.*

Proof Sketch: ($\Rightarrow$) Let $A = (Q, \to, Q_{\text{in}}, F)$ be a synchronized product such that $\langle TS_1, TS_2, \ldots, TS_k \rangle$ are the component transition systems over $\langle \Sigma_1, \Sigma_2, \ldots, \Sigma_k \rangle$. For each $f = \langle f_1, f_2, \ldots, f_k \rangle \in F$, extend TS_i to an automaton $A_i^f = (TS_i, f_i)$ and construct the direct product $A_f = (A_1^f \parallel A_2^f \parallel \cdots \parallel A_k^f)$. Then, $L(A) = \bigcup_{f \in F} L(A_f)$.

($\Leftarrow$) Conversely, let L be a finite union of direct product languages $\{L_i\}_{i \in [1..m]}$, where each L_i is recognized by a direct product $A^i = (A_1^i \parallel A_2^i \parallel \cdots \parallel A_k^i)$. For $j \in [1..k]$, let $A_j^i = (Q_j^i, \to_j^i, Q_{\text{in}}^{ij}, F_j)$ be the jth component of A^i. We construct a synchronous product $\hat{A} = (\hat{A}_1 \parallel \hat{A}_2 \parallel \cdots \parallel \hat{A}_k)$ as follows. For each component j, we let $\hat{Q}_j$ be the disjoint union $\biguplus_{i \in [1..m]} Q_j^i$ and define the set of initial states of component j be $\bigcup_{i \in [1..m]} Q_{\text{in}}^{ij}$. The local transition relations of each component are

given by the union $\bigcup_{i \in [1..m]} \to_j^i$. The crucial point is to define the global set of final states as $(F_1^1 \times F_2^1 \times \cdots \times F_k^1) \cup (F_1^2 \times F_2^2 \times \cdots \times F_k^2) \cup \cdots \cup (F_1^m \times F_2^m \times \cdots \times F_k^m)$. This ensures that the synchronized product accepts only if all components agree on the choice of L_i. $\square$

Proposition 9.6. *Synchronized product languages are closed under boolean operations.*

Proof Sketch: Let L_1 and L_2 be synchronized product languages. Then, by definition, $L_1 = L_{11}' \cup L_{12}' \cup \cdots \cup L_{1k_1}'$ and $L_2 = L_{21}'' \cup L_{22}'' \cup \cdots \cup L_{2k_2}''$, where $\{L_{11}', L_{12}', \ldots, L_{1k_1}'\}$ and $\{L_{21}'', L_{22}'', \ldots, L_{2k_2}''\}$ are both sets of direct product languages. It is immediate that $L_1 \cup L_2$ is the union of these two collections, so $L_1 \cup L_2$ is a synchronized product language.

To show closure under complementation, we prove that any synchronized product language is recognized by a *deterministic* synchronized product automaton. If we assume this, we can complement a synchronized product language by exchanging final and non-final states. In other words, if L is a synchronized product language recognized by a deterministic synchronized product automaton $A = (Q, \to, Q_{\text{in}}, F)$, then $\bar{L}$, the complement of L, is recognized by the automaton $\bar{A} = (Q, \to, Q_{\text{in}}, Q \setminus F)$.

Let us assume we are working with respect to a distribution $\langle \Sigma_1, \Sigma_2, \ldots, \Sigma_k \rangle$ of Σ. Every synchronized product language L over Σ is a finite union $L_1 \cup L_2 \cup \cdots \cup L_m$ of direct product languages. We establish our claim by induction on m.

If $m = 1$, we have already seen that we can construct the direct product automaton $A = (A_1 \parallel A_2 \parallel \cdots \parallel A_k)$ recognizing L, where for $i \in [1..k]$, A_i is a deterministic automaton recognizing $L\downarrow_{\Sigma_i}$.

Now, let $L = L_1 \cup L_2 \cup \ldots \cup L_m$ where L_1 is a direct product language and $L' = L_2 \cup \ldots \cup L_m$ is a synchronized product language. We can assume that L_1 is recognized by a deterministic direct product automaton $A = (Q_A, \to_A, Q_{\text{in}}^A, F_A) = (A_1 \parallel A_2 \parallel \cdots \parallel A_k)$ and L', by the induction hypothesis, is recognized by a deterministic synchronized product $B = (Q_B, \to_B, Q_{\text{in}}^B, F_B)$ defined with respect to transition systems $\langle TS_1, TS_2, \ldots, TS_k \rangle$. For each $i \in [1..k]$, let $A_i = (Q_i, \to_i, Q_{\text{in}}^i, F_i)$ and $TS_i = (\hat{Q}_i, \Rightarrow_i, \hat{Q}_{in}^i)$. Define a new deterministic transition system $\widetilde{TS}_i$ with states $Q_i \times \hat{Q}_i$, initial states $\{(q_1, q_2) \mid q_1 \in Q_{\text{in}}^i, q_2 \in \hat{Q}_{in}^i\}$ and transitions of the form $(q_1, q_1') \xrightarrow{a} (q_2, q_2')$ iff $(q_1, a, q_1') \in \to_i$ and $(q_2, a, q_2') \in \Rightarrow_i$. Clearly, each $\widetilde{TS}_i$ is a deterministic transition system. We now construct a deterministic synchronized product automaton recognizing L from $\langle \widetilde{TS}_1, \widetilde{TS}_2, \ldots, \widetilde{TS}_k \rangle$ by setting $\widetilde{F} = ((F_1 \times \hat{Q}_1) \times (F_2 \times \hat{Q}_2) \times \cdots (F_k \times \hat{Q}_k)) \cup \{\langle (q_1, f_1), (q_2, f_2), \ldots, (q_k, f_k) \rangle \mid q_i \in Q_i, \langle f_1, f_2, \ldots, f_k \rangle \in F_B\}$. $\square$

Synchronized product automata are still not as expressive as we would like.

Example 9.7. Let $\theta = \langle \{a, c\}, \{b, c\} \rangle$. Then,

$$L = \left[\left[shuffle(\{ab\}) + shuffle(\{aabb\}) \right].c \right]^*$$

is not a synchronized product language.

Proof. If L is a synchronized product language, then L can be expressed as a finite union $L_1 \cup L_2 \cup \cdots \cup L_k$ of direct product languages. Let us write 0 for the word abc and 1 for word $aabbc$. Consider the following set of $k+1$ words of length k with at most one 1: $A_k = \{00\ldots0, 10\ldots0, 010\ldots0, \ldots, 00\ldots01\}$. By the pigeonhole principle, there must be two words $u, v \in A_k$ that belong to the same direct product component L_j, $j \in [1..k]$.

There are two cases to consider.

- Suppose that u and v differ at only one position. Then, without loss of generality, it must be the case that $u = 00\ldots0$ has no 1's. Let v have a 1 at position m, $m \in [1..k]$. Construct a new word $w = (abc)^{m-1}(abbc)(abc)^{k-m}$. It is easy to see that $w \downarrow_{\{a,c\}} = u \downarrow_{\{a,c\}}$ and $w \downarrow_{\{b,c\}} = v \downarrow_{\{b,c\}}$. So, $w \in shuffle(L_j)$ and hence $w \in L_j \subseteq L$ by Proposition 9.1, which is a contradiction.
- Suppose that u and v differ at two positions. Then u has a 1 at position m and v has a 1 at position m' for some $1 \leq m < m' \leq k$. Construct a word $w = (abc)^{m-1}(aabc)(abc)^{m'-m-1}(abbc)(abc)^{k-m'}$. Once again, it is easy to see that $w \downarrow_{\{a,c\}} = u \downarrow_{\{a,c\}}$ and $w \downarrow_{\{b,c\}} = v \downarrow_{\{b,c\}}$. So, $w \in shuffle(L_j)$ and hence $w \in L_j \subseteq L$ by Proposition 9.1, which is a contradiction. $\qquad\square$

9.5. Asynchronous automata

To construct a distributed automaton that can recognize the language from Example 9.7, we have to further enhance the structure of distributed automata.

In direct products and synchronized products, when an action a occurs, all components that participate in a must move simultaneously. However, each component is free to choose its local move independent of all other components. In other words, no information is exchanged between components at the time of synchronization.

For instance, if we try to recognize the language of Example 9.7 in our existing model, we have no way of preventing the c-move enabled after one a in the first component from synchronizing with the c-move enabled after two b's in the second component.

To overcome this limitation, Zielonka proposed an enriched definition of the transition relation for each letter a [1]. As usual, let $loc(a)$ denote the components that participate in a. Then, an a-state is a tuple that belongs to the product $\prod_{i \in loc(a)} Q_i$. Let Q_a denote the set of all a-states. We define the a-transition relation Δ_a to be a subset of $Q_a \times Q_a$. In other words, whenever an a occurs, all

the components that participate in a share information about their current local states and *jointly* decide on new local states for themselves. These automata are called *asynchronous automata*.[*]

Asynchronous automaton Let $\langle \Sigma_1, \Sigma_2, \ldots, \Sigma_k \rangle$ be a distribution of Σ. For each $i \in [1..k]$, let Q_i be a finite set of states. For each action $a \in \Sigma$, 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.

- $Q = Q_1 \times Q_2 \times \cdots \times Q_k$
- Let $\langle q_1, q_2, \ldots, q_k \rangle, \langle q'_1, q'_2, \ldots, q'_k \rangle \in Q$. Then $\langle q_1, q_2, \ldots, q_k \rangle \xrightarrow{a} \langle q'_1, q'_2, \ldots, q'_k \rangle$ if
  - For $loc(a) = \{i_1, i_2, \ldots, i_j\}$, $(\langle q_{i_1}, q_{i_2}, \ldots, q_{i_j} \rangle, \langle q'_{i_1}, q'_{i_2}, \ldots, q'_{i_j} \rangle) \in \Delta_a$.
  - For each $j \notin loc(a)$, $q_j = q'_j$.
- $Q_{\text{in}} = Q_{\text{in}}^1 \times Q_{\text{in}}^2 \times \ldots \times Q_{\text{in}}^k$.
- $F \subseteq Q_1 \times Q_2 \times \ldots \times Q_k$.

In other words, though each component has a set of local states as before, the transition relation for each action is *global* within the set of components where it occurs. A synchronized product automaton can be modelled as an asynchronous automaton by setting Δ_a to be the product $\prod_{i \in loc(a)} \xrightarrow{a}_i$.

Here is an asynchronous automaton for the language of Example 9.7.

Example 9.8. The states of components 1 and 2 are $\{q_0, q_1, q_2\}$ and $\{q'_0, q'_1, q'_2\}$, respectively. There is only one initial and one final state, which is $\langle q_0, q'_0 \rangle$ in both cases.

The transition relations are as follows:

- $\Delta_a = \{(q_0, q_1), (q_1, q_2)\}$
- $\Delta_b = \{(q'_0, q'_1), (q'_1, q'_2)\}$
- $\Delta_c = \{(\langle q_1, q'_1 \rangle, \langle q_0, q'_0 \rangle), (\langle q_2, q'_2 \rangle, \langle q_0, q'_0 \rangle)\}$.

In other words, the two components can reset their local states via c to q_0 and q'_0 only if they are jointly in the state $\langle q_1, q'_1 \rangle$ or $\langle q_2, q'_2 \rangle$. There is no move, for instance, from $\langle q_1, q'_2 \rangle$ via c back to $\langle q_0, q'_0 \rangle$.

9.6. Mazurkiewicz traces

A distributed alphabet induces an independence relation on actions—two actions a and b are independent if they occur on disjoint sets of processes. In all three

---

[*]The term asynchronous refers to the fact that components process their inputs locally, without reference to a shared global clock. The "exchange of information" at each move is instantaneous, so the communication between automata is actually synchronous!

models of distributed automata that we have considered so far, this means that an occurrence of a cannot enable or disable b, or vice versa.

Instead of deriving independence from a concrete distribution of the alphabet, we can begin with an alphabet equipped with an abstract independence relation. This is the starting point of the theory of traces, initiated by Mazurkiewicz as a language-theoretic formalism for studying concurrent systems [2].

Independence relation   An *independence relation* over Σ is a symmetric, irreflexive relation $I \subseteq \Sigma \times \Sigma$. An alphabet equipped with an independence relation (Σ, I) is called a *concurrent alphabet*.

It is clear that the natural independence relation $\mathcal{I}_{loc}$ induced by a distributed alphabet (Σ, loc), $(a, b) \in \mathcal{I}_{loc}$ iff $loc(a) \cap loc(b) = \emptyset$, is irreflexive and symmetric. The following example shows that different distributions may yield the same independence relation.

Example 9.9. Let $\Sigma = \{a, b, c, d\}$. The three distributions $\widetilde{\Sigma} = \langle \{a, c, d\}, \{b, c, d\} \rangle$, $\widetilde{\Sigma}' = \langle \{a, c, d\}, \{b, c\}, \{b, d\} \rangle$ and $\widetilde{\Sigma}'' = \langle \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c, d\} \rangle$ all give rise to the independence relation $I = \{(a, b), (b, a)\}$.

Given a concurrent alphabet $(\Sigma, \mathcal{I})$, there are several ways to construct a distributed alphabet (Σ, loc) so that the independence relation $\mathcal{I}_{loc}$ induced by loc coincides with $\mathcal{I}$.

We begin by building the dependence graph for $(\Sigma, \mathcal{I})$. Let $\mathcal{D} = (\Sigma \times \Sigma) \setminus \mathcal{I}$. $\mathcal{D}$ is called the *dependence relation*. Construct a graph $G_{\mathcal{D}} = (V_{\mathcal{D}}, E_{\mathcal{D}})$ with $V_{\mathcal{D}} = \Sigma$ and $(a, b) \in E_{\mathcal{D}}$ provided $(a, b) \in \mathcal{D}$.

One way to distribute Σ is to create a process p_e for every edge $e \in E_{\mathcal{D}}$. For each letter a, we then set $loc(a)$ to be the set of processes (edges) incident on the vertex a. Alternately, we can create a process p_C for each maximal clique C in $G_{\mathcal{D}}$. Then, for each letter a and each clique C, $p_C \in loc(a)$ iff the vertex labelled a belongs to C.

In both cases, it is easy to see that $\mathcal{I}_{loc} = \mathcal{I}$. So, we can go back and forth between a concurrent alphabet $(\Sigma, \mathcal{I})$ and a distributed alphabet (Σ, loc) whose induced independence relation $\mathcal{I}_{loc}$ is $\mathcal{I}$.

Trace equivalence   Let $(\Sigma, \mathcal{I})$ be the concurrent alphabet. $\mathcal{I}$ induces a natural *trace equivalence* $\sim$ on Σ^*: two words w and w' are related by $\sim$ iff w' can be obtained from w by a sequence of permutations of adjacent independent letters. More formally, $w \sim w'$ if there is a sequence of words $v_1, v_2, \ldots, v_k$ such that $w = v_1$, $w' = v_k$ and for each $i \in [1..k{-}1]$, there exist words u_i, u_i' and letters a_i, b_i satisfying

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

Proposition 9.10.

- *The equivalance relation $\sim$ is a congruence on Σ^* with respect to concatenation: If $u \sim u'$ then for any words w_1 and w_2, $w_1 u w_2 \sim w_1 u' w_2$.*
- *Both right and left cancellation preserve $\sim$-equivalence: $wu \sim wu'$ implies $u \sim u'$ and $uw \sim u'w$ implies $u \sim u'$.*

9.6.1. *Mazurkiewicz traces*

Equivalence classes of words   A *(Mazurkiewicz) trace* over $(\Sigma, \mathcal{I})$ is an equivalence class of words with respect to $\sim$. For $w \in \Sigma^*$, we write $[w]$ to denote the trace corresponding to w—$[w] = \{w' \mid w' \sim w\}$.

The intuition is that all words in a trace describe the same underlying computation of a concurrent system. Each word in a trace corresponds to a reordering of independent events. A more direct way to capture this intuition is to represent a trace as a labelled partial order.

Traces as labelled partial orders   A *(Mazurkiewicz) trace* over $(\Sigma, \mathcal{I})$ is a labelled partial order $t = (\mathcal{E}, \leq, \lambda)$ where

- $\mathcal{E}$ is a set of *events*
- $\lambda : \mathcal{E} \to \Sigma$ labels each event by a letter
- $\leq$ is a partial order over $\mathcal{E}$ satisfying the following conditions:
    - $(\lambda(e), \lambda(f)) \in D$ implies $e \leq f$ or $f \leq e$
    - $e \lessdot f$ implies $(\lambda(e), \lambda(f)) \in \mathcal{D}$, where

    $$\lessdot \ = \ < \setminus <^2 = \{(e, f) \mid e < f \ and \ \nexists g. \ e < g < f\}$$

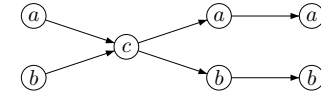   is the immediate successor relation in t.

Fig. 9.2.   The trace $[bacabba]$ as a labelled partial order

Example 9.11. 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 9.2 shows the trace $t = (\mathcal{E}, \leq, \lambda)$ corresponding to the word $bacabba$. The arrows between the events denote the relation $\lessdot$.

If we represent a trace as a labelled partial order, the set of linearizations of this partial order form an equivalence class $[w]$ of words with respect to $\mathcal{I}$. Conversely, it is not difficult to show that each equivalence class $[w]$ generates a unique labelled

partial order of which it is the set of linearizations. In fact, the way in which $\leq$ is generated from the dependence relation $\mathcal{D}$ allows us to construct the labelled partial order corresponding to a trace from a single linearization.

Trace languages Traces, like words, form a monoid under concatenation, with the empty trace as the unit. The concatenation operation is easiest to define in terms of the labelled partial order representation—given two traces $t_1 = (\mathcal{E}_1, \leq_1, \lambda_1)$ and $t_2 = (\mathcal{E}_2, \leq_2, \lambda_2)$, the trace $t_i \cdot t_2$ is the trace $t' = (\mathcal{E}', \leq', \lambda')$ where $\mathcal{E}' = \mathcal{E}_1 \cup \mathcal{E}_2$, $\lambda'(e) = \lambda_1(e)$ if $e \in \mathcal{E}_1$ and $\lambda_2(e)$ if $e \in \mathcal{E}_2$, and $\leq'$ is generated by $\lessdot_1 \cup \lessdot_2 \cup \{(x, y) \mid x \text{ maximal in } \mathcal{E}_1, y \text{ minimal in } \mathcal{E}_2, (\lambda_1(x), \lambda_2(y)) \in \mathcal{D}\}$.

A *trace language* is a set of traces or, alternatively a subset of the trace monoid. However, we prefer to treat trace languages as string languages which satisfy a closure condition. We say that $L \subseteq \Sigma^*$ is a trace language if L is $\sim$-*consistent*—i.e., for each $w \in \Sigma^*$, w is in L iff every word in $[w]$ is in L. Since traces correspond to equivalence classes of strings, there is a 1-1 correspondence between subsets of the trace monoid and $\sim$-consistent languages over Σ^*.

In the string framework, we say a trace language L is recognizable if it is accepted by a finite-state automaton. Once again, it is not difficult to show that there is a 1-1 correspondence between recognizable subsets of the trace monoid and recognizable $\sim$-consistent languages over Σ^* (see, for instance, [3]).

Henceforth, whenever we use the terms trace language and recognizable trace language, we shall be referring to the definitions in terms of $\sim$-consistent subsets of Σ^* rather than in terms of subsets of the trace monoid.

One of the most fundamental results in trace theory says that every recognizable trace language has a distributed implementation in terms of asynchronous automata.

Theorem 9.12 (Zielonka). *Let L be a recognizable trace language over a concurrent alphabet $(\Sigma, \mathcal{I})$. Then, for every distributed alphabet (Σ, loc) such that $\mathcal{I}_{loc} = \mathcal{I}$, we can construct an asynchronous automaton A over (Σ, loc) with $L(A) = L$.*

To prove Zielonka's theorem, we require a distributed timestamping algorithm that is fundamental for many constructions involving asynchronous automata.

9.7. Distributed time-stamping

Let $\mathcal{P} = \{p_1, p_2, \dots, p_N\}$ be a set of processes which synchronize with each other from time to time and exchange information about themselves and others. The problem we look at is the following: whenever a set $P \subseteq \mathcal{P}$ synchronizes, the processes in P must decide *amongst themselves* which of them has the latest information, direct or indirect, about each process p in the system. We call this the *gossip problem*.

A naïve solution to the gossip problem is to label interactions using a counter whose value increases as time progresses. There are two problems with this approach.

- The counter values (or time-stamps) grow without bound and cannot be generated and stored by finite-state automata.
- Each interaction involves only a subset of processes, so time-stamps have to be generated in a distributed manner.

Our goal is to develop an algorithm to solve the gossip problem that is both finite state and local.

We model the interactions of the processes in $\mathcal{P}$ by a distributed alphabet (Σ, loc) that contains an action X for each nonempty subset $X \subseteq \mathcal{P}$, with the obvious distribution function $loc(X) = X$. A sequence of interactions between the processes is then a trace over (Σ, loc).

Let $t = (\mathcal{E}, \leq, \lambda)$ be a trace representing a computation of our system. For convenience, we assume that t always begins with an initial event $e_\perp$ labelled by $\mathcal{P}$, in which all processes participate. This models the initial exchange of information that is implicit in the fact that all processes agree on a global initial state.

Process-wise ordering Let $t = (\mathcal{E}, \leq, \lambda)$ be a trace. For each event $e \in \mathcal{E}$, we write $p \in e$ to mean that $p \in loc(\lambda(e)) = \lambda(e)$. Each process p orders the events in which it participates. Let us define $\lessdot_p$ to be the strict ordering

$$e \lessdot_p f \stackrel{\text{def}}{=} e < f, \ p \in e \cap f \text{ and for all } e < g < f, \ p \notin g.$$

It is clear that set of all p-events in $\mathcal{E}$ is totally ordered by $\leq_p$, the reflexive, transitive closure of $\lessdot_p$. It is also easy to observe that the overall partial order $\leq$ is generated by $\{\lessdot_p\}_{p \in P}$. If $e \leq f$ we say that e is *below* f. Note that the special event $e_\perp$ is always below every event in t.
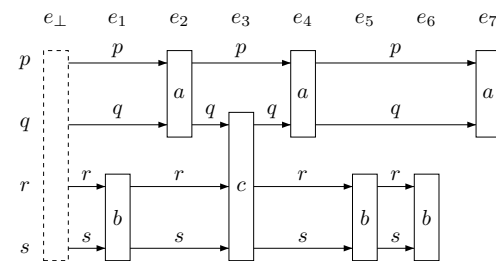

Fig. 9.3. Process-wise ordering in a trace

Example 9.13. 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 9.3 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.

9.7.1. *Ideals*

The main source of difficulty in solving the gossip problem is the fact that the processes in $\mathcal{P}$ need to compute the global information about a trace t while each process only has access to a local, partial view of t. Although partial views of t correspond to subsets of $\mathcal{E}$, not every subset of $\mathcal{E}$ arises from such a partial view. Those subsets of $\mathcal{E}$ which do correspond to partial views of u are called ideals.

Ideals A set of events $I \subseteq \mathcal{E}$ is called an *(order) ideal* if $e \in I$ and $f \leq e$ then $f \in I$.

The requirement that an ideal be downward closed with respect to $\leq$ guarantees that the observation it represents is consistent—whenever an event e has been observed, so have all the events in the computation which necessarily precede e.

Because of our interpretation of $e_\perp$ as an event which takes place *before* the actual computation begins, the minimum possible partial view of a word u is the ideal $\{e_\perp\}$. Henceforth, we assume that every ideal I we consider is non-empty and contains $e_\perp$.

Example 9.14. Let us look once again at Figure 9.3. $\{e_\perp, e_2\}$ is an ideal, but $\{e_\perp, e_2, e_3\}$ is not, since $e_1 \leq e_3$ but $e_1 \notin \{e_\perp, e_2, e_3\}$.

The following observations are immediate.

Proposition 9.15. *Let $t = (\mathcal{E}, \leq, \lambda)$ be a trace.*

- *$\mathcal{E}$ is an ideal.*
- *For any $e \in \mathcal{E}$, the set $\downarrow e = \{f \in \mathcal{E} \mid f \leq e\}$ is an ideal, called the* principal *ideal* generated by e. *The events in $\downarrow e$ are the only events in $\mathcal{E}$ that are "known" to the processes in e when e occurs.*
- *An ideal I is said to be generated by a set of events X if $I = \bigcup_{e \in X} \downarrow e$. Any ideal I is generated by its maximal events $I_{\max} = \{e \mid e \text{ is } \leq\text{-maximal in } I\}$.*
- *If I and J are ideals then $I \cup J$ and $I \cap J$ are ideals.*

Example 9.16. In Figure 9.3, $\{e_\perp, e_1, e_2, e_3, e_5\}$ is the principal ideal $\downarrow e_5$. The ideal $\{e_\perp, e_1, e_2, e_3, e_4, e_5\}$ is generated by $\{e_4, e_5\}$.

Views Let I be an ideal. The maximum p-event in I, $\max_p(I)$, is the last event in I in which p has taken part. In other words, for every $e \in I$, if e is a p-event then $e \leq \max_p(I)$. Since all p-events are totally ordered by $\leq$, and $p \in e_\perp \in I$ for all processes p and for all ideals I, $\max_p(I)$ is well defined.

Let I be an ideal. The *p-view* of I, $\partial_p(I)$ is the set $\downarrow\max_p(I)$. For $P \subseteq \mathcal{P}$, the P-view of I, $\partial_P(I)$, is the *joint view* $\bigcup_{p \in \mathcal{P}} \partial_p(I)$.

Example 9.17. In Figure 9.3, let I denote the ideal $\{e_\perp, e_1, e_2, e_3, e_4, e_5, e_6\}$. $\max_q(I) = e_4$ and hence $\partial_q(I) = \{e_\perp, e_1, e_2, e_3, e_4\}$. On the other hand, though $\max_r(I) = e_6$, $\partial_r(I) \neq I$. Rather, $\partial_r(I) = I \setminus \{e_4\}$. The joint view $\partial_{\{q,r\}}(I) = I = \partial_{\mathcal{P}}(I)$.

9.7.2. *Primary and secondary information*

Latest information Let $p, q \in \mathcal{P}$ and I be an ideal. The latest information p has about q in I is $\max_q(\partial_p(I))$, the $\leq$-maximum q-event in $\partial_p(I)$. We denote this event by $latest_{p \to q}(I)$. Observe that $latest_{p \to p}(I) = \max_p(I)$.

Example 9.18. In Figure 9.3, $\max_p(\mathcal{E}) = e_7$ whereas $\max_s(\mathcal{E}) = e_6$. We have $latest_{p \to q}(\mathcal{E}) = e_7$, $latest_{p \to s}(\mathcal{E}) = e_3$ and $latest_{s \to p}(\mathcal{E}) = e_2$.

Primary information Let I be an ideal and $p, q \in \mathcal{P}$. The *primary information* of p after I, $primary_p(I)$, is the set $\{latest_{p \to q}(I)\}_{q \in \mathcal{P}}$. In other words this is the best information that p has about every other process in I. As usual, for $P \subseteq \mathcal{P}$, $primary_P(I) = \bigcup_{p \in P} primary_p(I)$.

More precisely, $primary_p(I)$ is an *indexed* set of events—each event $e = latest_{p \to q}(I)$ in $primary_p(I)$ is represented as a triple (p, q, e). However, we will often ignore the fact that $primary_p(I)$ is an *indexed* set of events and treat it, for convenience, as just a set of events. Thus, for an event $e \in I$, we shall write $e \in primary_p(I)$ to mean that there exists a process $q \in \mathcal{P}$ such that $(p, q, e) \in primary_p(I)$, and so on.

The task at hand is to design a mechanism for processes to compare their primary information when they synchronize. When two processes p and q synchronize, their joint view of the current computation is $\partial_{\{p,q\}}(\mathcal{E})$. At this point, for every other process r, p has in its local information an event $e_r = latest_{p \to r}(\mathcal{E})$ and q has, similarly, an event $e'_r = latest_{q \to r}(\mathcal{E})$. After the current event e, both the p-view and the q-view will correspond to the principal ideal $\downarrow e \subseteq \mathcal{E}$. Assuming that r does not participate in e, $\max_r(\downarrow e)$ will be one of e_r and e'_r. So, p and q have to be able to locally decide whether $e_r \leq e'_r$ or vice versa. If $e_r < e'_r$, then $e_r \in \partial_p(\mathcal{E}) \cap \partial_q(\mathcal{E})$ while $e'_r \in \partial_q(\mathcal{E}) \setminus \partial_p(\mathcal{E})$. Thus, updating primary information is equivalent to computing whether a primary event lies within the intersection $\partial_p(\mathcal{E}) \cap \partial_q(\mathcal{E})$ or outside.

Our first observation is a characterization of the maximal events in the intersection in terms of primary information.
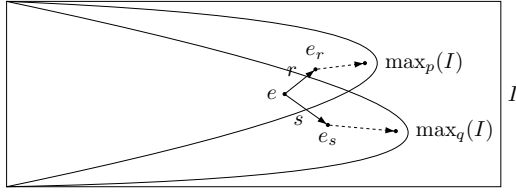
Fig. 9.4.   Maximal events in the intersection of two ideals are primary events

Lemma 9.19. *The maximal elements of $\partial_p(I) \cap \partial_q(I)$ are a subset of $primary_p(I) \cap primary_q(I)$.*

Proof.   If $\max_p(I) \leq \max_q(I)$ then $\partial_p(I) = \partial_p(I) \cap \partial_q(I)$ and the only maximal event in this intersection is $e = \max_p(I)$ which is $latest_{p \to p}(I)$ and $latest_{q \to p}(I)$ and hence in $primary_p(I) \cap primary_q(I)$. A symmetric argument holds if $\max_q(I) \leq \max_p(I)$.

The nontrivial case arises when $\max_p(I)$ and $\max_q(I)$ are incomparable. Let e be a maximal event in the intersection. Since $e \leq \max_p(I)$ and $e \leq \max_q(I)$, there is a "path" from e to $\max_p(I)$ in the trace, and another path from e to $\max_q(I)$. So, for some $r, s \in \mathcal{P}$, e, we must have an r event $e_r \in \partial_p(I) \setminus \partial_q(I)$ and an s event $e_s \in \partial_q(I) \setminus \partial_p(I)$ such that $e \lessdot_r e_r \leq \max_p(I)$ and $e \lessdot_s e_s \leq \max_q(I)$ (see Figure 9.4). Notice that e itself is both an r-event and an s-event. Since $e \in \partial_q(I)$ but $e_r \notin \partial_q(I)$ and $e \lessdot_r e_r$, it follows that $e = latest_{q \to r}(I)$. By a symmetric argument, $e = latest_{p \to s}(I)$, so $e \in primary_p(I) \cap primary_q(I)$.   □

This characterization yields the following result.

Lemma 9.20. *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 $e \leq f$ iff there exists $g \in primary_p(I) \cap primary_q(I)$ such that $e \leq g$.*

Proof.   Clearly $e \leq f$ iff $e \in \partial_p(I) \cap \partial_q(I)$ iff e is dominated by some maximal element in $\partial_p(I) \cap \partial_q(I)$ iff, by Lemma 9.19, there exists $g \in primary_p(I) \cap primary_q(I)$ such that $e \leq g$.   □

To effectively perform the comparison suggested by Lemma 9.20, each process maintains the partial order between events in its primary information.

Primary graph   Let I be an ideal and $p \in \mathcal{P}$. The *primary graph* of p in I is the set of primary events together with the partial order between them inherited from the underlying trace.

With primary graphs, it is clear how to perform the comparison indicated in Lemma 9.20. Processes p and q identify an event g that is common to both their

sets of primary information such that $e \leq g$ in p's primary graph. In this manner, p and q can decide for every other process r which of $latest_{p \to r}(I)$ and $latest_{q \to r}(I)$ is better. To complete the update, we have to rebuild the primary graph after updating the primary information of p and q.

Lemma 9.21. *The primary graphs of p and q can be locally reconstructed after updating primary infomation.*

Proof.   Let e and f be two events in the updated primary information of p and q (recall that both processes have the *same* primary information after synchronization). If both e and f were inherited from the same process, say p, we order them in the new graph if and only if they were ordered in the original primary graph of p.

The interesting case is when e is inherited from p and f from q. In this case, we must have had $e \in \partial_p(I) \setminus \partial_q(I)$ and $f \in \partial_q(I) \setminus \partial_q(I)$. This means that e and f were not ordered in I, so we do not order them in the updated primary graph.   □

This procedure generalizes to any arbitrary set $P \subseteq \mathcal{P}$ which synchronizes after a trace t. The processes in P share their primary graphs and compare this information pairwise. Using Lemma 9.20, for each $q \in \mathcal{P} \setminus P$ they decide who has the "latest information" about q and correctly order these events. Each process then comes away with the *same* primary graph, incorporating the best information available among the processes in P.

9.7.3. *Labelling events consistently*

To make Lemma 9.20 effective, we must make the assertions "locally checkable"— for example, if $e = latest_{p \to r}(I)$ and $f = latest_{q \to r}(I)$, processes p and q must be able to decide if there exists an event $g \in \partial_p(I) \cap \partial_q(I)$ between e and f. This can be checked locally provided events in $\mathcal{E}$ are labelled unambiguously.

Since events are processed locally, we must locally assign labels to events in $\mathcal{E}$ so that we can check whether events in the primary graphs of two different processes are equal by comparing their labels. A naïve solution would be for the processes in $loc(a)$ to jointly assign a (sequential) time-stamp to each new occurrence of a, for every letter a. The problem with this approach is that we will need an unbounded set of time-stamps, since u could get arbitrarily large.

Instead we would like a scheme that uses only a finite set of labels to distinguish events. This means that several different occurrences of the same action will eventually get the same label. Since updating primary graphs relies on comparing labels, we must ensure that this reuse of labels does not lead to any confusion.

However, from Lemma 9.20, we know that to compare primary information, we only need to look at the events which are currently in the primary sets of each process. So, it is sufficient if the labels assigned to these sets are consistent across

the system—that is, if the same label appears in the current primary information of different processes, then this label does in fact denote the same event in underlying trace.

When a new action a occurs, the processes in $loc(a)$ have to assign it a label that is different from all a-events that are currently in the primary information of all processes. Since the cardinality of $primary_{\mathcal{P}}(\mathcal{E})$ is bounded, such a new label must exist. The catch is to detect which labels are currently in use and which are not.

Unfortunately, the processes in a cannot directly see all the a-events which belong to the primary information of the entire system. An a-event e may be part of the primary information of processes *outside* a—that is, $e \in primary_{\mathcal{P}\setminus a}(\mathcal{E}_u) \setminus primary_a(\mathcal{E}_u)$.
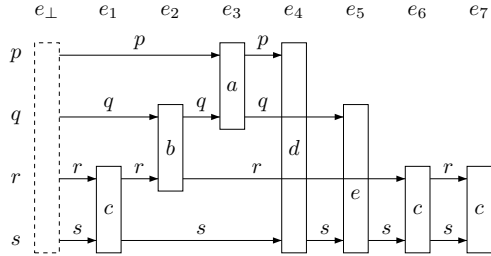
Fig. 9.5.   Keeping track of active labels

Example 9.22. 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 9.5 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 $primary_r(\mathcal{E}) = \{(r, p, e_4), (r, q, e_5), (r, r, e_7), (r, s, e_7)\}$.

To enable the processes in a to know about all a-events in $primary_{\mathcal{P}}(\mathcal{E}_u)$, we need to maintain secondary information.

Secondary information   The *secondary information* of p after I, $secondary_p(I)$, is the (indexed) set $\bigcup_{q \in \mathcal{P}} primary_q(\downarrow latest_{p \to q}(I))$. In other words, this is the latest information that p has in I about the primary information of q, for each $q \in \mathcal{P}$. Once again, for $P \subseteq \mathcal{P}$, $secondary_P(I) = \bigcup_{p \in P} secondary_p(I)$.

Each event in $secondary_p(I)$ is of the form $latest_{q \to r}(\downarrow latest_{p \to q}(I))$ for some $q, r \in \mathcal{P}$. This is the latest r-event which q knows about up-to the event $latest_{p \to q}(I)$. We abbreviate $latest_{q \to r}(\downarrow latest_{p \to q}(I))$ by $latest_{p \to q \to r}(I)$.

Just as we represented events in $primary_p(I)$ as triples of the form (p, q, e), where $p, q \in \mathcal{P}$ and $e \in I$, we represent each secondary event $e = latest_{p \to q \to r}(I)$ in $secondary_p(I)$ as a quadruple (p, q, r, e). Recall that we often ignore the fact that $primary_p(I)$ and $secondary_p(I)$ are *indexed* sets of events and treat them, for convenience, as just sets of events.

Notice that each primary event $latest_{p \to q}(I)$ is also a secondary event $latest_{p \to p \to q}(I)$ (or, equivalently, $latest_{p \to q \to q}(I)$). So, following our convention that $primary_p(I)$ and $secondary_p(I)$ be treated as sets of events, we write $primary_p(I) \subseteq secondary_p(I)$.
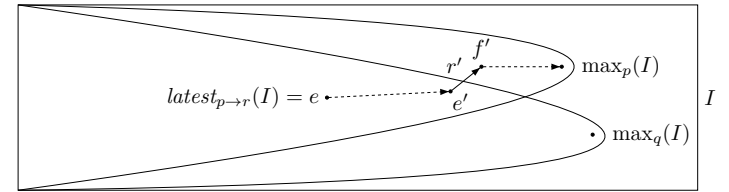

Fig. 9.6.   Identifying active labels through secondary information

Lemma 9.23. *Let I be an ideal and $p \in \mathcal{P}$. If $e \in primary_p(I)$ then for every $q \in e$, $e \in secondary_q(I)$.*

Proof.   Let $e = latest_{p \to r}(I)$ for some $r \in \mathcal{P}$ and let $q \in e$. We will show that $e = latest_{q \to r' \to r}(I)$ for some $r' \in \mathcal{P}$. We know that there is a path $e \lessdot f_1 \lessdot \cdots \lessdot \max_p(I)$, since $e \in \partial_p(I)$. This path starts inside $\partial_p(I) \cap \partial_q(I)$.

If this path never leaves $\partial_p(I) \cap \partial_q(I)$ then $\max_p(I) \in \partial_q(I)$. Since $\max_p(I)$ is the maximum p-event in I, it must be the maximum p-event in $\partial_q(I)$. So, $e = \max_r(\max_p(I)) = \max_r(\max_p(\partial_q(I))) = latest_{q \to p \to r}(I)$ and we are done.

If this path does leave $\partial_p(I) \cap \partial_q(I)$, we can find an event e' along the path such that $e \leq e' \lessdot_{r'} f' \leq \max_p(I)$, where $e' \in \partial_p(I) \cap \partial_q(I)$, $f' \in \partial_p(I) \setminus \partial_q(I)$ and $r' \in e' \cap f'$ (see Figure 9.6). It is easy to see that $e' = latest_{q \to r'}(I)$. Since $e = \max_r(\partial_p(I))$, $e \in \downarrow e' \subseteq \partial_p(I)$, we have $e = \max_r(\downarrow e')$. Hence $e = latest_{r' \to r}(\downarrow e') = latest_{q \to r' \to r}(I)$. $\qquad \square$

Corollary 9.24. *Let I be an ideal, $p \in \mathcal{P}$ and e be a p-event in I. If $e \notin secondary_p(I)$ then $e \notin primary_{\mathcal{P}}(I)$.*

So, a process p can keep track of which of its labels are "in use" in the system by maintaining secondary information. Each p-event e initially belongs to $primary_e(I)$, and hence to $secondary_e(I)$, where we also use e to denote the subset of $\mathcal{P}$ that

synchronizes at this event. As the computation progresses, e gradually "recedes" into the background and disappears from the primary sets of the system. Eventually, when e disappears from $secondary_p(I)$, p can be sure that e no longer belongs to $primary_{\mathcal{P}}(I)$.

Since $secondary_p(I)$ is a bounded set, p knows that only finitely many of its labels are in use at any given time. So, by using a sufficiently large finite set of labels, each new event can always be assigned an unambiguous label by the processes which take part in the event.

It is easy to see that secondary information can be updated along with primary information. If $latest_{p \rightarrow r}(I)$ is better than $latest_{q \rightarrow r}(I)$, then all secondary events of the form $latest_{q \rightarrow r \rightarrow s}(I)$ should also be replaced by the corresponding events $latest_{p \rightarrow r \rightarrow s}(I)$.

9.7.4. The "gossip" automaton

Using our analysis of the primary graph and secondary information maintained by processes, we can now design a deterministic asynchronous automaton to consistently update the primary information of each process whenever a set of processes synchronize.

For $p \in \mathcal{P}$, each local state of p will consist of its primary graph and secondary information, stored as indexed collections or arrays. Each event in these arrays is represented as a pair $\langle P, \ell \rangle$, where P is the subset of processes that synchronized at the event and $\ell \in \mathcal{L}$, a finite set of labels. We shall establish a bound on $|\mathcal{L}|$ shortly.

The initial state is the global state where for all processes p, all entries in these arrays correspond to the initial event $e_\perp$. The event $e_\perp$ is denoted by $\langle \mathcal{P}, \ell_0 \rangle$ for an arbitrary but fixed label $\ell_0 \in \mathcal{L}$.

The local transition functions $\rightarrow_a$ modify the local states for processes in a as follows.

(i) When a new a-labelled event e occurs after u, the processes in a assign a label $\langle a, \ell \rangle$ to e which does not appear in $secondary_a(\mathcal{E}_u)$. Corollary 9.24 guarantees that this new label does not appear in $primary_{\mathcal{P}}(\mathcal{E}_u)$.
   Let $N = |\mathcal{P}|$. Since each process keeps track of N^2 secondary events and at most N processes can synchronize at an event, there need be only N^3 labels in $\mathcal{L}$. (In fact, in Lemma 9.25 below, we show that it suffices to have N^2 labels in $\mathcal{L}$.)

(ii) The processes participating in e now share their primary graphs and update their primary information about each process $q \notin e$, as described in Lemma 9.20.

The gossip automaton does not "recognize" a set of traces. Rather, it updates the primary graphs and secondary information of all processes appropriately whenever an action is added to the trace, such that at any point in the computation, we can compare the primary information of a set of processes using the information present in their local states.

Lemma 9.25. *In the gossip automaton, the number of local states of each process* $p \in \mathcal{P}$ *is at most* $2^{O(N^2 \log N)}$*, where* $N = |\mathcal{P}|$*.*

Proof. A local state for p consists of its primary graph and secondary information. We estimate how many bits are required to store this.

Recall that for any ideal I, each event in $primary_p(I)$ is also present in $secondary_p(I)$. So it suffices to store just the labels of secondary events. These events are stored in an array with N^2 entries, where each entry is implicitly indexed by a pair from $\mathcal{P} \times \mathcal{P}$. We can store the primary graph as an adjacency matrix with N^2 entries.

Each new event e is assigned a label of the form $\langle P, \ell \rangle$, where P was the set of processes that participated in e and $\ell \in \mathcal{L}$.

We argued earlier that it suffices to have N^3 labels in $\mathcal{L}$. Actually, we can make do with N^2 labels by modifying our transition function slightly. When a letter a is read, instead of immediately labelling the new event, the processes in a first compare and update their primary, secondary information about processes from $\mathcal{P} \setminus a$. These updates concern events which have already been labelled, so the fact that the new event has not yet been labelled is not a problem. Once this is done, all the processes in a will have the same primary and secondary information. At this stage, there are (less than) N^2 distinct labels present in $secondary_a(I)$. So, if $|\mathcal{L}| = N^2$ the processes are guaranteed to find a label they can use for the new event. Regardless of which update strategy we choose, $\ell \in \mathcal{L}$ can be written down using $O(\log N)$ bits.

To write down $P \subseteq \mathcal{P}$, we need, in general, N bits. This component of the label is required to guarantee that all secondary events in the system have distinct labels, since the set $\mathcal{L}$ is common across all processes. However, we do not really need to use all of P in the label for e to ensure this property. If we order $\mathcal{P}$ as $\{p_1, p_2, \ldots, p_N\}$, it suffices to label e by $\langle p_i, \ell \rangle$ where, among the processes in P, p_i has the least index with respect to our ordering of $\mathcal{P}$.

Thus, we can modify our automaton so that the processes label each event by a pair $\langle p, \ell \rangle$, where $p \in \mathcal{P}$ and $\ell \in \mathcal{L}$. This pair can be written down using $O(\log N)$ bits. Overall there are N^2 such pairs in the array of secondary events, so the this can be described using $O(N^2 \log N)$ bits and the primary graph can be represented by $O(N^2)$ bits. Therefore, the number of distinct local states of p is bounded by $2^{O(N^2 \log N)}$. $\square$

9.8. Zielonka's Theorem

It is not difficult to see that any language accepted by an asynchronous automaton over (Σ, loc) is a recognizable trace language over the corresponding concurrent alphabet $(\Sigma, \mathcal{I}_{loc})$. The distributed nature of the automaton guarantees that it is a trace language. To see that the language is recognizable, we note that every asynchronous automaton $\mathcal{A} = (Q, \rightarrow, Q_{in}, F)$ gives rise to a finite state automaton

$\mathcal{B}$ accepting the same language as $\mathcal{A}$. The states of $\mathcal{B}$ are the global states of $\mathcal{A}$ and the transition relation of $\mathcal{B}$ is given by the global transition relation of $\mathcal{A}$. Since the initial and accepting states of $\mathcal{A}$ are specified as global states, they can directly serve as the initial and final states of $\mathcal{B}$. It is straightforward to verify that $\mathcal{B}$ accepts the same language as $\mathcal{A}$.

On the other hand, the converse is difficult to show. For a given recognizable trace language L over a concurrent alphabet $(\Sigma, \mathcal{I})$, does there exist an asynchronous automaton $\mathcal{A}$ over a distributed alphabet (Σ, loc) such that $\mathcal{A}$ accepts L *and* the independence relation $\mathcal{I}_{loc}$ induced by *loc* is *exactly* $\mathcal{I}$?

Zielonka's fundamental result is that this is indeed the case [1]. In other words, asynchronous automata accept precisely the set of recognizable trace languages and thus constitute a natural distributed machine model for this class of languages.

Fix a recognizable trace language L over a concurrent alphabet $(\Sigma, \mathcal{I})$, as well as a distribution $loc : \Sigma \to (2^{\mathcal{P}} \setminus \{\emptyset\})$ such that the induced independence relation $\mathcal{I}_{loc}$ is the same as $\mathcal{I}$. We shall construct a deterministic asynchronous automaton $\mathcal{A} = (Q, \to, Q_{\mathrm{in}}, F)$ over (Σ, loc) recognizing L.

Let $\mathcal{B} = (S, \Sigma, \delta, s_0, S_F)$ be the minimum deterministic finite state automaton accepting L, where S denotes the set of states, $\delta : S \times \Sigma \to S$ the transition function, $s_0 \in S$ the initial state and $S_F \subseteq S$ the set of accepting states. As usual, we shall extend δ to a transition function $S \times \Sigma^* \to S$ describing state transitions for input words rather than just single letters. For convenience, we denote this extended transition function also by δ.

The main hurdle in constructing an asynchronous automaton $\mathcal{A}$ from the original DFA $\mathcal{B}$ is the following: On reading an input word u, we must be able to compute whether $\delta(s_0, u) \in S_F$. As we have seen, u is just one linearization of the trace $[u] = (\mathcal{E}_u, \leq_u, \lambda_u)$. After reading u each process in $\mathcal{A}$ only has partial information about $\delta(s_0, [u])$—a process p only "knows about" the events that lie in the p-view $\downarrow \max_p(\mathcal{E}_u)$. We have to devise a scheme to recover the state $\delta(s_0, u)$ from the partial information available with each process after reading $[u]$.

Another complication is that processes can only maintain a finite amount of information. So, we need a way of representing arbitrary words in a bounded, finite way. This can be done quite easily—the idea is to record for each word w, its "effect" as dictated by our automaton $\mathcal{B}$.

We first recall a basic fact about recognizable languages.

Definition 9.26. Any language $\hat{L}$ defines a syntactic congruence $\equiv_{\hat{L}}$ on Σ^* as follows:

$$u \equiv_{\hat{L}} u' \overset{\mathrm{def}}{=} \forall w_1, w_2 \in \Sigma^*, \ w_1 u w_2 \in \hat{L} \ iff \ w_1 u' w_2 \in \hat{L}.$$

It is well known that $\hat{L}$ is recognizable if and only if $\equiv_{\hat{L}}$ is of finite index [4].

Now, consider the relation $\equiv_L$ for the language L we are looking at. We can associate with each word u a function $f_u : S \to S$, where S is the set of states of $\mathcal{B}$, such that $f_u(s) = s'$ iff $\delta(s, u) = s'$. Thus, f_u is a representation of the word u as a

"state transformer". Given two words $u, w \in \Sigma^*$, it is easy to see that $f_{uw} = f_w \circ f_u$, where $\circ$ denotes function composition.

Since $\mathcal{B}$ is the minimum DFA recognizing L, it follows that for any words $u, w \in \Sigma^*$, $u \equiv_L w$ if and only if $f_u = f_w$.

Clearly the function $f_w : S \to S$ corresponding to a word w has a bounded representation. So, if we could compute the function f_u corresponding to the input u, we would be able to determine whether $\delta(s_0, u) \in S_F$, since $\delta(s_0, u) = f_u(s_0)$.

However, we still have the original problem arising from the distributed nature of $\mathcal{A}$. Even if each process $p \in \mathcal{P}$ were to maintain the entire p-view of $\mathcal{E}_u$, the only information that we could reasonably hope to extract from the combined view of all the processes is some linearization of the trace $(\mathcal{E}_u, \leq_u, \lambda_u)$. From this labelled partial order, we cannot always recover u uniquely—in general, we can only reconstruct a word $u' \sim u$. Hence, we can at best hope to recover $f_{u'}$ for some $u' \sim u$.

Fortunately, this is not a bottleneck. From the definition of a trace language, it follows that all words that are $\sim$-equivalent are also $\equiv_L$-equivalent.

Proposition 9.27. *Let $\hat{L}$ be a trace language over a concurrent alphabet $(\Sigma, \mathcal{I})$. For any $u, u' \in \Sigma^*$, if $u \sim u'$ then $u \equiv_{\hat{L}} u'$.*

Proof. Suppose $u \sim u'$ but $u \not\equiv_{\hat{L}} u'$. Then, without loss of generality, we can find words w_1 and w_2 such that $w_1 u w_2 \in \hat{L}$ but $w_1 u' w_2 \notin \hat{L}$. Since $w_1 u w_2 \sim w_1 u' w_2$, this contradicts the assumption that $\hat{L}$ is $\sim$-consistent. $\square$

From our previous observation about $\mathcal{B}$, it follows that whenever $u' \sim u$, $u' \equiv_L u$, so $f_{u'} = f_u$. In other words, to determine whether $\delta(s_0, u) \in S_F$, it is sufficient to compute the function $f_{u'}$ corresponding to any word $u' \sim u$. Thus, we can write $f_{[u]}$ to denote the function associated with all linearizations of a trace u.

This, then, is our new goal: for any input word u, we want to compute in $\mathcal{A}$ the function $f_{[u]} : S \to S$ using some representative u' of the trace $[u]$. This still involves finding a scheme to combine the partial views of processes in a sensible way.

We begin by formally defining a projection of a word. A word $u \in \Sigma^*$ of length n can be viewed as a function $u : [1..n] \to \Sigma$ assigning a letter to each position in the word, where $[1..n]$ abbreviates the set $[1..n]$. In the trace $[u] = (\mathcal{E}_u, \leq_u, \lambda_u)$, we henceforth implicitly assume that $\mathcal{E}_u = \{e_{\perp}, e_1, e_2, \ldots, e_n\}$, where $e_{\perp}$ is the fictitious initial event that we have assumed for convenience and for $j \in [1..n]$, e_j is the event corresponding to the letter $u(j)$.

Projection of a word Let $u : [1..n] \to \Sigma$ whose corresponding trace is $[u] = (\mathcal{E}_u, \leq_u, \lambda_u)$, and let $X \subseteq \mathcal{E}_u$ where $X \setminus \{e_{\perp}\} = \{e_{i_1}, e_{i_2}, \ldots, e_{i_k}\}$, with $i_1 < i_2 < \cdots < i_k$. Then $u[X]$, *the projection* of u with respect to X, is the word $u(i_1)u(i_2) \cdots u(i_k)$. If $X \setminus \{e_{\perp}\} = \emptyset$ then $u[X] = \varepsilon$, the empty string.

Ideals revisited   So far, we have implicitly assumed that all ideals are non-empty. However, to construct the asynchronous automaton $\mathcal{A}$ it will be convenient to work with the empty ideal as well. So, henceforth, whenever we encounter an ideal I, unless we explicitly say that I is non-empty we do not rule out the possibility that $I = \emptyset$. Clearly, if $I = \emptyset$, the notions $\max_p(I)$, $primary_p(I)$ and $secondary_p(I)$ are not defined and we shall apply these operators only to non-empty ideals. We also adopt a convention regarding P-views of an ideal. Recall that for $P \subseteq \mathcal{P}$, the P-view $\partial_P(I)$ of a non-empty ideal I is the set of events $\bigcup_{p \in P} \downarrow\max_p(I)$. If $P = \emptyset$, we shall define $\partial_P(I) = \emptyset$.

We begin with the following fact, which is crucial in our construction of $\mathcal{A}$.

Lemma 9.28. *Let u be a word, $[u] = (\mathcal{E}_u, \leq_u, \lambda_u)$ the corresponding trace, and $I, J \subseteq \mathcal{E}_u$ be ideals such that $I \subseteq J$. Then $u[J] \sim u[I]u[J \setminus I]$.*

Proof Sketch: A basic result in trace theory is that $u \sim w$ if and only if $u\downarrow_{\{a,b\}} = w\downarrow_{\{a,b\}}$ for every pair of dependent letters $(a, b) \in \mathcal{D}$ [5].

Suppose $(a, b) \in \mathcal{D}$ and $u[J]\downarrow_{\{a,b\}} \neq (u[I]u[J \setminus I])\downarrow_{\{a,b\}}$. Then there must be an occurrence of a and an occurrence of b in $u[J]\downarrow_{\{a,b\}}$ which have been transposed in $(u[I]u[J \setminus I])\downarrow_{\{a,b\}}$. Assume, as usual, that the events in the trace $[u] = (\mathcal{E}_u, \leq_u, \lambda_u)$ are numbered $\{e_\perp, e_1, e_2, \ldots, e_n\}$ in correspondence with the positions of the letters in u. Let $e_a = e_i$ and $e_b = e_j$ be the events from $\mathcal{E}_J$ corresponding to these occurrences of a and b in u. Without loss of generality, we assume that $i < j$.

It must be the case that $e_a \notin I$ and $e_b \in I$, since the only rearrangement we have performed is to send letters not in $u[I]$ to the right. Since $(a, b) \notin \mathcal{I}$, we can find a process $p \in loc(a) \cap loc(b)$. But then $e_a \leq_p e_b$ and so $e_a \leq e_b$. Since I is an ideal, $e_b \in I$ and $e_a \in \downarrow e_b$, we must have $e_a \in I$ as well, which is a contradiction. □

Corollary 9.29. *Let u be a word, $[u] = (\mathcal{E}_u, \leq_u, \lambda_u)$ the corresponding trace, and $I_1 \subseteq I_2 \subseteq \cdots \subseteq I_k \subseteq \mathcal{E}_u$ a sequence of nested ideals. Then $u[I_k] \sim u[I_1]u[I_2 \setminus I_1]\cdots u[I_k \setminus I_{k-1}]$.*

Proof.   Applying Lemma 9.28 once, we get $u[I_k] \sim u[I_{k-1}]u[I_k \setminus I_{k-1}]$. We then apply the lemma to each of $u[I_{k-1}]$, $u[I_{k-2}]$, ..., $u[I_2]$ in turn to obtain the required expression. □

9.8.1.   *Process residues*

Let us return to our problem: We want to compute in $\mathcal{A}$, on any input u, the function $f_{[u]}$ via some $u' \sim u$. We order the processes in $\mathcal{P}$ so that $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$ and construct subsets $\{Q_j\}_{j \in [1..N]}$, where $Q_1 = \{p_1\}$ and for $j \in [2..N]$, $Q_j = Q_{j-1} \cup \{p_j\}$.

Let $[u] = (\mathcal{E}_u, \leq_u, \lambda_u)$. Construct ideals $I_0, I_1, \ldots, I_N \subseteq \mathcal{E}_u$ where $I_0 = \emptyset$ and for $j \in [1..N]$, $I_j = I_{j-1} \cup \partial_{p_j}(\mathcal{E}_u)$. Clearly $I_j = \partial_{Q_j}(\mathcal{E}_u)$ for $j \in [1..N]$.

Since $\mathcal{E}_u = \partial_{\mathcal{P}}(\mathcal{E}_u) = \partial_{Q_N}(\mathcal{E}_u) = I_N$ and $I_0 \subseteq I_1 \subseteq \cdots \subseteq I_N$, we can write down the following expression based on Corollary 9.29.

$$u = u[I_N] \sim u[I_0]u[I_1 \setminus I_0]\cdots u[I_N \setminus I_{N-1}]$$

For $j \in [2..N]$, $I_j \setminus I_{j-1} = \partial_{Q_j}(\mathcal{E}_u) \setminus \partial_{Q_{j-1}}(\mathcal{E}_u)$ is the same as $\partial_{p_j}(\mathcal{E}_u) \setminus \partial_{Q_{j-1}}(\mathcal{E}_u)$. So, we can rewrite our earlier expression in a more useful form as:

$$
\begin{aligned}
u &= u[\partial_{Q_N}(\mathcal{E}_u)] \\
&\sim u[\emptyset]u[\partial_{p_1}(\mathcal{E}_u) \setminus \emptyset]u[\partial_{p_2}(\mathcal{E}_u) \setminus \partial_{Q_1}(\mathcal{E}_u)]\cdots u[\partial_{p_N}(\mathcal{E}_u) \setminus \partial_{Q_{N-1}}(\mathcal{E}_u)] \quad (\Diamond)
\end{aligned}
$$

The word $u[\partial_{p_j}(\mathcal{E}_u) \setminus \partial_{Q_{j-1}}(\mathcal{E}_u)]$ is the portion of u that p_j has seen but which the processes in Q_{j-1} have not seen. This is a special case of what we call a residue.

Residues   Let $u \in \Sigma^*$ be a word whose associated trace is $[u] = (\mathcal{E}_u, \leq_u, \lambda_u)$, $I \subseteq \mathcal{E}_u$ an ideal and $p \in \mathcal{P}$ a process. $\mathcal{R}(u, p, I)$ denotes the word $u[\partial_p(\mathcal{E}_u) \setminus I]$ and is called the *residue u at p with respect to I*.

For ideals X and Y, recall that $X \setminus Y = X \setminus (X \cap Y)$, where $X \cap Y$ is also an ideal. So any residue $\mathcal{R}(u, p, I)$ can equivalently be written as $\mathcal{R}(u, p, \partial_p(\mathcal{E}_u) \cap I)$. We will often make use of this fact.

Since $u[\partial_{p_j}(\mathcal{E}_u) \setminus \partial_{Q_{j-1}}(\mathcal{E}_u)]$ can be rewritten as $\mathcal{R}(u, p_j, \partial_{Q_{j-1}}(\mathcal{E}_u))$, we can reformulate $(\Diamond)$ as follows:

$$
\begin{aligned}
u &= u[\partial_{Q_N}(\mathcal{E}_u)] \\
&\sim \mathcal{R}(u, p_1, \emptyset)\mathcal{R}(u, p_2, \partial_{Q_1}(\mathcal{E}_u))\mathcal{R}(u, p_3, \partial_{Q_2}(\mathcal{E}_u))\ldots \mathcal{R}(u, p_N, \partial_{Q_{N-1}}(\mathcal{E}_u))
\end{aligned}
$$

Let us give a special name to residues of this form.

Process residues $\mathcal{R}(u, p, I)$ is a *process residue* if $\mathcal{R}(u, p, I) = \mathcal{R}(u, p, \partial_P(\mathcal{E}_u))$ for some $P \subseteq \mathcal{P}$. We say that $\mathcal{R}(u, p, \partial_P(\mathcal{E}_u))$ is the *P-residue of u at p*.

Notice that $\mathcal{R}(u, p, \emptyset)$ is also a process residue, corresponding to the empty set of processes (by our convention that $\partial_\emptyset(\mathcal{E}_u) = \emptyset$.) Further, $\mathcal{R}(u, p, \emptyset) = u[\partial_p(\mathcal{E}_u)]$, the partial word corresponding to the p-view of $\mathcal{E}_u$.

Example 9.30. Consider our old example—the trace $[bacabba]$ depicted in Figure 9.3. Let $I = \{0, e_1, e_2, e_3\}$. $\partial_s(\mathcal{E}_u) \setminus I = \{e_5, e_6\}$, so $\mathcal{R}(u, s, I) = bb$. Moreover, $\mathcal{R}(u, s, I) = \mathcal{R}(u, s, \partial_p(\mathcal{E}_u))$, so it is the p-residue of u at s.

Suppose that along every input word u, each process p maintains all its P-residues $\mathcal{R}(u, p, \partial_P(\mathcal{E}_u))$, $P \subseteq \mathcal{P}$, as functions from S to S. As we remarked earlier, each of these functions can be represented in a finite, bounded manner. Since each process needs to keep track of only 2^N P-residues, where $N = |\mathcal{P}|$, all these functions can be incorporated into the local state of the process.

Going back to the expression ($\Diamond$), we can compute the function $f_{u[\partial_{Q_N}(\mathcal{E}_u)\setminus\emptyset]}$ corresponding to $u[\partial_{Q_N}(\mathcal{E}_u)\setminus\emptyset]$ by composing the functions corresponding to the residues $\mathcal{R}(u,p_1,\emptyset)$, $\mathcal{R}(u,p_2,\partial_{Q_1}(\mathcal{E}_u))$, ..., $\mathcal{R}(u,p_N,\partial_{Q_{N-1}}(\mathcal{E}_u))$. Notice that $u[\partial_{Q_N}(\mathcal{E}_u)\setminus\emptyset]=u$. So, we can then decide whether the state $\delta(s_0,u)$ belongs to S_F by applying the function $f_{u[\partial_{Q_N}(\mathcal{E}_u)]}$ to s_0.

Thus, our automaton $\mathcal{A}$ will accept u if $\delta(s_0,u)$ as computed using the process residues corresponding to the expression ($\Diamond$) lies in S_F. Recall that the accepting states of $\mathcal{A}$ are specified as global states. So, at the end of the word u, we are permitted to observe "externally", as it were, the states of *all* the processes in $\mathcal{A}$ before deciding whether to accept u.

The only hitch now is with computing process residues "on line", as $\mathcal{A}$ reads u. The problem is the following: Let $p\in\mathcal{P}$ and $P\subseteq\mathcal{P}$. If we extend u to ua where $p\notin loc(a)$, it could well happen that $\partial_p(\mathcal{E}_{ua})\setminus\partial_P(\mathcal{E}_{ua})\neq\partial_p(\mathcal{E}_u)\setminus\partial_P(\mathcal{E}_u)$, even though $\partial_p(\mathcal{E}_u)=\partial_p(\mathcal{E}_{ua})$.

Example 9.31. Consider the trace [*bacabba*] shown in Figure 9.3. After the subword *bac*, the p-residue at s is bc, corresponding to $\{e_1,e_3\}$. However, when this word is extended to *baca*, the p-residue at s becomes ε, though s does not participate in the final a.

9.8.2. *Primary residues*

Process residues at p can change without p being aware of it. This means that we cannot hope to directly maintain and update process residues locally as $\mathcal{A}$ reads u. To remedy this, we define a new type of residue called a primary residue.

Primary residues Let us call $\mathcal{R}(u,p,I)$ a *primary residue* if I is generated by a subset E of $primary_p(\mathcal{E}_u)$.

Clearly, for $p,q\in\mathcal{P}$, $\mathcal{R}(u,p,\partial_q(\mathcal{E}_u))$, can be rewritten as $\mathcal{R}(u,p,\partial_p(\mathcal{E}_u)\cap\partial_q(\mathcal{E}_u))$. So, by the previous result the q-residue $\mathcal{R}(u,p,\partial_q(\mathcal{E}_u))$ is a primary residue $\mathcal{R}(u,p,\downarrow E)$ for some $E\subseteq primary_p(\mathcal{E}_u)$. Further, p can effectively determine the set E given the primary information of both p and q. In fact, it will turn out that *all* process residues can be effectively described in terms of primary residues.

Example 9.32. In the word $u=bacabba$ shown in Figure 9.3, $\mathcal{R}(u,s,\partial_p(\mathcal{E}_u))$ corresponds to the primary residue $\mathcal{R}(u,s,\downarrow\{latest_{s\to q}(\mathcal{E}_u)\})$.

Our strategy will now be to maintain primary residues rather than process residues for each process p. The useful property we exploit is that the primary residues at p change *only* when p participates in an event.

Notice that this does not contradict our earlier observation that process residues at p can change independent of p. Even if a synchronization not involving p happens to modify the P-residue at p, the new P-residue remains a primary residue of p, albeit for a different subset of p's primary events.

Further, we show that when p participates in an event, it can recompute its primary residues using *just* the information it receives during the synchronization. At the end of the word u, the expression ($\Diamond$) written in terms of process residues that is used to compute $\delta(s_0,u)$ can be effectively rewritten in terms of primary residues. These residues will be available with each process in $\mathcal{P}$, thereby enabling us to calculate $\delta(s_0,u)$.

When discussing how to update primary information in the previous section, we have observed that the maximal events in the intersection of two local views $\partial_p(I)\cap\partial_q(I)$ of an ideal are always primary events for both p and q (Lemma 9.19). We begin with some consequences of this lemma.

Corollary 9.33. *Let $u\in\Sigma^*$ and $p\in\mathcal{P}$.*

(i) *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 $\mathcal{R}(u,p,I\cup J)=\mathcal{R}(u,p,\downarrow(E_I\cup E_J))$.*

(ii) *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 $primary_{\{p\}\cup Q}(\mathcal{E}_u)$.*

(iii) *Let $q,r\in\mathcal{P}$ such that $latest_{p\to r}(\mathcal{E}_u)\leq latest_{q\to r}(\mathcal{E}_u)$. Then $\mathcal{R}(u,p,\partial_r(\partial_q(\mathcal{E}_u)))$ 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 $primary_p(\mathcal{E}_u)$ and $secondary_q(\mathcal{E}_u)$.*

Proof.

(i) 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 $\mathcal{R}(u,p,I\cup J)=\mathcal{R}(u,p,\downarrow(E_I\cup E_J))$.

(ii) Let $Q=\{q_1,q_2,\ldots,q_k\}$. We can rewrite $\mathcal{R}(u,p,\partial_Q(\mathcal{E}_u))$ as $\mathcal{R}(u,p,\bigcup_{i\in[1..k]}\partial_{q_i}(\mathcal{E}_u))$. From Lemma 9.19 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 (i) of this Corollary, it then follows 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 $E=\bigcup_{i\in[1..k]}E_i$.

(iii) 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 Lemma 9.19, $\partial_p(J)\cap\partial_r(J)$ 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 $secondary_q(\mathcal{E}_u)$. $\qquad \square$

Part (ii) of the preceding Corollary makes explicit our claim that every process residue $\mathcal{R}(u, p, \partial_Q(\mathcal{E}_u))$, $Q \subseteq \mathcal{P}$, can be effectively rewritten as a primary residue $\mathcal{R}(u, p, \downarrow E)$, $E \subseteq primary_p(\mathcal{E}_u)$, based on the information available in $primary_{p \cup \{Q\}}(\mathcal{E}_u)$. In case $Q = \emptyset$, $\mathcal{R}(u, p, \partial_Q(\mathcal{E}_u))$ is given by the primary residue corresponding to $\emptyset \subseteq primary_p(\mathcal{E}_u)$.

9.8.3. *Computing primary residues locally*

We now describe how, while reading a word u, each process p maintains the functions f_w for each primary residue w of u at p.

Initially, at the empty word $u = \varepsilon$, every primary residue from $\{\mathcal{R}(u, p, \downarrow E)\}_{p \in \mathcal{P}, E \subseteq primary_p(\mathcal{E}_u)}$ is just the empty word ε. So, all primary residues are represented by the identity function $Id : S \to S$.

Let $u \in \Sigma^*$ and $a \in \Sigma$. Assume inductively that every $p \in \mathcal{P}$ has computed at the end of u the function f_w for each primary residue $w \in \{\mathcal{R}(u, p, \downarrow E)\}_{E \subseteq primary_p(\mathcal{E}_u)}$. We want to compute for each p the corresponding functions after the word ua, whose associated trace is $(\mathcal{E}_{ua}, \leq_{ua}, \lambda_{ua})$.

For processes not involved in a, these values do not change.

Proposition 9.34. *If $p \notin loc(a)$ then every subset $E \subseteq primary_p(\mathcal{E}_{ua})$ is also a subset of $primary_p(\mathcal{E}_u)$ and the primary residue $\mathcal{R}(ua, p, \downarrow E)$ is the same as the primary residue $\mathcal{R}(u, p, \downarrow E)$.*

Proof. This follows immediately from the fact that $\partial_p(\mathcal{E}_{ua}) = \partial_p(\mathcal{E}_u)$ and $primary_p(\mathcal{E}_{ua}) = primary_p(\mathcal{E}_u)$. $\qquad \square$

So, the interesting case is when p participates in a. We show how to calculate all the new primary residues for p using the information available with the processes in $loc(a)$ after u.

Lemma 9.35. *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 at u of the processes in $loc(a)$ using the information in $primary_{loc(a)}(\mathcal{E}_u)$ and $secondary_{loc(a)}(\mathcal{E}_u)$.*

Proof. Let e_a be the event corresponding to the new letter a—that is, $\mathcal{E}_{ua} \setminus \mathcal{E}_u = \{e_a\}$. There are two cases to consider.

Case 1: $(e_a \in E)$

Since $\downarrow E = \downarrow e_a = \partial_p(\mathcal{E}_{ua})$, the residue $\mathcal{R}(ua, p, \downarrow E) = \mathcal{R}(ua, p, \downarrow e_a)$ is the empty word ε. So the corresponding function is just the identity function $Id : S \to S$.

Case 2: $(e_a \notin E)$

We want to compute the function f_w corresponding to the word $w = ua[\partial_p(\mathcal{E}_{ua}) \setminus \downarrow E]$. By Lemma 9.28, we know that

$$ua[\partial_p(\mathcal{E}_{ua})] \sim ua[\downarrow E]ua[\partial_p(\mathcal{E}_{ua}) \setminus \downarrow E]. \tag{9.1}$$

But $ua[\partial_p(\mathcal{E}_{ua})] = u[\partial_{loc(a)}(\mathcal{E}_u)]a$ and so we have

$$ua[\partial_p(\mathcal{E}_{ua})] = u[\partial_{loc(a)}(\mathcal{E}_u)]a \sim u[\downarrow E]u[\partial_{loc(a)}(\mathcal{E}_u) \setminus \downarrow E]a. \tag{9.2}$$

Since $e_a \notin \downarrow E$, $ua[\downarrow E] = u[\downarrow E]$. Thus, cancelling $u[\downarrow E]$ from the right hand sides of (9.1) and (9.2) above, we have $u[\partial_{loc(a)}(\mathcal{E}_u) \setminus \downarrow E]a \sim ua[\partial_p(\mathcal{E}_{ua}) \setminus \downarrow E]$. So, to compute the function f_w, it suffices to compute the function corresponding to $u[\partial_{loc(a)}(\mathcal{E}_u) \setminus \downarrow E]a$.

Let $loc(a) = \{p_1, p_2, \ldots, p_k\}$, where $p = p_1$. Construct sets of processes $\{Q_i\}_{i \in [1..k]}$ such that $Q_1 = \{p_1\}$ and $Q_i = Q_{i-1} \cup \{q_i\}$ for $i \in [2..k]$.

Construct ideals $\{I_j\}_{j \in [0..k]}$ as follows: $I_0 = \downarrow E$ and for $j \in [1..k]$, $I_j = I_{j-1} \cup \mathcal{E}_u|_{p_j}$. Clearly, $I_0 \subseteq I_1 \subseteq \cdots \subseteq I_k \subseteq \mathcal{E}_u$.

By Corollary 9.29, $u[I_k] \sim u[I_0]u[I_1 \setminus I_0] \cdots u[I_k \setminus I_{k-1}]$. Since $u[I_k] = u[\partial_{loc(a)}(\mathcal{E}_u)]$ and $u[I_0] = u[\downarrow E]$, from (9.2) above it follows that the word $u[\partial_{loc(a)}(\mathcal{E}_u) \setminus \downarrow E]a$ which we seek is $\sim$-equivalent to the word $u[I_1 \setminus I_0] \cdots u[I_k \setminus I_{k-1}]a$.

Claim: For each $j \in [1..k]$, $u[I_j \setminus I_{j-1}]$ is a primary residue $\mathcal{R}(u, p_j, \downarrow F_j)$, where $F_j \subseteq primary_{p_j}(\mathcal{E}_u)$. Further, p_j can determine F_j from the information in $primary_{p_j}(\mathcal{E}_u)$ and $secondary_{loc(a)}(\mathcal{E}_u)$.

Assuming the claim, for each word $w_j = u[I_j \setminus I_{j-1}]$, we can find the corresponding function $f_{w_j} : S \to S$ among the primary residues stored by p_j after u. The composite function $f_a \circ f_{w_k} \circ f_{w_{k-1}} \circ \cdots \circ f_{w_1}$ then gives us the function corresponding to the word $u[I_1 \setminus I_0] \cdots u[I_k \setminus I_{k-1}]a$, which is what we need.

Proof of Claim: The way that primary events are updated guarantees that each event $e \in E$ was a primary event in $\mathcal{E}_u$, before a occurred, for one of the processes in $loc(a)$; i.e., $E \subseteq primary_{loc(a)}(\mathcal{E}_u)$. For $i \in [1..k]$, let $E_i = E \cap primary_{p_i}(\mathcal{E}_u)$.

First consider $u[I_1 \setminus I_0]$.

Let $E' = E \setminus E_1$. $I_1 \setminus I_0$ is the same as $\partial_{p_1}(\mathcal{E}_u) \setminus \downarrow(E_1 \cup E')$, which is the same as $\partial_{p_1}(\mathcal{E}_u) \setminus (\downarrow E_1 \cup \downarrow E')$. We want to compute $u[I_1 \setminus I_0] = \mathcal{R}(u, p_1, \downarrow E_1 \cup \downarrow E')$.

Each event $e \in E'$ is a primary event of the form $latest_{p_1 \to q_e}(\mathcal{E}_{ua})$ for some $q_e \in \mathcal{P}$. Further, for some $i \in [2..k]$, e was also the primary event $latest_{p_i \to q_e}(\mathcal{E}_u)$ before e_a occurred. Since p_1 has inherited this information from p_i, it must have been

the case that $latest_{p_1 \to q_e}(\mathcal{E}_u) \leq latest_{p_i \to q_e}(\mathcal{E}_u)$. So, by part (iii) of Corollary 9.33, the residue $\mathcal{R}(u, p_1, \downarrow e) = \mathcal{R}(u, p_1, \partial_{q_e}(\partial_{p_i}(\mathcal{E}_u)))$ corresponds to a primary residue $\mathcal{R}(u, p_1, \downarrow G_e)$, where p_1 can determine $G_e \subseteq primary_{p_1}(\mathcal{E}_u)$ from $primary_{p_1}(\mathcal{E}_u)$ and $secondary_{p_i}(\mathcal{E}_u)$.

So, by part (i) of Corollary 9.33, $\mathcal{R}(u, p_1, \downarrow E') = \mathcal{R}(u, p_1, \bigcup_{e \in E'} \downarrow e)$ is a primary residue $\mathcal{R}(u, p_1, \downarrow G_1)$ where $G_1 = \bigcup_{e \in E'} G_e$.

$\mathcal{R}(u, p_1, \downarrow E_1)$ is a primary residue since $E_1 \subseteq primary_p(\mathcal{E}_u)$. Applying part (i) of Corollary 9.33 again, $\mathcal{R}(u, p_1, (\downarrow E_1 \cup \downarrow E'))$ corresponds to the primary residue $\mathcal{R}(u, p_1, \downarrow F_1)$, where $F_1 = E_1 \cup G_1$.

Now consider $u[I_j \setminus I_{j-1}]$ for $j \in [2..k]$.

$I_j \setminus I_{j-1}$ is the same as $\partial_{p_j}(\mathcal{E}_u) \setminus (\downarrow E \cup \partial_{Q_{j-1}}(\mathcal{E}_u))$ so we want to compute the residue $\mathcal{R}(u, p_j, \downarrow E \cup \partial_{Q_{j-1}}(\mathcal{E}_u))$.

By a similar argument to the one for $u[I_1 \setminus I_0]$, p_j can compute a set $G_j \subseteq primary_{p_j}(\mathcal{E}_u)$ such that $\mathcal{R}(u, p_j, \downarrow E)$ corresponds to the primary residue $\mathcal{R}(u, p_j, \downarrow G_j)$.

By part (ii) of Corollary 9.33, p_j can compute from $primary_{Q_j}(\mathcal{E}_u)$ a set $H_j \subseteq primary_{p_j}(\mathcal{E}_u)$ such that $\mathcal{R}(u, p_j, \partial_{Q_{j-1}}(\mathcal{E}_u))$ corresponds to the primary residue $\mathcal{R}(u, q, \downarrow H_j)$.

We now use part (i) of Corollary 9.33 to establish that $\mathcal{R}(u, p_j, \downarrow E \cup \partial_{Q_{j-1}}(\mathcal{E})_u)$ corresponds to the primary residue $\mathcal{R}(u, p_j, \downarrow F_j)$, where $F_j = G_j \cup H_j$. □

9.8.4. *An asynchronous automaton for L*

Our analysis of process residues and primary residues immediately yields a deterministic asynchronous automaton $\mathcal{A}$ which accepts the language L. Recall that $\mathcal{B} = (S, \Sigma, \delta, s_0, S_F)$ is the minimal DFA recognizing L.

For $p \in \mathcal{P}$, each local state of p will consist of the following:

- Primary and secondary information for p, as stored by the gossip automaton.
- For each subset E of the primary events of p, a function $f_E : S \to S$ recording the (syntactic congruence class of the) primary residue $\mathcal{R}(u, p, E \downarrow)$ at the end of any word u.

At the initial state, for each process p, all the primary, secondary and tertiary information of p points to the initial event $e_\perp$. For each subset E of primary events, the function f_E is the identity function $Id : S \to S$.

The transition functions $\to_a$ modify the local states of $loc(a)$ as follows:

- Primary, secondary and tertiary information is updated as in the gossip automaton.
- The functions corresponding to primary residues are updated as described in the proof of Lemma 9.35.

Other than comparing primary information, the only operation used in updating the primary residues at p (Lemma 9.35) is function composition. This is easily achieved using the data available in the states of the processes which synchronized.

The final states of $\mathcal{A}$ are those where the value jointly computed from the primary residues in $\mathcal{P}$ yields a state in S_F. More precisely, order the processes as $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$. Construct subsets of processes $\{Q_i\}_{i \in [1..N]}$ such that $Q_1 = \{p_1\}$ and for $i \in [2..N]$, $Q_i = Q_{i-1} \cup \{p_i\}$.

Let $\vec{v} = \{v_1, v_2, \ldots, v_N\}$ be a global state of $\mathcal{A}$ such that $\mathcal{A}$ is in $\vec{v}$ after reading an input word u.

By Corollary 9.33 (ii), for each $i \in [2..N]$, we can compute from $\{v_1, v_2, \ldots, v_i\}$ a subset E_i of the primary information of p_i such that the Q_{i-1}-residue of p_i is also the primary residue of p_i with respect to E_i. Let f_i denote this primary residue. In addition, from the state v_1, we can extract the function f_1 corresponding to the primary residue $\mathcal{R}(u, p, \emptyset)$.

From the expression $(\diamondsuit)$, we know that the composite function $f_N \circ f_{N-1} \circ \cdots \circ f_1$ is exactly the function f_u associated with the input word u leading to the global state $\vec{v}$. So, we put $\vec{v}$ in the set of accepting states F of $\mathcal{A}$ iff $f_N \circ f_{N-1} \circ \cdots \circ f_1(s_0) \in S_F$.

Notice that it does not matter how we order the states in $\vec{v}$ when we try to decide whether $\vec{v} \in F$. We keep track of residues in all processes in a symmetric fashion, and the expression $(\diamondsuit)$ holds regardless of how we order $\mathcal{P}$. So, if $\vec{v}$ is a valid (i.e., reachable) global state, the composite function $f_N \circ f_{N-1} \circ \cdots \circ f_1$ which we compute from $\vec{v}$ is always the *same*, no matter how we order $\mathcal{P}$.

From our analysis of residues in this section, we have the following result.

Theorem 9.36. *The language accepted by $\mathcal{A}$ is exactly L.*

The size of $\mathcal{A}$

Proposition 9.37. *Let $M = |S|$ and $N = |\mathcal{P}|$, where S is the set of states of $\mathcal{B}$, the DFA recognizing L, and $\mathcal{P}$ is the set of processes in the corresponding asynchronous automaton $\mathcal{A}$ which we construct to accept L. Then, the number of local states of each process $p \in \mathcal{P}$ is at most $2^{O(2^N M \log M)}$.*

Proof. We estimate the number of bits required to store a local state of a process p.

From Lemma 9.25, we know that the primary and secondary information that we require to keep track of the latest gossip can be stored in $O(N^2 \log N)$ bits.

The new information we store in each local state of p is the collection of primary residues. Each residue, which is a function from S to S, can be written down as an array with M entries, each of $\log M$ bits; i.e., $M \log M$ bits in all. Each primary residue corresponds to a subset of primary events. There are N primary events and so, in general, we need to store 2^N residues. Thus, all the residues can be stored using $2^N M \log M$ bits.

So, the entire state can be written down using $O(2^N M \log M)$ bits, whence the number of distinct local states of p is bounded by $2^{O(2^N M \log M)}$. $\qquad\square$

9.9. Discussion

We have characterized the languages recognized by direct product automata as those that are shuffle-closed. On the other hand, we have shown that asynchronous automata accept precisely the class of recognizable trace languages. A very interesting and difficult open problem is to precisely characterize the class of languages recognized by synchronized products. A positive result has been obtained for a very restricted subclass in [6].

The bounded time-stamping algorithm is a fundamental building block for many important constructions involving asynchronous automata. As we have seen, it is at the heart of the proof of Zielonka's theorem. Time-stamping can also be used to define a subset construction for directly determinizing asynchronous automata [7]. Another application of bounded time-stamping is in providing an automata-theoretic decision procedure for TrPTL, a linear time temporal logic with local modalities that is interpreted over traces [8].

Zielonka's theorem was first described in [1]. Subsequently, an equivalent result in terms of an alternative distributed model called *asynchronous cellular automata* was presented in [3]. Our proof is taken from [9] and broadly follows the structure of the original proof in [1], except that the distributed time-stamping function that is an explicit intermediate step in our construction is implicit in the original proof of Zielonka. We believe that clearly separating and identifying the role played by time-stamping helps to make the proof more digestible. Recently, the residue based construction described here has been refined in [10] to show that of the exponentially many primary residues maintained by each process, only polynomially many are actually distinct. This observation eliminates one exponential in the overall complexity of the construction.

Zielonka's theorem can be seen as an algorithm to synthesize a distributed implementation from a sequential specification. Another way to present the same problem is to ask when a global state space can be decomposed into local state spaces such that the product of these local state spaces is isomorphic to the original global state space. This problem can been solved for direct products and asynchronous automata—see [11] for an overview of the results.

However, the following distributed synthesis problem is open: Decompose a global state space into a product of local state spaces that is *bisimilar* to the original state space. A bisimulation [12, 13] is a relation that holds between a pair of transition systems that can simulate each other very faithfully. This is potentially a more useful formulation of the problem since branching time properties are preserved by bisimulation.

References

[1] W. Zielonka: Notes on finite asynchronous automata, *R.A.I.R.O.—Inform. Théor. Appl.*, **21** (1987) 99–135.

[2] A. Mazurkiewicz: Basic notions of trace theory, in: J.W. de Bakker, W.-P. de Roever and G. Rozenberg (eds.), *Linear time, branching time and partial order in logics and models for concurrency, LNCS* **354** (1989) 285–363.

[3] R. Cori, Y. Metivier and W. Zielonka: Asynchronous mappings and asynchronous cellular automata, *Inform. and Comput.*, **106** (1993) 159–202.

[4] J. Hopcroft and J.D. Ullman: *Introduction to automata, languages and computation*, Addison-Wesley (1979).

[5] I.J. Aalbersberg and G. Rozenberg: Theory of traces, *Theoret. Comput. Sci.*, **60** (1988) 1–82.

[6] J. Berstel, L. Boasson and M. Latteux: Mixed languages. *Theoret. Comput. Sci.* **332**(1–3) (2005) 179–198.

[7] N. Klarlund, M. Mukund and M. Sohoni: Determinizing asynchronous automata, *Proc. ICALP 1994*, Springer LNCS **820** (1994) 130-141.

[8] P.S. Thiagarajan: TrPTL: A trace based extension of linear time temporal logic, *Proc. 9th IEEE LICS* (1994) 438–447.

[9] M. Mukund and M. Sohoni: Gossiping, asynchronous automata and Zielonka's theorem, *Report TCS-94-2*, Chennai Mathematical Institute, Chennai, India (1994).

[10] B Genest and A Muscholl: Constructing Exponential-Size Deterministic Zielonka Automata. *Proc. ICALP 2006*, Springer LNCS **4052** (2006) 565–576.

[11] M. Mukund: From global specifications to distributed implementations. in *Synthesis and Control of Discrete Event Systems*, B. Caillaud, P. Darondeau and L. Lavagno (eds), Kluwer (2002) 19–34.

[12] R. Milner: *Communication and Concurrency*, Prentice-Hall, London (1989).

[13] D. Park: Concurrency and Automata on Infinite Sequences. *Proc. 5th GI-Conference Karlsruhe*, *Theoretical Computer Science* **104** (1981) 167-183.

Index