Internal Report TCS-94-2
February, 1994

Gossiping, Asynchronous Automata and Zielonka's Theorem

Madhavan Mukund and Milind Sohoni

School of Mathematics
SPIC Science Foundation
92 G.N. Chetty Road, T. Nagar
Madras 600 017, INDIA
E-mail: {madhavan,sohoni}@ssf.ernet.in

Abstract

In this paper, we first tackle a natural problem from distributed computing, involving time-stamps. We then show that our solution to this problem can be applied to provide a simplified proof of Zielonka's theorem—a fundamental result in the theory of concurrent systems.

Let $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$ be a set of computing agents or processes which synchronize with each other from time to time and exchange information about themselves and others. The *gossip problem* is the following: Whenever a set $P \subseteq \mathcal{P}$ meets, the processes in P must decide *amongst themselves* which of them has the latest information, direct or indirect, about each agent p in the system.

We propose an algorithm to solve this problem which is finite-state and local. Formally, this means that our algorithm can be implemented as an asynchronous automaton.

Solving the gossip problem appears to be a basic step in tackling other problems involving asynchronous automata. Here, we apply our solution to derive an alternative proof of Zielonka's fundamental result that deterministic asynchronous automata accept precisely the class of recognizable trace languages. For a given recognizable trace language L over a concurrent alphabet $(\Sigma, \mathcal{I})$, we show how to construct a deterministic asynchronous automaton with the same underlying independence structure which accepts L.

Zielonka's original proof of this theorem is quite intricate and hard to grasp. To the best of our knowledge, ours is the first simplified proof of this result which works directly with asynchronous automata.

Introduction

This paper has two main aims. We first tackle a natural problem from distributed computing, involving time-stamps. We then show that our solution to this problem can be applied to provide a simplified proof of a fundamental result in the theory of concurrent systems—Zielonka's theorem that asynchronous automata accept precisely the class of recognizable trace languages [Zie1].

Let $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$ be a set of computing agents or processes which synchronize with each other from time to time and exchange information about themselves and others. The *gossip problem* is the following: Whenever a set $P \subseteq \mathcal{P}$ meets, the processes in P must decide *amongst themselves* which of them has the latest information, direct or indirect, about each agent p in the system.

This is easily accomplished if the agents decide to "time-stamp" every synchronization and pass these time-stamps along with each exchange of information. This does not require that all their clocks be synchronized. For example, each process can use an independent counter. When a set $P \subseteq \mathcal{P}$ meets, the processes in P jointly agree on a new value for their counters which exceeds the maximum of the counter values currently held by them. Thus, for any process p, the time-stamps assigned to synchronization events involving p form a strictly increasing sequence (albeit with gaps between successive time-stamps). So, the problem of deciding who has the latest information about p reduces to that of checking for the largest time-stamp.

This scheme has the following drawback: As the computation progresses these counter values increase without bound and most of the agents' time would be taken up in passing on large numbers, as opposed to actual gossip.

We propose an algorithm using counters which take on values from a bounded, finite set. We assign an independent counter to each subset of processes which can potentially synchronize. These counters are updated when the corresponding sets of processes meet. The update is performed jointly by the processes which meet.

Since our set of counter values is bounded, time-stamps have to be reused and, in general, different synchronizations involving a particular set of processes will acquire the same time-stamp during a computation. Despite this, our algorithm guarantees that whenever a set $P \subseteq \mathcal{P}$ meets, the processes in P can decide correctly which of them has the best information about any other agent p in the system. Thus, in essence, the processes in $\mathcal{P}$ may be finite state machines and yet manage to keep track of the latest information about other agents. Further, the algorithm itself does not induce any additional communications.[1]

We formalize the gossip problem and our solution to it in terms of asynchronous automata. These machines were first introduced by Zielonka and are a natural generalization of finite-state automata for modelling concurrent systems [Zie1]. An asynchronous automaton consists of a set of finite-state agents which synchronize to process their input. Each letter a in the input alphabet Σ is assigned a subset $\theta(a)$ of processes which jointly update their state when reading a. The processes outside $\theta(a)$ remain unchanged during this move—in fact, they are oblivious to the occurrence of a.

A *distributed alphabet* (Σ, θ) of this type gives rise to an *independence relation* $\mathcal{I}$

[1]An earlier proof of this result appeared in [MS]. However, the proof we provide here is new, more concise and more uniform with respect to different underlying patterns of synchronization.

between letters: $(a, b) \in \mathcal{I}$ iff a and b are processed by disjoint sets of components—i.e., $\theta(a) \cap \theta(b) = \emptyset$.

An alphabet with an independence relation is also called a *concurrent alphabet*. These were introduced in the theory of concurrent systems by Mazurkiewicz as a technique for studying such systems from the viewpoint of formal language theory [Maz]. Given a concurrent alphabet $(\Sigma, \mathcal{I})$, $\mathcal{I}$ induces a natural equivalence relation $\sim$ on Σ^*: two words u and u' are related by $\sim$ iff u' can be obtained from u by a sequence of permutations of adjacent independent letters. The equivalence class $[w]$ containing w is called a *trace*. Thus, the trace $[w]$ describes all possible ways of interleaving independent actions in w without affecting the overall computation.

A language $L \subseteq \Sigma^*$ is said to be a *trace language* over $(\Sigma, \mathcal{I})$ if L is closed under $\sim$. In other words, for each $w \in \Sigma^*$, if $w \in L$ then $[w] \subseteq L$. A trace language is *recognizable* if it is accepted by a conventional finite-state automaton over Σ.

However, since conventional automata are sequential, it is quite awkward to precisely characterize the class of automata which recognize trace languages. Asynchronous automata, on the other hand, are natural acceptors for these languages. If we distribute Σ in such a way that the induced independence relation is $\mathcal{I}$, we are guaranteed that the set of strings accepted by the automaton is recognizable and closed under $\sim$.

Despite this obvious connection, it is not easy to prove that the class of languages accepted by asynchronous automata coincides with the class of recognizable trace languages. This fundamental result was first established by Zielonka [Zie1]. Given a conventional finite automaton recognizing a trace language over $(\Sigma, \mathcal{I})$, he demonstrated how to construct directly a deterministic asynchronous automaton over a distributed alphabet (Σ, θ) which accepts the same language, such that the independence relation induced by θ is precisely $\mathcal{I}$.

Zielonka's original proof of this theorem is quite intricate and difficult to grasp. It turns out that our algorithm for keeping track of the latest information can be exploited to provide a simple, "operational" proof of this theorem. The states of the automaton we construct are highly structured. As a result, the transition function which updates the information in the states can be described in an "algorithmic" manner. This is in contrast to the automaton yielded by Zielonka's construction, where the set of states is just an enormous collection of names and the transition relations are specified by exhaustively listing out all the entries in the transition table. So, though both constructions yield state spaces of the same order of magnitude, the automaton we construct can be effectively presented much more concisely than the one Zielonka constructs.

Our proof is not the first attempt to simplify Zielonka's argument. Given that the statement of his theorem is at once both fundamental and elegant, both he and others have worked on alternative, more accessible proofs [CMZ, Die]. However, all of these new proofs have been based on a different machine model called asynchronous *cellular* automata. The drawback is that, unlike asynchronous automata, these new machines are not easy to visualize and do not correspond to any natural framework of distributed computation. We shall discuss the connections between our construction and these proofs based on asynchronous cellular automata in greater detail in the concluding section of the paper, where it will be easier to point out the similarities and differences between the two approaches. For the moment, we just note that the construction we give here is, to the best of our knowledge, the first simplified proof of Zielonka's theorem based directly on asynchronous automata.

The paper is organized as follows. In the next section, we introduce asynchronous automata and formalize the gossip problem in terms of these automata. To do this, we define a natural partial order on events in the system. In Section 2 we introduce ideals and frontiers, both of which play a crucial role in the rest of the paper. Sections 3 and 4 describe how to maintain, compare and update in a local manner the latest information about other processes. The next section puts all these ideas together and formally describes the "gossip automaton" which solves the first problem we set out to tackle.

We then move to recognizable trace languages and Zielonka's theorem. Section 6 provides the necessary background on traces. (We restrict ourselves to the notions we need. We ensure, however, that the presentation is self-contained.) In Section 7 we then introduce the notion of residues and use it to provide our proof of Zielonka's theorem. We compare our construction with Zielonka's original construction in Section 8.

In the concluding Discussion, we place our results in perspective. We discuss similarities and differences with other work on "gossiping" and bounded time-stamps [HHL, IL, DS, CS]. We also compare our proof with those based on asynchronous cellular automata [CMZ, Die]. Finally, we discuss other applications of the gossip automaton in logic and the theory of asynchronous automata [KMS, Thi].

1 Preliminaries

Let $\mathcal{P}$ be a finite set of processes which synchronize periodically and let the set of possible synchronizations permitted in the system be denoted $\mathcal{C}$, where $\mathcal{C} \subseteq (2^{\mathcal{P}} - \{\emptyset\})$. So, each element $c \in \mathcal{C}$ is a non-empty subset of $\mathcal{P}$. When c occurs, the processes in c share all information about their local states and update their states accordingly.

We model a computation of the system as a sequence of communications—that is, a word $u \in \mathcal{C}^*$. Let u be of length m. It is convenient to think of u as a function $u : [1..m] \to \mathcal{C}$, where for natural numbers i and j, $[i..j]$ abbreviates the set $\{i, i+1, \ldots, j\}$ if $i \leq j$ and $[i..j] = \emptyset$ otherwise. By this convention, the empty word ε is denoted by the unique function $\emptyset \to \mathcal{C}$.

Events With $u : [1..m] \to \mathcal{C}$, we associate a set of *events* $\mathcal{E}_u$. Each event e is of the form $(i, u(i))$, where $i \in [1..m]$. In addition, it is convenient to include an *initial event* denoted 0. Thus, $\mathcal{E}_u = \{0\} \cup \{(i, u(i)) \mid i \in [1..m]\}$.

The initial event marks an implicit synchronization of all the processes before the start of the actual computation. So, if u is the empty word ε, $\mathcal{E}_u = \{0\}$.

Usually, we will write $\mathcal{E}$ for $\mathcal{E}_u$. For $p \in \mathcal{P}$ and $e \in \mathcal{E}$, we write $p \in e$ to denote that $p \in u(i)$ when $e = (i, u(i))$; for the initial event 0, we define $p \in 0$ to hold for all $p \in \mathcal{P}$. If $p \in e$, then we say that e is a *p-event*.

Ordering relations on $\mathcal{E}$ The word u imposes a total order on events in $\mathcal{E}$: define $e < f$ if $e \neq f$ and either $e = 0$ or $e = (i, u(i))$, $f = (j, u(j))$, and $i < j$. We write $e \leq f$ if $e = f$ or $e < f$. Moreover, each process p orders the events in which it participates: define $\lessdot_p$ to be the strict ordering

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

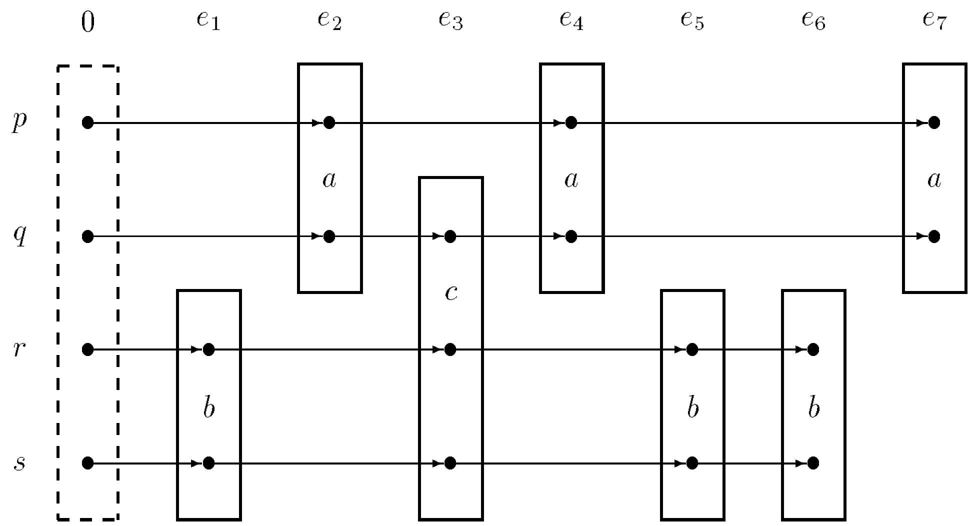

Figure 1: An example

The set of all p-events in $\mathcal{E}$ is totally ordered by $\lessdot_p^*$, the reflexive, transitive closure of $\lessdot_p$.

Define $e \sqsubset f$ if for some p, $e \lessdot_p f$ and $e \sqsubseteq f$ if $e = f$ or $e \sqsubset f$. Let $\sqsubseteq^*$ denote the transitive closure of $\sqsubseteq$. If $e \sqsubseteq^* f$ then we say that e is *below* f. The *causality relation* $\sqsubseteq^*$ is a partial order which models the cause and effect relationship between events in $\mathcal{E}$ more accurately than the temporal order $<$. Synchronizations between disjoint sets of processes can be performed independently—in particular, if two such synchronizations occur consecutively in u, they could also be transposed without affecting the outcome of the computation. Thus, many different rearrangements of the letters is u will, in general, give rise to isomorphic structures $(\mathcal{E}, \sqsubseteq^*)$.

EXAMPLE: *Let $\mathcal{P} = \{p, q, r, s\}$ and $\mathcal{C} = \{a, b, c\}$ where $a = \{p, q\}$, $b = \{r, s\}$ and $c = \{q, r, s\}$. Figure 1 shows the events $\mathcal{E}$ corresponding to the word* bacabba. *The dashed box corresponds to the "mythical" event 0, which we insert at the beginning for convenience.*

In the figure, the arrows between the events denote the relations $\lessdot_p$, $\lessdot_q$, $\lessdot_r$ and $\lessdot_s$. From these, we can compute $\sqsubset$ and $\sqsubseteq^*$. Thus, for example, we have $e_1 \sqsubseteq^* e_4$ since $e_1 \lessdot_r e_3 \lessdot_q e_4$.

Note that 0 is below every event. Also, for each $p \in \mathcal{P}$, the set of all p-events in $\mathcal{E}$ is totally ordered by $\sqsubseteq^*$ since $\lessdot_p^*$ is contained in $\sqsubseteq^*$.

The set of events below e is denoted $e\!\downarrow$. These represent the only synchronizations in $\mathcal{E}$ which are "known" to the processes in e when e occurs.

Latest information Let $\mathcal{E}$ be the set of events of the communication sequence $u : [1..m] \to \mathcal{C}$. The $\sqsubseteq^*$-maximum p-event in $\mathcal{E}$ is denoted $max_p(\mathcal{E})$. $max_p(\mathcal{E})$ is the last event in $\mathcal{E}$ in which p has taken part. Since $p \in 0 \in \mathcal{E}$ and all p-events are totally ordered by $\sqsubseteq^*$, $max_p(\mathcal{E})$ is well-defined.

Let $p, q \in \mathcal{P}$. The latest information p has about q in $\mathcal{E}$ corresponds to the $\sqsubseteq^*$-maximum q-event in the subset of events $max_p(\mathcal{E})\!\downarrow$. We denote this event by $latest_{p \to q}(\mathcal{E})$. Since all q-events are totally ordered by $\sqsubseteq^*$ and $q \in 0 \sqsubseteq^* max_p(\mathcal{E})$, $latest_{p \to q}(\mathcal{E})$ is well-defined.

EXAMPLE: *Continuing with our example, in Figure 1,* $max_p(\mathcal{E}) = e_7$ *whereas* $max_s(\mathcal{E}) = e_6$. $latest_{p\to q}(\mathcal{E}) = e_7$, *but* $latest_{p\to s}(\mathcal{E}) = e_3$. *On the other hand,* $latest_{s\to p}(\mathcal{E}) = e_2$.

For any processes $p, p', q \in \mathcal{P}$, the events $latest_{p\to q}(\mathcal{E})$ and $latest_{p'\to q}(\mathcal{E})$ are both q-events and are thus always comparable with respect to $\sqsubseteq^*$. Our first goal is to design a scheme whereby each process p maintains a bounded amount of information locally, so that whenever a set of processes $c \subseteq \mathcal{P}$ synchronizes they can decide amongst themselves which of them has heard most recently from every process in the system. More formally, for every $q \in \mathcal{P}$, all the processes in c should be able to jointly compute which of the events $\{latest_{p\to q}(\mathcal{E})\}_{p\in c}$ is maximum with respect to $\sqsubseteq^*$.

We make precise the notions of *bounded* and *local* information using asynchronous automata.

Asynchronous automata

Distributed alphabet Let $\mathcal{P}$ be a finite set of processes as before. A *distributed alphabet* is a pair (Σ, θ) where Σ is a finite set of *actions* and $\theta : \Sigma \to (2^{\mathcal{P}} - \{\emptyset\})$ assigns a non-empty set of processes to each $a \in \Sigma$.

State spaces With each process p, we associate a finite set of states denoted V_p. Each state in V_p is called a *local state*. For $P \subseteq \mathcal{P}$, we use V_P to denote the product $\prod_{p\in P} V_p$. An element $\vec{v}$ of V_P is called a P*-state*. A $\mathcal{P}$-state is also called a *global state*. Given $\vec{v} \in V_P$, and $P' \subseteq P$, we use $\vec{v}_{P'}$ to denote the projection of $\vec{v}$ onto $V_{P'}$.

Asynchronous automaton An *asynchronous automaton* $\mathfrak{A}$ over (Σ, θ) is of the form $(\{V_p\}_{p\in\mathcal{P}}, \{\to_a\}_{a\in\Sigma}, \mathcal{V}_0, \mathcal{V}_F)$, where $\to_a \subseteq V_{\theta(a)} \times V_{\theta(a)}$ is the *local transition relation* for a, and $\mathcal{V}_0, \mathcal{V}_F \subseteq V_{\mathcal{P}}$ are sets of *initial* and *final* global states. Intuitively, each local transition relation $\to_a$ specifies how the processes $\theta(a)$ that meet on a may decide on a joint move. Other processes do not change their state. Thus we define the *global transition relation* $\Rightarrow \subseteq V_{\mathcal{P}} \times \Sigma \times V_{\mathcal{P}}$ by $\vec{v} \stackrel{a}{\Longrightarrow} \vec{v}'$ if $\vec{v}_{\theta(a)} \to_a \vec{v}'_{\theta(a)}$ and $\vec{v}_{\mathcal{P}-\theta(a)} = \vec{v}'_{\mathcal{P}-\theta(a)}$.

$\mathfrak{A}$ is called *deterministic* if the global transition relation $\Longrightarrow$ of $\mathfrak{A}$ is a function from $V_{\mathcal{P}} \times \Sigma$ to $V_{\mathcal{P}}$ and the set of initial states $\mathcal{V}_0$ is a singleton. Notice that $\Longrightarrow$ is a function iff each local transition relation $\to_a$, $a \in \Sigma$, is a function from $V_{\theta(a)}$ to $V_{\theta(a)}$. All the asynchronous automata we deal with in this paper will be deterministic.

Runs Given a word $u : [1..m] \to \Sigma$, a *run* of $\mathfrak{A}$ on u is a function $\rho : [0..m] \to V_{\mathcal{P}}$ such that $\rho(0) \in \mathcal{V}_0$ and for $i \in [1..m]$, $\rho(i{-}1) \stackrel{u(i)}{\Longrightarrow} \rho(i)$. If $\mathfrak{A}$ is deterministic, each word u gives rise to a unique run which we denote ρ_u.

The word u is *accepted* by $\mathfrak{A}$ if there is a run ρ of $\mathfrak{A}$ on u such that $\rho(m) \in \mathcal{V}_F$. $L(\mathfrak{A})$, the language recognized by $\mathfrak{A}$, is the set of words accepted by $\mathfrak{A}$.

For the moment, we will not look at asynchronous automata as language recognizers. Instead, we want to treat them as devices for locally computing families of functions.

Locally computable functions Let *Val* be a set (of *values*). A Σ-indexed family of functions is a set $\mathcal{F}_\Sigma = \{f_a : \Sigma^* \to \mathit{Val}\}_{a\in\Sigma}$. So, $\mathcal{F}_\Sigma$ contains a function f_a for each letter $a \in \Sigma$.

$\mathcal{F}_\Sigma$ is *locally computable* if we can find a deterministic asynchronous automaton $\mathfrak{A} = (\{V_p\}_{p\in\mathcal{P}}, \{\rightarrow_a\}_{a\in\Sigma}, \mathcal{V}_0, \mathcal{V}_F)$ and a family of *local functions* $\mathcal{G}_\Sigma = \{g_a : V_{\theta(a)} \rightarrow \mathit{Val}\}_{a\in\Sigma}$, such that for each word $u : [1..m] \rightarrow \Sigma$, $f_a(u) = g_a(\vec{v}_{\theta(a)})$, where $\vec{v} = \rho_u(m)$ and ρ_u is the unique run of $\mathfrak{A}$ over u.

In other words, the processes in $\theta(a)$ can locally compute the value $f_a(u)$ for any $u \in \Sigma^*$ by applying the function g_a to the (unique) $\theta(a)$-state reached by the automaton after reading u.

Our problem involving the latest information of processes in $\mathcal{P}$ can now be formalized in terms of asynchronous automata.

Given $\mathcal{C} \subseteq (2^{\mathcal{P}} - \{\emptyset\})$, let $\Sigma = \{\hat{c}\}_{c\in\mathcal{C}}$. The distribution function θ is defined in the obvious way—for each $\hat{c} \in \Sigma$, $\theta(\hat{c}) = c$. For convenience, henceforth we shall drop the distinction between a subset $c \in \mathcal{C}$ and the corresponding letter $\hat{c} \in \Sigma$ and refer to both as just c. Thus, we will use V_c to denote the set of $\theta(\hat{c})$-states and $\rightarrow_c$ to denote the local transition function for $\hat{c}$.

Let $u : [1..m] \rightarrow \Sigma$ be a communication sequence and $c \subseteq \mathcal{P}$. For each $q \in \mathcal{P}$, we denote by $\mathit{best}_c(u,q)$ the set of processes in c which have the most recent information about q at the end of u—i.e.,

$$\mathit{best}_c(u,q) = \{p \in c \mid \forall p' \in c.\ \mathit{latest}_{p'\rightarrow q}(\mathcal{E}_u) \sqsubseteq^* \mathit{latest}_{p\rightarrow q}(\mathcal{E}_u)\}.$$

Let $\mathit{Val} = (2^{\mathcal{P}} - \{\emptyset\})^{\mathcal{P}}$. So, each member of Val is a function from $\mathcal{P}$ to non-empty subsets of $\mathcal{P}$. Our first goal is to show that the family of functions $\{\mathit{latest\text{-}gossip}_c : \Sigma^* \rightarrow \mathit{Val}\}_{c\in\Sigma}$ is locally computable, where:

$$\forall u \in \Sigma^*.\ \forall c \in \Sigma.\ \mathit{latest\text{-}gossip}_c(u) \textit{ is the function } \{p \mapsto \mathit{best}_c(u,p)\}_{p\in\mathcal{P}}.$$

2 Ideals and Frontiers

For the moment, let us fix a communication sequence $u : [1..m] \rightarrow \Sigma$ and the corresponding set of events $\mathcal{E}$.

The main source of difficulty in solving the gossip problem is the fact that the processes in $\mathcal{P}$ need to compute global information about the communication sequence u while each process only has access to a local, "partial" view of u. Although partial views of u 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 I is closed with respect to $\sqsubseteq^*$—i.e., $e \in I$ and $f \sqsubseteq^* e$ implies $f \in I$ as well [Sta]. We shall always refer to order ideals as just *ideals*.[2]

The requirement that an ideal be closed with respect to $\sqsubseteq^*$ 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.

[2]In the theory of partial orders, *order ideals* and *ideals* are distinct concepts. Ideals are normally assumed to be subsets which are $\sqsubseteq^*$-closed *and* directed. We shall, however, deal only with order ideals in this paper and so our terminology should cause no confusion.

Intuitively, the minimum possible partial view of a word u is the ideal $\{0\}$. This is because of our interpretation of 0 as an event which takes place *before* the actual computation begins. Hence, we shall assume that every ideal we consider is non-empty. Since 0 lies below every event in $\mathcal{E}$, $0 \in I$ for every non-empty ideal I.

Clearly the entire set $\mathcal{E}$ is an ideal, as is $e\!\downarrow$ for any $e \in \mathcal{E}$. Ideals of the form $e\!\downarrow$ are special ideals, called *principal ideals*—$e\!\downarrow$ is the principal ideal *generated by* e. It is not difficult to show that any ideal I is the union of the principal ideals generated by the set of events $E = \{e \mid e \textit{ is } \sqsubseteq^*\textit{-maximal in } I\}$. In general, if E is a set of events such that $I = \bigcup_{e \in E} e\!\downarrow$, we say that I is generated by E. It is easy to see that if I and J are ideals, so are $I \cup J$ and $I \cap J$.

Example: *Let us look once again at Figure 1.* $\{0, e_2\}$ *is an ideal, but* $\{0, e_2, e_3\}$ *is not, since* $e_1 \sqsubseteq^* e_3$ *but* $e_1 \notin \{0, e_2, e_3\}$. $\{0, e_1, e_2, e_3, e_5\}$ *is the principal ideal* $e_5\!\downarrow$*, whereas* $\{0, e_1, e_2, e_3, e_4, e_5\}$ *is a non-principal ideal generated by* $\{e_4, e_5\}$.

We need to generalize the notion of $max_p(\mathcal{E})$, the maximum p-event in $\mathcal{E}$, to all ideals $I \subseteq \mathcal{E}$.

P-views For an ideal I, the $\sqsubseteq^*$-maximum p-event in I is denoted $max_p(I)$. The *p-view of I* is the set $I|_p = max_p(I)\!\downarrow$. So, $I|_p$ is the set of all events in I which p can "see". For $P \subseteq \mathcal{P}$, the P-view of I, denoted $I|_P$, is $\bigcup_{p \in P} I|_p$, which is also an ideal. In particular, we have $I|_{\mathcal{P}} = I$.

Example: *In Figure 1, let I denote the ideal* $\{0, e_1, e_2, e_3, e_4, e_5, e_6\}$. $max_q(I) = e_4$ *and hence* $I|_q = \{0, e_1, e_2, e_3, e_4\}$. *On the other hand, though* $max_r(I) = e_6$, $I|_r \neq I$; $I|_r = I - \{e_4\}$. *The joint view* $I|_{\{q,r\}} = I = I|_{\mathcal{P}}$.

For an ideal I, the views $I|_p$ and $I|_q$ seen by two processes $p, q \in \mathcal{P}$ are, in general, incomparable. The events in I where these two views begin to diverge—the *frontier* of $I|_p \cap I|_q$—play a crucial role in our analysis.

Frontiers Let I be an ideal and $p, q, r \in \mathcal{P}$. We say that an event e is an *r-sentry* for p with respect to q if $e \in I|_p \cap I|_q$ and $e \lessdot_r f$ for some $f \in I|_q - I|_p$. Thus e is an event known to both p and q whose r-successor is known only to q. Notice that there need not always be an r-sentry for p with respect to q.

The pq-frontier at I, $\mathit{frontier}_{pq}(I)$ is defined as follows:

$$\mathit{frontier}_{pq}(I) = \{e \in I \mid \exists r \in \mathcal{P}.\ e \textit{ is an } r\textit{-sentry for } p \textit{ with respect to } q\}$$

Observe that this definition is asymmetric—in general, $\mathit{frontier}_{pq}(I) \neq \mathit{frontier}_{qp}(I)$.

Example: *As before, in Figure 1, let I denote the ideal* $\{0, e_1, e_2, e_3, e_4, e_5, e_6\}$. $I|_q \cap I|_r = \{0, e_1, e_2, e_3\}$. $\mathit{frontier}_{rq}(I) = \{e_2, e_3\}$*—$e_2$ is a p-sentry for r with respect to q whereas e_3 is a q-sentry. On the other hand,* $\mathit{frontier}_{qr}(I) = \{e_3\}$. *$e_3$ is both an r-sentry as well as an s-sentry for q with respect to r.*

As the example demonstrates, an event $e \in \mathit{frontier}_{pq}(I)$ could simultaneously be an r-sentry for p for several different processes r. However, it is not difficult to show that for any process r, there is at most one r-sentry for p with respect to q.

3 Primary and secondary information

For a word u and processes $p, q \in \mathcal{P}$, we have already defined $latest_{p\to q}(\mathcal{E})$, the latest information that p has about q after u. We now extend this definition to arbitrary ideals.

Primary information Let I be an ideal and $p, q \in \mathcal{P}$. Then $latest_{p\to q}(I)$ denotes the $\sqsubseteq^*$-maximum q-event in $I|_p$. So, $latest_{p\to q}(I)$ is the latest q-event in I that p knows about.

The *primary information* of p after I, $primary_p(I)$, is the set $\{latest_{p\to q}(I)\}_{q\in\mathcal{P}}$. 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). As usual, for $P \subseteq \mathcal{P}$, $primary_P(I) = \bigcup_{p\in P} primary_p(I)$.

As we have already remarked, for all $q \in \mathcal{P}$, the set of q-events in $I|_p$ is always nonempty, since $q \in 0 \in I|_p$. Further, since all q-events are totally ordered by $\lhd_q^*$ and hence by $\sqsubseteq^*$, the maximum q-event in $I|_p$ is well-defined. Notice that $latest_{p\to p}(I) = max_p(I)$.

To compare primary events, processes need to maintain additional information. It turns out that it is sufficient for each process to keep track of all the other processes' primary information.

Secondary information The *secondary information* of p after I, $secondary_p(I)$, is the (indexed) set $\bigcup_{q\in\mathcal{P}} primary_q(latest_{p\to q}(I)\downarrow)$. 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 $latest_{q\to r}(latest_{p\to q}(I)\downarrow)$ for some $q, r \in \mathcal{P}$. This is the latest r-event which q knows about upto the event $latest_{p\to q}(I)$. We abbreviate $latest_{q\to r}(latest_{p\to q}(I)\downarrow)$ 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).

However, we will 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. 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 $(p, q, e) \in primary_p(I)$—i.e., $e = latest_{p\to q}(I)$. Similarly, $e \in secondary_p(I)$ will indicate that for some $q, r \in \mathcal{P}$, $(p, q, r, e) \in secondary_p(I)$. We extend this to other set-theoretic operations as well. So, for instance, if we say $e \in primary_p(I) \cap secondary_q(I)$, we mean that we can find $p', q', q'' \in \mathcal{P}$ such that $(p, p', e) \in primary_p(I)$ and $(q, q', q'', e) \in secondary_q(I)$.

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 $primary_p(I) \subseteq secondary_p(I)$.

Comparing primary information

Our goal is to compare and update the primary information of processes whenever they meet. For this, we need the following observation regarding the significance of events lying on frontiers.

Lemma 1 *Let I be an ideal, $p, q \in \mathcal{P}$ and $e \in frontier_{pq}(I)$ an r-sentry for p with respect to q. Then $e = latest_{p\to r}(I)$. Also, for some $r' \in \mathcal{P}$, $e = latest_{q\to r'\to r}(I)$. So, $e \in primary_p(I) \cap secondary_q(I)$.*

Proof Since e is an r-sentry, for some $f \in I|_q - I|_p$, $e \lhd_r f$. Suppose that $latest_{p\to r}(I) = e' \neq e$. Since all r-events are totally ordered by $\lhd_r^*$, we must have $e \lhd_r^+ e'$ (where $\lhd_r^+$ is the transitive closure of the irreflexive relation $\lhd_r$). However, $e \lhd_r f$ as well, so we have $e \lhd_r f \lhd_r^* e'$. This means that $f \in I|_p$, which is a contradiction.

Next, we must show that $e = latest_{q\to r'\to r}(I)$ for some $r' \in \mathcal{P}$. We know that there is a path $e \sqsubset f_1 \sqsubset \cdots \sqsubset max_p(I)$, since $e \in I|_p$. This path starts inside $I|_p \cap I|_q$.

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

If this path does leave $I|_p \cap I|_q$, we can find an event e' along the path such that $e \sqsubseteq^* e' \lhd_{r'} f' \sqsubseteq^* max_p(I)$, where $e' \in I|_p \cap I|_q$, $f' \in I|_p - I|_q$ and $r' \in e' \cap f'$. In other words, e' is an r'-sentry for q with respect to p. We know by our earlier argument that $e' = latest_{q\to r'}(I)$. It must be the case that $e = latest_{r'\to r}(e'\downarrow)$. For, if $latest_{r'\to r}(e'\downarrow) = e'' \neq e$, then $e \lhd_r^+ e'' \sqsubseteq^* e' \sqsubseteq^* max_p(I)$. Since $e'' \in I|_p$ and $e \lhd_r^+ e''$, $e \neq latest_{p\to r}(I)$, which is a contradiction. So, $e = latest_{r'\to r}(e'\downarrow) = latest_{q\to r'\to r}(I)$ and we are done. □

Our observation about frontier events immediately gives us a way to compare primary information using both primary and secondary information.

Lemma 2 *Let I be an ideal and $p, q, r \in \mathcal{P}$. Let $e = latest_{p\to r}(I)$ and $f = latest_{q\to r}(I)$. Then $e \sqsubseteq^* f$ iff $e \in secondary_q(I)$.*

Proof

($\Leftarrow$) Suppose $e \in secondary_q(I)$. Then $r \in e \in I|_q$ and so $e \sqsubseteq^* f \in I|_q$ by the definition of $latest_{q\to r}(I)$.

($\Rightarrow$) If $e = f$, $e \in primary_q(I) \subseteq secondary_q(I)$, and there is nothing to prove. If $e \neq f$, then there exists an event e' such that $e \lhd_r e' \lhd_r^* f$ and so $e \in I|_p \cap I|_q$. We know that $e' \in I|_q - I|_p$, so e is an r-sentry in $frontier_{pq}(I)$. But then, by our previous lemma, $e \in primary_p(I) \cap secondary_q(I)$ and we are done. □

Suppose p and q synchronize at an action a after u. At this point they "share" their primary and secondary information. If q can find the event $latest_{p\to r}(\mathcal{E}_u)$ in its set of secondary events $secondary_q(\mathcal{E}_u)$, q knows that its latest r-event $latest_{q\to r}(\mathcal{E}_u)$ is at least as recent as $latest_{p\to r}(\mathcal{E}_u)$. So, after the synchronization, $latest_{q\to r}(\mathcal{E}_{ua})$ is the same as $latest_{q\to r}(\mathcal{E}_u)$, whereas p inherits this information from q—i.e., $latest_{p\to r}(\mathcal{E}_{ua}) = latest_{q\to r}(\mathcal{E}_u)$. In this way, for each $r \in \mathcal{P}$, p and q locally update their primary information about r in $\mathcal{E}_{ua}$. Clearly $latest_{p\to q}(\mathcal{E}_{ua}) = latest_{q\to p}(\mathcal{E}_{ua}) = e_a$, where e_a is the new event—i.e., $\mathcal{E}_{ua} - \mathcal{E}_u = \{e_a\}$.

This procedure generalizes to any arbitrary set $P \subseteq \mathcal{P}$ which synchronizes after u. The processes in P share their primary and secondary information and compare this information pairwise. Using Lemma 2, for each $q \in \mathcal{P} - P$ they decide who has the "latest information" about q. Each process then comes away with the best primary information from P.

Once we have compared primary information, updating secondary information is straightforward. Clearly, if $latest_{q \to r}(I)$ is better than $latest_{p \to r}(I)$, then every secondary event $latest_{q \to r \to r'}(I)$ must also be better than the corresponding event $latest_{p \to r \to r'}(I)$. So, secondary information can be locally updated too. In other words, to consistently update primary and secondary information, it suffices to correctly compare primary information, which is achieved by Lemma 2.

After a synchronization involving $P \subseteq \mathcal{P}$, notice that all processes in P will come away with the *same* set of primary and secondary events.

From the preceding argument, it is clear that the new event belongs to the primary (and hence secondary) information of the processes which synchronize at that event. Further, the update procedure reveals that if an event disappears from the secondary information of all the processes, it will never reappear as secondary information at some later stage. This is captured formally in the following proposition.

Proposition 3 *Let $u, w \in \Sigma^*$ such that $w = ua$ for some $a \in \Sigma$. Let e_a denote the new event in w—i.e., $\mathcal{E}_w - \mathcal{E}_u = \{e_a\}$. Then:*

- $e_a \in primary_{\mathcal{P}}(\mathcal{E}_w)$.
- $primary_{\mathcal{P}}(\mathcal{E}_w) \subseteq \{e_a\} \cup primary_{\mathcal{P}}(\mathcal{E}_u)$.
- $secondary_{\mathcal{P}}(\mathcal{E}_w) \subseteq \{e_a\} \cup secondary_{\mathcal{P}}(\mathcal{E}_u)$.

4 Locally updating primary/secondary information

To make Lemma 2 effective, we must make the assertions "locally checkable"—e.g., if $e = latest_{p \to r}(I)$, processes p and q must be able to decide if $e \in secondary_q(I)$.

Recall that e is represented in $primary_p(I)$ as a triple of the form (p, r, e). So, to check if $e \in secondary_q(I)$, q has to look for a quadruple of the form $(q, r', r'', e) \in secondary_q(I)$, where $r', r'' \in \mathcal{P}$. This can be checked locally provided events in $\mathcal{E}_u$ are labelled unambiguously while u is being read.

Clearly, labelling each event e as a pair $(i, u(i))$ is impossible since, in general, there is no agent which can consistently supply all processes with the "correct" value of i. Instead, we may naïvely assume that events in $\mathcal{E}_u$ are locally assigned distinct labels—in effect, at each action a, the processes in a together assign a (sequential) time-stamp to the new occurrence of a. In this manner, the processes in $\mathcal{P}$ can easily assign consistent local time-stamps for each action which will let them compute the relations $\lessdot_p^*$ between events.

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 which 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 the update of primary and secondary information relies on comparing labels, we must ensure that this reuse of labels does not lead to any confusion.

However, from Lemma 2, we know that to compare primary information, we only need to look at the events which are currently in the primary and secondary sets of each

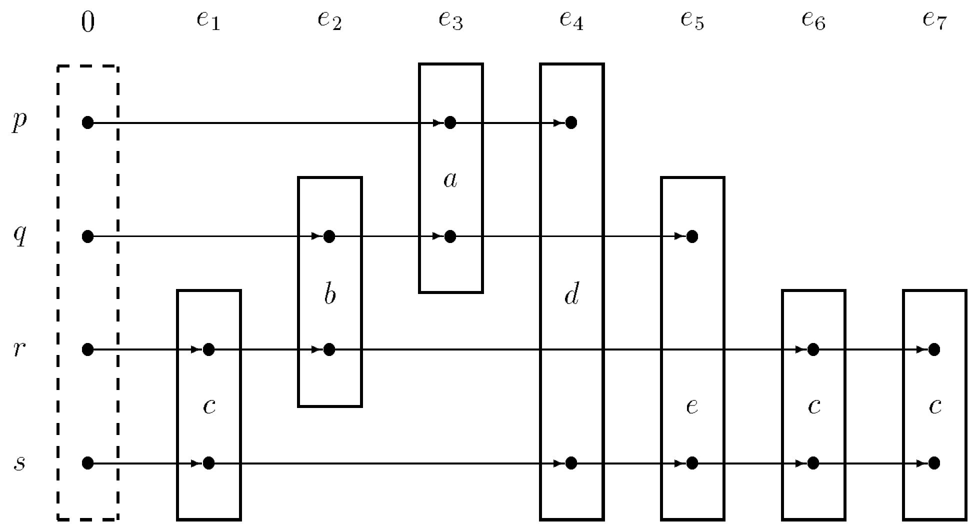

Figure 2: Another example

process. So, it is sufficient if the labels assigned to these sets are consistent across the system—i.e., if the same label appears in the current primary or secondary information of different processes, the corresponding event is actually the same.

Notice that we do not need to maintain a global temporal order on labels across the system. Lemma 2 assures us that to compare events of interest to us, it suffices to check for *equality* of labels assigned to the events.

Suppose we have such a labelling on u and we want to extend this to a consistent labelling on $w = ua$—i.e., we need to assign a label to the new a-event. By Proposition 3, it suffices to use a label which is distinct from the labels of all the a-events currently in the secondary information of $\mathcal{E}_u$. Since the cardinality of $secondary_{\mathcal{P}}(\mathcal{E}_u)$ 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 secondary information of the entire system. An a-event e may be part of the secondary information of processes *outside* a—i.e., $e \in secondary_{\mathcal{P}-a}(\mathcal{E}_u) - secondary_a(\mathcal{E}_u)$.

Example: *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 2 shows the events $\mathcal{E}$ corresponding to the word cbadecc.*

At the end of this word, $e_1 = latest_{p \to q \to s}(\mathcal{E})$. However, $e_1 \notin secondary_s(\mathcal{E})$;

$$\begin{aligned} secondary_s(\mathcal{E}) = \{ \; & (s,p,p,e_4),(s,p,q,e_3),(s,p,r,e_2),(s,p,s,e_4),\\ & (s,q,p,e_4),(s,q,q,e_5),(s,q,r,e_2),(s,q,s,e_5),\\ & (s,r,p,e_4),(s,r,q,e_5),(s,r,r,e_7),(s,r,s,e_7),\\ & (s,s,p,e_4),(s,s,q,e_5),(s,s,r,e_7),(s,s,s,e_7) \; \}. \end{aligned}$$

Since $max_r(\mathcal{E}) = max_s(\mathcal{E})$, $secondary_r(\mathcal{E}) = secondary_s(\mathcal{E})$ when viewed as sets of events. So, $e_1 \notin secondary_r(\mathcal{E})$ either. Thus, e_1 is a c-event which belongs to $secondary_{\mathcal{P}}(\mathcal{E}) - secondary_c(\mathcal{E})$.

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

Tertiary information The *tertiary information* of p after I, $tertiary_p(I)$, is the (indexed) set $\bigcup_{q \in \mathcal{P}} secondary_q(latest_{p \to q}(I){\downarrow})$. In other words, this is the latest information that p has in I about the secondary information of q, for all $q \in \mathcal{P}$. As before, for $P \subseteq \mathcal{P}$, $tertiary_P(I) = \bigcup_{p \in P} tertiary_p(I)$.

Each event in $tertiary_p(I)$ is of the form $latest_{q \to r \to s}(latest_{p \to q}(I){\downarrow})$ for some $q, r, s \in \mathcal{P}$. We abbreviate $latest_{q \to r \to s}(latest_{p \to q}(I){\downarrow})$ by $latest_{p \to q \to r \to s}(I)$. We represent each event $e = latest_{p \to q \to r \to s}(I)$ as a quintuple (p, q, r, s, e) in $tertiary_p(I)$. However, for convenience we will work with $tertiary_p(I)$ as though it were simply a set of events, rather than an indexed set, just as we have been doing with primary and secondary information.

Just as $primary_p(I) \subseteq secondary_p(I)$, clearly $secondary_p(I) \subseteq tertiary_p(I)$ since each secondary event $latest_{p \to q \to r}(I)$ is also a tertiary event $latest_{p \to p \to q \to r}(I)$ (or, equivalently, $latest_{p \to q \to q \to r}(I)$ and so on).

Lemma 4 *Let I be an ideal and $p \in \mathcal{P}$. If $e \in secondary_p(I)$ then for every $q \in e$, $e \in tertiary_q(I)$.*

Proof Let $e \in secondary_p(I)$ and $q \in e$. Concretely, let $e = latest_{p \to p' \to p''}(I)$ for some $p', p'' \in \mathcal{P}$. We know that $e \in I|_p \cap I|_q$ and there is a path $e \sqsubset f_1 \sqsubset \cdots \sqsubset max_p(I)$ leading from e to $max_p(I)$ which passes through $e' = latest_{p \to p'}(I)$.

Suppose this path never leaves $I|_p \cap I|_q$. Then $max_p(I) \in I|_q$ and so $max_p(I) = latest_{q \to p}(I)$. This means that $e \in secondary_p(latest_{q \to p}(I){\downarrow}) \subseteq tertiary_q(I)$ and we are done.

Otherwise, the path from e to $max_p(I)$ does leave $I|_p \cap I|_q$ at some stage.

If $e' \notin I|_p \cap I|_q$ then for some $f, f' \in \mathcal{E}$ and some $r \in \mathcal{P}$ we have $f \in I|_p \cap I|_q$, $f' \in I|_p - I|_q$ and $e \sqsubseteq^* f \lessdot_r f' \sqsubseteq^* e'$. This means that $f \in frontier_{qp}(I)$ is an r-sentry and by our earlier argument we know that $f = latest_{q \to r}(I)$. So $e = latest_{q \to r \to p''}(I) = latest_{q \to q \to r \to p''}(I) \in tertiary_q(I)$.

On the other hand, if $e' \in I|_p \cap I|_q$ we can find an r-sentry $f \in frontier_{qp}(I)$ on the path from e' to $max_p(I)$, for some $r \in \mathcal{P}$. We once again get $f = latest_{q \to r}(I)$ and so $e = latest_{q \to r \to p' \to p''}(I) \in tertiary_q(I)$.

$\square$

We shall use this lemma in the following form.

Corollary 5 *Let I be an ideal, $p \in \mathcal{P}$ and e a p-event in I. If $e \notin tertiary_p(I)$ then $e \notin primary_{\mathcal{P}}(I) \cup secondary_{\mathcal{P}}(I)$.*

So, a process p can keep track of which of its labels are "in use" in the system by maintaining tertiary information. Each p-event e initially belongs to $primary_e(I)$, and hence to $secondary_e(I)$ and $tertiary_e(I)$ as well. (Recall that for an event e, we also use e to denote the subset of $\mathcal{P}$ which meets at e). As the computation progresses, e gradually "recedes" into the background and disappears from the primary and secondary sets of the system. Eventually, when e disappears from $tertiary_p(I)$, p can be sure that e no longer belongs to $primary_{\mathcal{P}}(I) \cup secondary_{\mathcal{P}}(I)$.

Since $tertiary_p(I)$ 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.

5 The "gossip" automaton

Using our analysis of primary, secondary and tertiary information of processes, we can now design a deterministic asynchronous automaton to keep track of the "latest gossip"—i.e., to consistently update primary information whenever a set of processes synchronizes.

For $p \in \mathcal{P}$, each local state of p will consist of its primary, secondary and tertiary 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 0. The event 0 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 $tertiary_a(\mathcal{E}_u)$. Corollary 5 guarantees that this new label does not appear in $primary_{\mathcal{P}}(\mathcal{E}_u)$ or $secondary_{\mathcal{P}}(\mathcal{E}_u)$.

Proposition 3 guarantees that all new primary and secondary events—i.e., events in $primary_{\mathcal{P}}(\mathcal{E}_{ua})$ and $secondary_{\mathcal{P}}(\mathcal{E}_{ua})$—are assigned distinct labels.

Let $N = |\mathcal{P}|$. Since each process keeps track of N^3 tertiary events and at most N processes can synchronize at an event, there need be only N^4 labels in $\mathcal{L}$. (In fact, in Lemma 7 below, we show that it suffices to have N^3 labels in $\mathcal{L}$.)

(ii) The processes participating in e now compare their primary information about each process $q \notin e$ by checking labels of events across their primary and secondary sets as described in Lemma 2.

(iii) Each process then updates its primary, secondary and tertiary information according to the new information it receives. Tertiary information, like secondary information, can be locally updated once the processes have decided who has the best primary information—if $p, q \in a$ and $latest_{p \to r}(\mathcal{E}_u)$ is better than $latest_{q \to r}(\mathcal{E}_u)$ for $r \in \mathcal{P}$, then any tertiary information of the form $latest_{p \to r \to r' \to r''}(\mathcal{E}_u)$ must necessarily be better than the corresponding information $latest_{q \to r \to r' \to r''}(\mathcal{E}_u)$, for $r', r'' \in \mathcal{P}$.

This automaton does not have any final states, since we do not need to accept any language. Instead, we define for each $a \in \Sigma$ a function $g_a : V_a \to (2^{\mathcal{P}} - \{\emptyset\})^{\mathcal{P}}$ which checks the primary and secondary information of processes in a and computes for each $q \in \mathcal{P}$, the set of processes in a which have the most recent information about q.

(A small technical point: g_a must be defined for all states in V_a. However, not all combinations of local states may be "meaningful". We can easily assemble local states to form an a-state for which the inductive assertions do not hold, as a result of which our procedure for comparing primary information breaks down. However, since such a-states are unreachable, we can ignore this problem and simply assign a default value to g_a in these cases—for instance, the default value could be the function $\{p \mapsto a\}_{p \in \mathcal{P}}$.)

This immediately yields the first result we set out to establish.

Theorem 6 *Let (Σ, θ) be the distributed alphabet corresponding to $\mathcal{C} \subseteq (2^{\mathcal{P}} - \{\emptyset\})$. The family of functions $\{\mathit{latest\text{-}gossip}_c : \Sigma^* \to (2^{\mathcal{P}} - \{\emptyset\})^{\mathcal{P}}\}_{c \in \Sigma}$ is locally computable.*

So far, we have been working with a special distributed alphabet (Σ, θ), where θ is an injective function from Σ to $2^{\mathcal{P}} - \{\emptyset\}$. In general, we could have $a, b \in \Sigma$ such that $\theta(a) = \theta(b)$. However, in the construction of the gossip automaton, the "name" of the action attached to each event is irrelevant. For any word u, the structure of $(\mathcal{E}_u, \sqsubseteq^*)$ depends only on the sets of processes which synchronize at each event. So, if $\theta(a) = \theta(b)$, we have $\mathit{latest\text{-}gossip}_a(u) = \mathit{latest\text{-}gossip}_b(u)$ for all words u and, correspondingly, the local functions g_a and g_b are the same.

So, to construct the gossip automaton for a distributed alphabet (Σ, θ) where θ is not injective, we can ignore the distinction between letters which have the same distribution. In other words, we set the communication alphabet $\mathcal{C} \subseteq (2^{\mathcal{P}} - \{\emptyset\})$ to the range of θ and proceed as we have done here.

The size of the gossip automaton

Lemma 7 *In the gossip automaton, the number of local states of each process $p \in \mathcal{P}$ is at most $2^{O(N^3 \log N)}$, where $N = |\mathcal{P}|$.*

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

Recall that for any ideal I, each event in $\mathit{primary}_p(I)$ is also present in $\mathit{secondary}_p(I)$. Similarly, each event in $\mathit{secondary}_p(I)$ is also present in $\mathit{tertiary}_p(I)$. So it suffices to store just the labels of tertiary events. These events are stored in an array with N^3 entries, where each entry is implicitly indexed by a triple from $\mathcal{P} \times \mathcal{P} \times \mathcal{P}$.

Each new event e was 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^4 labels in $\mathcal{L}$. Actually, we can make do with N^3 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 and tertiary information about processes from $\mathcal{P} - a$. 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, secondary and tertiary information. At this stage, there are (less than) N^3 distinct labels present in $\mathit{tertiary}_a(I)$. So, if $|\mathcal{L}| = N^3$ 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^3 such pairs in the array of tertiary events, so the whole state can be described using $O(N^3 \log N)$ bits. Therefore, the number of distinct local states of p is bounded by $2^{O(N^3 \log N)}$. $\square$

6 Recognizable trace languages

We now apply the gossip automaton to establish the second major result of this paper—an "algorithmic" proof of Zielonka's theorem that asynchronous automata accept precisely the class of recognizable trace languages. We begin by providing a brief introduction to recognizable trace languages.

Concurrent alphabet A distributed alphabet (Σ, θ) gives rise to a natural *independence relation* $\mathcal{I}_\theta$ between letters: $(a, b) \in \mathcal{I}_\theta$ iff $\theta(a) \cap \theta(b) = \emptyset$. Thus, a and b are independent when processed by disjoint sets of agents in the system. Clearly, the relation $\mathcal{I}_\theta$ is irreflexive and symmetric. Such a relation is called an independence relation.

An alphabet equipped with an independence relation is also called a *concurrent alphabet*. This notion was introduced by Mazurkiewicz as a technique for studying concurrent systems from the viewpoint of formal language theory [Maz].

Traces and trace languages Given a concurrent alphabet $(\Sigma, \mathcal{I})$, $\mathcal{I}$ induces a natural equivalence relation $\sim$ 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

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

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

The quotient of Σ^* by the congruence $\sim$ is the free partially commutative monoid induced by $\mathcal{I}$. We denote this quotient monoid $\mathcal{M}(\Sigma, \mathcal{I})$. The elements of $\mathcal{M}(\Sigma, \mathcal{I})$ correspond to equivalence classes of words of Σ^* under $\sim$ and are called *traces*. Let $[w]$ denote the $\sim$-equivalence class corresponding to the word w. Since the relation $\sim$ is a congruence, the composition operation in $\mathcal{M}(\Sigma, \mathcal{I})$ is given by

$$\forall u, v \in \Sigma^*.\ [u][v] = [uv].$$

Subsets of the trace monoid $\mathcal{M}(\Sigma, \mathcal{I})$ are called *trace languages*. To define recognizable trace languages we need to define monoid automata.

Monoid automaton Let $(M, \cdot, \varepsilon)$ be a monoid, where ε is the unit. A monoid automaton over M is a structure $\mathfrak{M} = (S, M, \delta, s_0, S_F)$ where S is a finite set of states, $s_0 \in S$ is the initial state, $S_F \subseteq S$ is the set of final states and $\delta : S \times M \to S$ is the transition function such that:

- $\forall s \in S.\ \delta(s, \varepsilon) = s$.
- $\forall s \in S.\ \forall m_1, m_2 \in M.\ \delta(s, m_1 \cdot m_2) = \delta(\delta(s, m_1), m_2)$.

$\mathfrak{M}$ is said to *recognize* the subset $T = \{m \in M \mid \delta(s_0, m) \in S_F\}$.

Recognizable trace languages So, a trace language T is *recognizable* iff we can find a monoid automaton over the trace monoid $\mathcal{M}(\Sigma, \mathcal{I})$ which recognizes T. (More formally, recognizability is defined in terms of homomorphisms into a finite monoid, but for our purposes, this definition suffices.)

We prefer to treat trace languages as string languages which satisfy a closure condition rather than as subsets of the trace monoid $\mathcal{M}(\Sigma, \mathcal{I})$. 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 $\mathcal{M}(\Sigma, \mathcal{I})$ and $\sim$-consistent languages over Σ^*.

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

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 $\mathcal{M}(\Sigma, \mathcal{I})$.

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

We begin by building the dependence graph for $(\Sigma, \mathcal{I})$. Let $\mathcal{D} = (\Sigma \times \Sigma) - \mathcal{I}$. $\mathcal{D}$ is called the *dependence relation*. Construct a graph $G_{\mathcal{D}}$ whose vertices are labelled by Σ. Draw an edge between vertices labelled a and b provided $(a, b) \in \mathcal{D}$.

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

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

It is not difficult to see that any language accepted by an asynchronous automaton over (Σ, θ) is a recognizable trace language over the corresponding concurrent alphabet $(\Sigma, \mathcal{I}_\theta)$. The distributed nature of the automaton guarantees that it is a trace language. To see that the language is recognizable, we note that for a given asynchronous automaton $\mathfrak{A} = (\{V_p\}_{p \in \mathcal{P}}, \{\rightarrow_a\}_{a \in \Sigma}, \mathcal{V}_0, \mathcal{V}_F)$, we can construct a conventional finite state automaton $\mathfrak{B}$ accepting the same language as $\mathfrak{A}$. The states of $\mathfrak{B}$ are the global states of $\mathfrak{A}$ and the transition relation of $\mathfrak{B}$ is given by the global transition relation $\Longrightarrow$ of $\mathfrak{A}$. Since the initial and accepting states of $\mathfrak{A}$ are specified as global states, they can directly serve as the initial and final states of $\mathfrak{B}$. It is straightforward to verify that $\mathfrak{B}$ accepts the same language as $\mathfrak{A}$.

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 $\mathfrak{A}$ over a distributed alphabet (Σ, θ) such that $\mathfrak{A}$ accepts L *and* the independence relation $\mathcal{I}_\theta$ induced by θ is *exactly* $\mathcal{I}$?

Zielonka's fundamental result is that this is indeed the case [Ziel]. 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.

However, Zielonka's original proof of this theorem is extremely intricate and difficult to follow. Here we provide a "algorithmic" proof of the theorem which crucially uses the local computability of the latest gossip function. We first present our proof in detail and then discuss the connections with Zielonka's original proof.

7 Residues and Zielonka's theorem

Fix a recognizable trace language L over a concurrent alphabet $(\Sigma, \mathcal{I})$, as well as a distribution $\theta : \Sigma \to (2^{\mathcal{P}} - \{\emptyset\})$ such that the induced independence relation $\mathcal{I}_\theta$ is the same as $\mathcal{I}$. We shall construct a deterministic asynchronous automaton $\mathfrak{A} = (\{V_p\}_{p\in\mathcal{P}}, \{\to_a\}_{a\in\Sigma}, \mathcal{V}_0, \mathcal{V}_F)$ over (Σ, θ) recognizing L.

Let $\mathfrak{B}$ be the finite state automaton accepting L. Without loss of generality, we can assume that $\mathfrak{B}$ is the minimal deterministic finite automaton (DFA) for L, as given by the Myhill-Nerode theorem [HU]. Let $\mathfrak{B}$ be of the form $(S, \Sigma, \delta, s_0, S_F)$ where S denotes the set of states of $\mathfrak{B}$, $\delta : S \times \Sigma \to S$ 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 our asynchronous automaton $\mathfrak{A}$ from the original DFA $\mathfrak{B}$ is the following: On reading an input word u, we must be able to compute whether $\delta(s_0, u) \in S_F$. Unfortunately, after reading u each process in $\mathfrak{A}$ only has partial information about $\delta(s_0, u)$—a process p only "knows about" those events from $\mathcal{E}_u$ which lie in the p-view $max_p(\mathcal{E}_u)\!\downarrow$. So, 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 minimal automaton $\mathfrak{B}$.

We first recall a basic fact about recognizable languages. Any language $\hat{L}$ defines a syntactic congruence $\equiv_{\hat{L}}$ on Σ^* as follows:

$$\textit{For } u, u' \in \Sigma^*,\ u \equiv_{\hat{L}} u' \textit{ provided for all } w_1, w_2 \in \Sigma^*,\ w_1 u w_2 \in \hat{L} \textit{ iff } w_1 u' w_2 \in \hat{L}.$$

By the Myhill-Nerode theorem, if $\hat{L}$ is recognizable, $\equiv_{\hat{L}}$ is of finite index [HU].

Now, consider the relation $\equiv_L$ defined by 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 $\mathfrak{B}$, such that $f_u(s) = s'$ iff $\delta(s, u) = s'$. So, f_u is a representation of the word u as a "state transformer". The following observations follow from the fact that $\mathfrak{B}$ is the minimal DFA recognizing L.

Proposition 8 *Let $u, w \in \Sigma^*$. Then:*

(i) $f_u = f_w$ *iff* $u \equiv_L w$.

(ii) $\delta(s_0, u) = f_u(s_0)$.

(iii) $f_{uw} = f_w \circ f_u$, *where* $\circ$ *denotes function composition.*

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 $\delta(s_0, u) \in S_F$—by part (ii) of the preceding proposition, $\delta(s_0, u) = f_u(s_0)$.

However, we still have the original problem arising from the distributed nature of $\mathfrak{A}$—each process $p \in \mathcal{P}$ will only see a part of u. Even if 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 the structure $(\mathcal{E}_u, \sqsubseteq^*)$. From this labelled partial order, we cannot always recover u uniquely—in general, we can only reconstruct a word u' which is $\sim$-equivalent to u.

For the same reason, if we attempt to piece together f_u from the information available with each process about u, 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 *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. $\Box$

So, to determine whether $\delta(s_0, u) \in S_F$, it is sufficient to compute the function $f_{u'}$ corresponding to some word $u' \sim u$. By Propositions 8 and 9, $\delta(s_0, u) = f_{u'}(s_0)$.

This, then, is our new goal: for any input word u, we want to compute in $\mathfrak{A}$ the function $f_{u'} : S \to S$ for 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 partial view of a word. Let $u : [1..n] \to \Sigma$ and $X \subseteq \mathcal{E}_u$ where $X - \{0\} = \{(i_1, u(i_1)), (i_2, u(i_2)), \ldots, (i_k, u(i_k))\}$ and $i_1 < i_2 < \cdots < i_k$. Let $u[X]$ denote the word $u(i_1)u(i_2)\cdots u(i_k)$. If $X - \{0\} = \emptyset$ 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 $\mathfrak{A}$ 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 $max_p(I)$, $primary_p(I)$, $secondary_p(I)$ and $tertiary_p(I)$ 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 $I|_P$ of a non-empty ideal I is the set of events $\bigcup_{p\in P} max_p(I){\downarrow}$. If $P = \emptyset$, we shall define $I|_P = \emptyset$.

We begin with a basic result about the equivalence relation $\sim$, which we state without proof (see, for instance, [AR]). We need some notation: For $w \in \Sigma^*$ and $A \subseteq \Sigma$, let $w{\downarrow_A}$ denote the word formed by deleting all letters from w except those in A.

Proposition 10 *Let $w, w' \in \Sigma^*$. Then $w \sim w'$ iff for each pair of dependent letters $(a, b) \in \mathcal{D}$, $w{\downarrow_{\{a,b\}}}= w'{\downarrow_{\{a,b\}}}$.*

We apply this result to derive the following fact, which is crucial in our construction of $\mathfrak{A}$.

Lemma 11 *Let u be a word and $I, J \subseteq \mathcal{E}_u$ be ideals such that $I \subseteq J$. Then $u[J] \sim u[I]u[J - I]$.*

Proof We shall show that $u[J]{\downarrow_{\{a,b\}}}= (u[I]u[J - I]){\downarrow_{\{a,b\}}}$ for each pair of dependent letters $(a, b) \in \mathcal{D}$. The result then follows from Proposition 10.

Suppose $(a, b) \in \mathcal{D}$ and $u[J]{\downarrow_{\{a,b\}}}\neq (u[I]u[J - I]){\downarrow_{\{a,b\}}}$. Then there must be an occurrence of a and an occurrence of b in $u[J]{\downarrow_{\{a,b\}}}$ which have been transposed in $(u[I]u[J - I]){\downarrow_{\{a,b\}}}$. Let $e_a = (i, a)$ and $e_b = (j, b)$ be the events from 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 $p \in \theta(a) \cap \theta(b)$. But then $e_a \lessdot_p^* e_b$ and so $e_a \sqsubseteq^* e_b$. Since I is an ideal, $e_b \in I$ and $e_a \in e_b{\downarrow}$, we must have $e_a \in I$ as well, which is a contradiction. $\square$

Corollary 12 *Let u be a word and $I_1 \subseteq I_2 \subseteq \cdots \subseteq I_k \subseteq \mathcal{E}_u$ a sequence of nested ideals. Then $u[I_k] \sim u[I_1]u[I_2 - I_1] \cdots u[I_k - I_{k-1}]$.*

Proof Applying Lemma 11 once, we get $u[I_k] \sim u[I_{k-1}]u[I_k - I_{k-1}]$. 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. $\square$

Let us return to our problem: We want to compute in $\mathfrak{A}$, on any input u, the function $f_{u'}$ corresponding to 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 $\mathcal{E}$ be the set of events corresponding to u. Construct ideals $I_0, I_1, \ldots, I_N \subseteq \mathcal{E}$ where $I_0 = \emptyset$ and for $j \in [1..N]$, $I_j = I_{j-1} \cup \mathcal{E}|_{p_j}$. Clearly $I_j = \mathcal{E}|_{Q_j}$ for $j \in [1..N]$.

Since $\mathcal{E} = \mathcal{E}|_{\mathcal{P}} = \mathcal{E}|_{Q_N} = I_N$ and $I_0 \subseteq I_1 \subseteq \cdots \subseteq I_N$, we can write down the following expression based on Corollary 12.

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

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

$$u = u[\mathcal{E}|_{Q_N}] \sim u[\emptyset]u[\mathcal{E}|_{p_1} - \emptyset]u[\mathcal{E}|_{p_2} - \mathcal{E}|_{Q_1}] \cdots u[\mathcal{E}|_{p_N} - \mathcal{E}|_{Q_{N-1}}] \qquad (\Diamond)$$

The word $u[\mathcal{E}|_{p_j} - \mathcal{E}|_{Q_{j-1}}]$ 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, $I \subseteq \mathcal{E}_u$ an ideal and $p \in \mathcal{P}$ a process. $\mathcal{R}(u, p, I)$ denotes the word $u[\mathcal{E}_u|_p - I]$ and is called the *residue* of u at p with respect to I.

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

Using our new notation, we can write $u[\mathcal{E}|_{p_j} - \mathcal{E}|_{Q_{j-1}}]$ as $\mathcal{R}(u, p_j, \mathcal{E}|_{Q_{j-1}})$. Let us give a special name to residues of this form: $\mathcal{R}(u, p, I)$ is a *process residue* if $\mathcal{R}(u, p, I) = \mathcal{R}(u, p, \mathcal{E}|_P)$ for some $P \subseteq \mathcal{P}$. We say that $\mathcal{R}(u, p, \mathcal{E}|_P)$ 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 $\mathcal{E}|_\emptyset = \emptyset$.) Further, $\mathcal{R}(u, p, \emptyset) = u[\mathcal{E}|_p]$, the partial word corresponding to the p-view of $\mathcal{E}$.

Example: *Consider our old example—the word bacabba depicted in Figure 1. Let $I = \{0, e_1, e_2, e_3\}$. $\mathcal{E}|_s - I = \{e_5, e_6\}$, so $\mathcal{R}(u, s, I) = bb$. Moreover, $\mathcal{R}(u, s, I) = \mathcal{R}(u, s, \mathcal{E}|_p)$, 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 $\mathcal{R}(u, p, \mathcal{E}|_P)$, $P \subseteq \mathcal{P}$, 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 $(\diamondsuit)$, we can compute the function $f_{u[\mathcal{E}|_{p_N} - \emptyset]}$ corresponding to $u[\mathcal{E}|_{p_N} - \emptyset]$ by composing the functions corresponding to the residues $\mathcal{R}(u, p_1, \emptyset)$, $\mathcal{R}(u, p_2, \mathcal{E}|_{Q_1}), \ldots, \mathcal{R}(u, p_N, \mathcal{E}|_{Q_{N-1}})$ (Proposition 8 (iii)). Notice that $u[\mathcal{E}|_{p_N} - \emptyset] = u$. So, by Proposition 8 (ii), we can then compute the state $\delta(s_0, u)$ by applying the function $f_{u[\mathcal{E}|_{p_N} - \mathcal{E}|_{p_1}]}$ to s_0.

Thus, our automaton $\mathfrak{A}$ will accept u if $\delta(s_0, u)$ as computed using the process residues corresponding to the expression $(\diamondsuit)$ lies in S_F.

(Recall that the accepting states of $\mathfrak{A}$ 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 $\mathfrak{A}$ before deciding whether to accept u.)

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

Example: *Consider the word bacabba shown in Figure 1. 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.*

So, process residues at p could change without p being aware of it. This means that we cannot hope to directly maintain and update process residues locally as $\mathfrak{A}$ reads u. To remedy this we need the following observation.

Lemma 13 *For any non-empty ideal I, and $p, q \in \mathcal{P}$, $I|_p \cap I|_q$ is generated by the set $primary_p(I) \cap primary_q(I)$.*

Proof We only deal with the interesting case, where $max_p(I)$ and $max_q(I)$ are incomparable with respect to $\sqsubseteq^*$. Let e be a maximal event in the ideal $I|_p \cap I|_q$. e has no successors in $I|_p \cap I|_q$ but $e \sqsubseteq^* max_p(I)$ and $e \sqsubseteq^* max_q(I)$. So, for some $r, s \in \mathcal{P}$, e must be an r-sentry for p with respect to q as well as an s-sentry for q with respect to p. By Lemma 1, $e = \mathit{latest}_{p\to r}(I) = \mathit{latest}_{q\to s}(I)$ and so $e \in \mathit{primary}_p(I) \cap \mathit{primary}_q(I)$.

Therefore, the set of maximal events in $I|_p \cap I|_q$ is contained in $\mathit{primary}_p(I) \cap \mathit{primary}_q(I)$, whence $I|_p \cap I|_q \subseteq (\mathit{primary}_p(I) \cap \mathit{primary}_q(I))\!\downarrow$.

On the other hand, let $e \in \mathit{primary}_p(I) \cap \mathit{primary}_q(I)$. Then $e \in I|_p \cap I|_q$ and so $e\!\downarrow \subseteq I|_p \cap I|_q$, since $I|_p \cap I|_q$ is an ideal. So $(\mathit{primary}_p(I) \cap \mathit{primary}_q(I))\!\downarrow \subseteq I|_p \cap I|_q$.

The degenerate case where $\mathit{max}_p(I)$ and $\mathit{max}_q(I)$ are ordered with respect to $\sqsubseteq^*$ is straightforward and we omit the argument. □

Let us call $\mathcal{R}(u, p, I)$ a *primary residue* if I is generated by a subset E of $\mathit{primary}_p(\mathcal{E})$. Clearly, for $p, q \in \mathcal{P}$, $\mathcal{R}(u, p, \mathcal{E}|_q)$, can be rewritten as $\mathcal{R}(u, p, \mathcal{E}|_p \cap \mathcal{E}|_q)$. So, by the previous result the q-residue $\mathcal{R}(u, p, \mathcal{E}|_q)$ is a primary residue $\mathcal{R}(u, p, E\!\downarrow)$ for some $E \subseteq \mathit{primary}_p(\mathcal{E})$. 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: *In the word $u = bacabba$ shown in Figure 1, $\mathcal{R}(u, s, \mathcal{E}|_p)$ corresponds to the primary residue $\mathcal{R}(u, s, \{\mathit{latest}_{s\to q}(\mathcal{E})\}\!\downarrow)$.*

So, our strategy will 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 ($\diamondsuit$) written in terms of process residues, which 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 $\delta(s_0, u)$.

We first spell out some consequences of Lemma 13.

Corollary 14 *Let $u \in \Sigma^*$ and $p \in \mathcal{P}$.*

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

(ii) Let $Q \subseteq \mathcal{P}$. Then $\mathcal{R}(u, p, \mathcal{E}|_Q)$ is a primary residue $\mathcal{R}(u, p, E\!\downarrow)$ for p. Further, p can effectively compute the set $E \subseteq \mathit{primary}_p(\mathcal{E})$ from the information in $\mathit{primary}_{\{p\}\cup Q}(\mathcal{E})$.

(iii) Let $q, r \in \mathcal{P}$ such that $\mathit{latest}_{p\to r}(\mathcal{E}) \sqsubseteq^* \mathit{latest}_{q\to r}(\mathcal{E})$. Then $\mathcal{R}(u, p, (\mathcal{E}|_q)|_r)$ is a primary residue $\mathcal{R}(u, p, E\!\downarrow)$ for p. Further, p can effectively compute the set $E \subseteq \mathit{primary}_p(\mathcal{E})$ from the information in $\mathit{primary}_p(\mathcal{E})$ and $\mathit{secondary}_q(\mathcal{E})$.

Proof

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

(ii) Let $Q=\{q_1,q_2,\ldots,q_k\}$. We can rewrite $\mathcal{R}(u,p,\mathcal{E}|_Q)$ as $\mathcal{R}(u,p,\bigcup_{i\in[1..k]}\mathcal{E}|_{q_i})$. From Lemma 13 it follows that for each $i\in[1..k]$, p can compute a set $E_i\subseteq primary_p(\mathcal{E})$ from the information in $primary_{\{p,q_i\}}(\mathcal{E})$ such that $\mathcal{R}(u,p,\mathcal{E}|_{q_i})=\mathcal{R}(u,p,E_i\!\downarrow)$. From part (i) of this Corollary, it then follows that $\mathcal{R}(u,p,\mathcal{E}|_Q)=\mathcal{R}(u,p,\bigcup_{i\in[1..k]}\mathcal{E}|_{q_i})=\mathcal{R}(u,p,E\!\downarrow)$ where $E=\bigcup_{i\in[1..k]}E_i$.

(iii) Let $J=\mathcal{E}|_p\cup(\mathcal{E}|_q)|_r$. J is an ideal. By the construction of J, $max_p(J)=max_p(\mathcal{E})$. From the assumption that $latest_{p\to r}(\mathcal{E})\sqsubseteq^* latest_{q\to r}(\mathcal{E})$, we have $max_r(J)=latest_{q\to r}(\mathcal{E})$. So, $J|_p=\mathcal{E}|_p$ and $J|_r=(\mathcal{E}|_q)|_r$. Since $\mathcal{R}(u,p,(\mathcal{E}|_q)|_r)=\mathcal{R}(u,p,\mathcal{E}|_p\cap(\mathcal{E}|_q)|_r)=\mathcal{R}(u,p,J|_p\cap J|_r)$, it suffices to find a subset $E\subseteq primary_p(\mathcal{E})$ which generates $J|_p\cap J|_r$.

By Lemma 13, $J|_p\cap J|_r$ is generated by $primary_p(J)\cap primary_r(J)$. Since $max_p(J)=max_p(\mathcal{E})$, $primary_p(J)=primary_p(\mathcal{E})$.

On the other hand, $primary_r(J)=primary_r(latest_{q\to r}(\mathcal{E})\!\downarrow)$. By definition, this is the set $\{latest_{q\to r\to s}(\mathcal{E})\}_{s\in\mathcal{P}}$.

So the set $E\subseteq primary_p(\mathcal{E})$ generating $J|_p\cap J|_r$ is given by $E=primary_p(J)\cap primary_r(J)=primary_p(\mathcal{E})\cap\{latest_{q\to r\to s}(\mathcal{E})\}_{s\in\mathcal{P}}$ and can be computed from $primary_p(\mathcal{E})$ and $secondary_q(\mathcal{E})$.

□

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

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 $\{\mathcal{R}(u,p,E\!\downarrow)\}_{p\in\mathcal{P},E\subseteq primary_p(\mathcal{E}_u)}$ 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 $w\in\{\mathcal{R}(u,p,E\!\downarrow)\}_{E\subseteq primary_p(\mathcal{E}_u)}$. We want to compute for each p the corresponding functions after the word ua.

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

Proposition 15 *If $p\notin\theta(a)$ then every subset $E\subseteq primary_p(\mathcal{E}_{ua})$ is also a subset of $primary_p(\mathcal{E}_u)$ and the primary residue $\mathcal{R}(ua,p,E\!\downarrow)$ is the same as the primary residue $\mathcal{R}(u,p,E\!\downarrow)$.*

Proof This follows immediately from the fact that $\mathcal{E}_{ua}|_p = \mathcal{E}_u|_p$ and $primary_p(\mathcal{E}_{ua}) = primary_p(\mathcal{E}_u)$. $\Box$

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 $\theta(a)$ after u.

Lemma 16 *Let $p \in \theta(a)$ and $E \subseteq primary_p(\mathcal{E}_{ua})$. The function f_w corresponding to the primary residue $w = \mathcal{R}(ua, p, E\!\downarrow)$ can be computed from the primary residues at u of the processes in $\theta(a)$ using the information in $primary_{\theta(a)}(\mathcal{E}_u)$ and $secondary_{\theta(a)}(\mathcal{E}_u)$.*

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

Case 1: $(e_a \in E)$
Since $E\!\downarrow = e_a\!\downarrow = \mathcal{E}_{ua}|_p$, the residue $\mathcal{R}(ua, p, E\!\downarrow) = \mathcal{R}(ua, p, e_a\!\downarrow)$ 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 $w = ua[\mathcal{E}_{ua}|_p - E\!\downarrow]$. By Lemma 11, we know that

$$ua[\mathcal{E}_{ua}|_p] \sim ua[E\!\downarrow]ua[\mathcal{E}_{ua}|_p - E\!\downarrow]. \tag{1}$$

But $ua[\mathcal{E}_{ua}|_p] = u[\mathcal{E}_u|_{\theta(a)}]a$ and so we have

$$ua[\mathcal{E}_{ua}|_p] = u[\mathcal{E}_u|_{\theta(a)}]a \sim u[E\!\downarrow]u[\mathcal{E}_u|_{\theta(a)} - E\!\downarrow]a. \tag{2}$$

Since $e_a \notin E\!\downarrow$, $ua[E\!\downarrow] = u[E\!\downarrow]$. Thus, cancelling $u[E\!\downarrow]$ from the right hand sides of (1) and (2) above, we have $u[\mathcal{E}_u|_{\theta(a)} - E\!\downarrow]a \sim ua[\mathcal{E}_{ua}|_p - E\!\downarrow]$. So, by Propositions 8 and 9, to compute the function f_w, it suffices to compute the function corresponding to $u[\mathcal{E}_u|_{\theta(a)} - E\!\downarrow]a$.

Let $\theta(a) = \{p_1, p_2, \ldots, p_k\}$, 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: $I_0 = E\!\downarrow$ 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 12, $u[I_k] \sim u[I_0]u[I_1 - I_0]\cdots u[I_k - I_{k-1}]$. Since $u[I_k] = u[\mathcal{E}_u|_{\theta(a)}]$ and $u[I_0] = u[E\!\downarrow]$, from (2) above it follows that the word $u[\mathcal{E}_u|_{\theta(a)} - E\!\downarrow]a$ which we seek is $\sim$-equivalent to the word $u[I_1 - I_0]\cdots u[I_k - I_{k-1}]a$.

Claim: For each $j \in [1..k]$, $u[I_j - I_{j-1}]$ is a primary residue $\mathcal{R}(u, p_j, F_j\!\downarrow)$, 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 $secondary_{\theta(a)}(\mathcal{E}_u)$.

Assuming the claim, for each word $w_j = u[I_j - I_{j-1}]$, 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 $u[I_1 - I_0]\cdots u[I_k - I_{k-1}]a$, 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 $\theta(a)$; i.e., $E \subseteq primary_{\theta(a)}(\mathcal{E}_u)$. For $i \in [1..k]$, let $E_i = E \cap primary_{p_i}(\mathcal{E}_u)$.

First consider $u[I_1 - I_0]$.

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

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 $latest_{p_1 \to q_e}(\mathcal{E}_u) \sqsubseteq^* latest_{p_i \to q_e}(\mathcal{E}_u)$. So, by part (iii) of Corollary 14, the residue $\mathcal{R}(u, p_1, e\!\downarrow) = \mathcal{R}(u, p_1, (\mathcal{E}_u|_{p_i})|_{q_e})$ corresponds to a primary residue $\mathcal{R}(u, p_1, G_e\!\downarrow)$, 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 14, $\mathcal{R}(u, p_1, E'\!\downarrow) = \mathcal{R}(u, p_1, \bigcup_{e \in E'} e\!\downarrow)$ is a primary residue $\mathcal{R}(u, p_1, G_1\!\downarrow)$ where $G_1 = \bigcup_{e \in E'} G_e$.

$\mathcal{R}(u, p_1, E_1\!\downarrow)$ is a primary residue since $E_1 \subseteq primary_p(\mathcal{E}_u)$. Applying part (i) of Corollary 14 again, $\mathcal{R}(u, p_1, (E_1\!\downarrow \cup E'\!\downarrow))$ corresponds to the primary residue $\mathcal{R}(u, p_1, F_1\!\downarrow)$, where $F_1 = E_1 \cup G_1$.

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

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

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

By part (ii) of Corollary 14, p_j can compute from $primary_{Q_j}(\mathcal{E}_u)$ a set $H_j \subseteq primary_{p_j}(\mathcal{E}_u)$ such that $\mathcal{R}(u, p_j, \mathcal{E}_u|_{Q_{j-1}})$ corresponds to the primary residue $\mathcal{R}(u, q, H_j\!\downarrow)$.

We now use part (i) of Corollary 14 to establish that $\mathcal{R}(u, p_j, E\!\downarrow \cup \mathcal{E}_u|_{Q_{j-1}})$ corresponds to the primary residue $\mathcal{R}(u, p_j, F_j\!\downarrow)$, where $F_j = G_j \cup H_j$.

□

An asynchronous automaton for L

Our analysis of process residues and primary residues immediately yields a deterministic asynchronous automaton $\mathfrak{A} = (\{V_p\}_{p \in \mathcal{P}}, \{\to_a\}_{a \in \Sigma}, \mathcal{V}_0, \mathcal{V}_F)$ which accepts the language L. Recall that $\mathfrak{B} = (S, \Sigma, \delta, s_0, S_F)$ is the minimal DFA recognizing L.

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

- Primary, secondary and tertiary 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 0. 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 $\theta(a)$ 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 16.

Other than comparing primary information, the only operation used in updating the primary residues at p (Lemma 16) is function composition. This is easily achieved using the data available in the states of the processes which synchronized.

The final states of $\mathfrak{A}$ 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 $\mathfrak{A}$ such that $\mathfrak{A}$ is in $\vec{v}$ after reading an input word u.

By Corollary 14 (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 $\mathcal{V}_F$ of $\mathfrak{A}$ 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 $\vec{v} \in \mathcal{V}_F$. 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 $\mathcal{P}$.

Of course, we have the same minor complication here which we came across when defining the local functions g_a which computed the latest gossip function: We have not bothered to verify whether $\vec{v}$ is a "meaningful" global state in $\mathfrak{A}$. However, as we argued then, those states which are not "meaningful" are also unreachable, so it does not matter if we accidentally add such states to $\mathcal{V}_F$.

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

Theorem 17 *The language accepted by* $\mathfrak{A}$ *is exactly* L.

The size of $\mathfrak{A}$

Proposition 18 *Let* $M = |S|$ *and* $N = |\mathcal{P}|$, *where* S *is the set of states of* $\mathfrak{B}$, *the minimal DFA recognizing* L, *and* $\mathcal{P}$ *is the set of processes in the corresponding asynchronous automaton* $\mathfrak{A}$ *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 7, we know that the primary, secondary and tertiary information that we require to keep track of the latest gossip can be stored in $O(N^3 \log N)$ 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 $2^{O(2^N M \log M)}$. $\square$

8 Comparison with Zielonka's construction

Let us compare our construction of $\mathfrak{A}$ with Zielonka's original proof [Zie1]. Zielonka starts by defining an equivalence relation $\approx$ on words (actually traces) which we can describe in our terminology as follows.

For sets X and Σ, a Σ-labelled graph over X is a pair (G, λ) where $G(X, E)$ is a directed graph whose vertex set is X and $\lambda : X \to \Sigma$ is a labelling function.

Let $\mathcal{P}$ be the set of processes in the asynchronous automaton we want to construct and $\Sigma' = \Sigma \cup \{\dagger\}$, for some $\dagger \notin \Sigma$. With each word $u \in \Sigma^*$, associate a Σ'-labelled graph $\textsc{Last}(u)$ over $\mathcal{P} \times \mathcal{P}$ where $\textsc{Last}(u) = (G(\mathcal{P} \times \mathcal{P}, E_u), \lambda_u)$ is given as follows:

- For $(p_1, q_1), (p_2, q_2) \in \mathcal{P} \times \mathcal{P}$, $(p_1, q_1)\ E_u\ (p_2, q_2)$ iff $\mathit{latest}_{p_1 \to q_1}(\mathcal{E}_u) \sqsubseteq^* \mathit{latest}_{p_2 \to q_2}(\mathcal{E}_u)$.
- For $(p, q) \in \mathcal{P} \times \mathcal{P}$, let $e \in \mathcal{E}_u$ be the event corresponding to $\mathit{latest}_{p \to q}(\mathcal{E}_u)$. If e is the initial event 0, then $\lambda_u((p, q)) = \dagger$. Otherwise e must be of the form (i, a), in which case $\lambda_u((p, q)) = a$.

So, the graph $\textsc{Last}(u)$ of [Zie1] records the structure of the primary information after u. This is the first notion needed to specify the equivalence relation $\approx$.

The second ingredient present is a version of residues. For each word u and each subset P of processes, the suffix of u with respect to P is the word $\mathcal{S}_P(u) = u[\mathcal{E}_u - \mathcal{E}_u|_P]$.

We can now define $\approx$. For all $u, u' \in \Sigma^*$, $u \approx u'$ iff the following conditions hold:

- $\textsc{Last}(u) = \textsc{Last}(u')$.
- For all $P \subseteq \mathcal{P}$, $\mathcal{S}_P(u) \equiv_L \mathcal{S}_P(u')$. (Recall that $\equiv_L$ is the syntactic congruence defined by the language L.)

It is easy to see that if $u \sim u'$ then $\textsc{Last}(u) = \textsc{Last}(u')$. Similarly, if $u \sim u'$ then for every $P \subseteq \mathcal{P}$, $\mathcal{S}_P(u) \sim \mathcal{S}_P(u')$ and so $\mathcal{S}_P(u) \equiv_L \mathcal{S}_P(u')$. Thus, if $u \sim u'$ then $u \approx u'$ and so it does not matter that we have defined $\approx$ in terms of words rather than in terms of traces as Zielonka originally did. Clearly, $\approx$ is of finite index for any recognizable trace language L.

Let $\langle u \rangle$ denote the equivalence class of u with respect to $\approx$. In Zielonka's construction, each process p keeps track of $\langle u[\mathcal{E}_u|_p] \rangle$, the equivalence class of the p-view of u. Thus, after reading u, the automaton is in the global state $(\langle u[\mathcal{E}_u|_{p_1}] \rangle, \langle u[\mathcal{E}_u|_{p_2}] \rangle, \ldots, \langle u[\mathcal{E}_u|_{p_N}] \rangle)$, where $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$. A global state of this form is an accepting state iff $u \in L$.

In general, the graph $\textsc{Last}(u)$ as well as the set of suffixes $\{\mathcal{S}_P(u)\}_{P \subseteq \mathcal{P}}$ describe *global* properties of the word u. However, when we restrict our attention to the p-view $u[\mathcal{E}_u|_p]$ of a process $p \in \mathcal{P}$, they correspond to *local* notions which we have already encountered. The graph $\textsc{Last}(u[\mathcal{E}_u|_p])$ describes the structure of the secondary information of p after

u. And each suffix $\mathcal{S}_P(u[\mathcal{E}_u|_p])$, $P \subseteq \mathcal{P}$, corresponds to the primary residue of u at p with respect to $E = \{latest_{p\to q}(\mathcal{E}_u)\}_{q\in P}$.

From this, it is clear that our construction is essentially a "disguised" version of Zielonka's original construction. What have we achieved by adopting this disguise?

The main benefit is that our construction is significantly more transparent than Zielonka's (which is generally acknowledged to be difficult to assimilate [CMZ, Pig].)

First of all, we clearly distinguish between process residues and primary residues. Process residues are used to compute the function f_u corresponding to the $\equiv_L$-congruence class of a word u via the expression ($\diamondsuit$) (page 19). However, since these residues cannot be maintained locally, each process keeps track of primary residues instead. Using primary information, process residues can be effectively rewritten in terms of primary residues and so it suffices to keep track of primary information and primary residues locally. This, essentially, is the intuition behind Zielonka's construction and the reason the original construction is rather opaque is because these connections are not made explicit in [Zie1].

The second difference is that the local transition functions are defined much more "naturally" in our setup. We explicitly maintain primary information and information about residues in the local states of processes. Thus, we can define the local transition functions as "procedures" or "algorithms" which manipulate the information stored in the local states in a concrete way. This makes it easier to understand, overall, what the asynchronous automaton is computing. In addition, it also enables us to effectively describe the automaton much more concisely than if the transition functions are defined in terms of the equivalence classes of $\approx$, as in [Zie1]. In Zielonka's construction, there is no structure to the names given to these equivalence classes. So, for specifying the transition functions, there is no alternative but to exhaustively list out all the entries in the form of an enormous table.

Let us now focus on the connection between the information maintained by the gossip automaton and the information present in the graphs $\{\textsc{Last}(u[\mathcal{E}_u|_p])\}_{p\in\mathcal{P}}$. It is clear that Zielonka's construction provides us with an effective procedure for locally updating these graphs whenever a synchronization occurs. However, this update mechanism is "uninformative" in the following sense: When a set of processes $c \subseteq \mathcal{P}$ synchronizes after u, we cannot directly extract the values of the functions $\{best_c(u,q)\}_{q\in\mathcal{P}}$ (page 6) which we are interested in computing via the gossip automaton. (This is not to say that this information cannot be computed *indirectly* from the way these graphs are updated in [Zie1]—all that we claim is that the procedure, as it stands, does not compute the functions we seek.)

Even if Zielonka's update procedure were modified to supply the values of $\{best_c(u,q)\}_{q\in\mathcal{P}}$ when $c \subseteq \mathcal{P}$ meets after u, the resulting automaton would be much larger, in real terms, than the gossip automaton. The first point to note is that the number of states generated by Zielonka's construction would be slightly larger than the number of states in the gossip automaton. Let $N = |\mathcal{P}|$ and $K = |\Sigma|$. Then there are $2^{O(N^4+N^2\log K)}$ distinct Σ'-labelled graphs over $\mathcal{P}\times\mathcal{P}$ and hence at least the same number of possible local states in the hypothetical gossip automaton extracted from Zielonka's construction. Compare this with the $2^{O(N^3\log N)}$ states of our automaton.

More crucially, we again observe that the local transition functions of our automaton are simple procedures manipulating concrete information stored in the local states of processes. Thus our gossip automaton can be described in space polynomial in N. On the other hand, since the transition functions in Zielonka's construction can only be described using a table spanning all possible choices from an (exponential) number of equivalence classes, we need space exponential in N to fully specify the corresponding automaton arising from Zielonka's construction.

9 Discussion

We now discuss the connections between our results and other work in related areas. First, we would like to point out that both "gossiping" and "bounded time-stamps" have been studied in the literature, but in contexts very different from ours.

Studies of "gossiping" in networks have traditionally focussed on efficiently disseminating a fixed piece of information (or gossip) from one node to all other nodes in a network [HHL]. The main aim is to find an optimal sequence of communications to distribute data for a given network topology.

Israeli and Li [IL] introduced the notion of "bounded time-stamps" and argued that these were fundamental in solving many problems in distributed systems—notably that of creating what are called "atomic registers"[LV]. However, both their work and that of others in the area [DS, CS] is based on a shared-memory model, which is quite different in spirit from the asynchronous automaton model. Thus, the intuition underlying their notion of time-stamping is quite different from ours.

Though our algorithm can be implemented as an asynchronous automaton, it correctly computes the latest gossip function locally for *any* input word. In other words, the set of communication sequences generated by the underlying system need not be regular. Our algorithm will also work on sequences generated, for instance, by N communicating Turing machines.

From this point of view, the construction of the gossip automaton establishes a non-obvious property for *all* synchronous systems. Suppose an agent p_1 has a private variable X which no other agent can modify, and agents $\{p_2, p_3, \ldots, p_N\}$ keep track of the latest value of X that they have heard of from p_1 (either directly or indirectly). Then, along *any* run of the system, bounded time-stamps suffice for determining which of $\{p_2, p_3, \ldots, p_N\}$ have the most recent value of X. This is important, for example, for crash recovery. If the system crashes and p_1 fails to come alive after the crash, the other agents can get together and synthesize an optimal "last-known" state of p_1 by comparing their information about p_1.

Turning now to Zielonka's theorem, as we mentioned in the Introduction, others have also come up with alternative proofs of this basic result [CMZ, Die]. However, all these new proofs are based on asynchronous *cellular* automata, introduced by Zielonka in [Zie2], and rely on the fact that asynchronous cellular automata can be converted into asynchronous automata in a relatively straightforward manner.

An asynchronous cellular automaton processes input strings over a concurrent alphabet $(\Sigma, \mathcal{I})$. Each letter a in Σ gives rise to a process p_a. p_a makes a move on input a by examining its own state together with the states of the processes $\{p_b \mid (a,b) \notin \mathcal{I}\}$. However, when p_a moves, it can only change its own state. Thus, p_a looks at processes in the "neighbourhood" of a in the dependence graph $G_{\mathfrak{D}}$ associated with $(\Sigma, \mathcal{I})$. In this sense, these automata generalize conventional cellular automata.

Asynchronous cellular automata, like asynchronous automata, have a built in notion of independence—if two letters do not appear in each other's neighbourhoods, the corresponding components can move without interfering with each other. So, they are guaranteed to recognize $\sim$-consistent languages. From the point of view of trace theory their structure seems more closely related to the algebraic structure of traces, thus making them more "friendly" machines to work with in proving Zielonka's theorem [CMZ, Die]. Loosely speaking, this can be ascribed to the fact that trace theory "concentrates" more on the alphabet Σ than on the processes $\mathcal{P}$ in an asynchronous automaton.

The main feature of the construction in [CMZ] is that it provides a general "recipe" for constructing asynchronous cellular automata from special types of functions called *asynchronous mappings.* These mappings correspond, in a sense, to our notion of locally computable functions. Asynchronous mappings provide more intuition about the behaviour of these automata than transition functions based on equivalence relations do. Thus, they help clarify the general mystery surrounding constructions like those used by Zielonka in [Zie1]. However, the "mechanical" procedure for transforming asynchronous mappings into asynchronous (cellular) automata does not provide a concise specification of the resulting automaton. The transition functions of the final automaton still have to be defined in terms of an enormous table, and, in our view, the overall description of the automaton remains unsatisfactorily large.

Like our construction, the proof given in [CMZ] factors through a bounded time-stamping argument. They keep track of what they call the first and second approximations of a trace. These are directed graphs which compare the partial views generated by maximal *letters* in a trace.

The first approximation of a trace $[u]$ records the causal order between the maximum a-event and the maximum b-event in $\mathcal{E}_u$, for each pair of letters $a, b \in \Sigma$. The second approximation of $[u]$ records the causal order between the maximum b-event in the a-view of $\mathcal{E}_u$ and the maximum c-event in the d-view of $\mathcal{E}_u$ for all $a, b, c, d \in \Sigma$, where for $a \in \Sigma$, the a-view of $\mathcal{E}_u$ corresponds to the principal ideal generated by the maximum a-event in $\mathcal{E}_u$.

Roughly speaking, the first approximation of a trace corresponds to keeping track of the causal order between the maximum p-events of $\mathcal{E}_u$ for all $p \in \mathcal{P}$. Similarly, the second approximation of a trace is similar to the graph $\textsc{Last}(u)$ (page 26) recording the global structure of the primary information in $\mathcal{E}_u$. It is not difficult to extract the causal order between primary events (i.e., the graph $\textsc{Last}(u)$) from the first and second approximations of a trace. However, recall that Zielonka's procedure for updating $\{\textsc{Last}(u[\mathcal{E}_u|_p)\}_{p\in\mathcal{P}}$ does not immediately yield a complete solution to the gossip problem. Similarly, locally keeping track of the first and second approximations of the a-views of a trace falls short of directly providing the values of the functions $\{best_c(u, q)\}_{q\in\mathcal{P}}$ when a subset $c \subseteq \mathcal{P}$ of processes meets after u.

Notice that approximations of a trace deal with elements in Σ, whereas primary and secondary "gossip" information is concerned *only* with processes and *ignores* the alphabet Σ altogether. There is, in general, a mismatch between the size of Σ and the number of processes in $\mathcal{P}$. So, it is not surprising that the gossip automaton as defined here does not keep track of the approximations of a trace. However, our automaton can easily be modified to keep track of primary, secondary and tertiary information with respect to letters rather than processes. So, the first and second approximation of a trace can be computed in our framework. (Observe that $|\Sigma|$ could be exponential in $|\mathcal{P}|$. As a result, computing approximations of a trace, instead of just keeping track of primary and secondary information with respect to processes, could blow up the size of the gossip automaton tremendously.)

We can go back and forth between asynchronous cellular automata and asynchronous automata through polynomial-size transformations, as shown by Pighizzini [Pig]. So, for computing properties of traces, we can work with either model and then apply Pighizzini's translation without incurring a significant cost in the state space.

However, at an operational level, asynchronous cellular automata and asynchronous automata are based on very different kinds of intuition. Thus, if we want to understand *how* to compute something using asynchronous automata, very little insight is gained by solving the problem on hand using asynchronous cellular automata and then translating that solution back to asynchronous automata. So, for instance, though bounded time-stamping algorithms appear both in the construction of [CMZ] and in the construction of the gossip automaton, it is difficult to see how our algorithm can be derived from the one described in [CMZ].

We feel that it is important to have a clear understanding of what can be computed locally and directly in the framework of asynchronous automata since these automata are natural models of distributed systems. They correspond to a very intuitive notion of synchronizing finite state machines which exchange information whenever they meet. Asynchronous automata are closely connected to well-established formalisms for describing concurrent systems. For instance, an automaton of this type can easily be represented as a labelled 1-safe Petri net [Zie1]. These automata also generalize synchronization mechanisms which have been studied in process calculi like TCSP [BHR].

On the other hand, at the operational level, asynchronous cellular automata do not correspond to any standard model of distributed systems. We can view the distributed state space of these automata as a concurrent-read exclusive-write shared memory. However, distributed algorithms based on the shared memory model normally assume that the store is equally shared by *all* processes. Instead, in an asynchronous cellular automaton, the memory is shared in a localized manner.

Building the gossip automaton appears to be a basic step in tackling many problems in the theory of asynchronous automata. In addition to our new proof of Zielonka's theorem, we have found other important applications of our solution to the gossip problem. In [KMS], a determinization construction is presented for asynchronous automata using a generalization of the classical subset construction for finite automata. This construction allows us to keep track of the global states which are currently valid at any stage of a computation by a non-deterministic asynchronous automaton. The data maintained by the gossip automaton plays a crucial role in determining how much information can be safely "forgotten" by the subset automaton without sacrificing global consistency.

We feel that this determinization construction will lead to a direct complementation construction for Büchi asynchronous automata [GP]. These automata recognize ω-regular trace languages, which are a natural trace-theoretic generalization of ω-regular string languages. Büchi asynchronous automata are beginning to play an important role in concurrency theory. They are known to be closed under complementation for algebraic reasons, but the only effective complementation construction provided so far has been for Büchi asychronous cellular automata [Mus]. There appears to be no straightforward way of applying this construction to Büchi asynchronous automata.

Recently, Thiagarajan [Thi] has developed an extension of propositional linear-time temporal logic which is interpreted over infinite traces rather than infinite linear sequences. This logic appears to be quite expressive, while remaining decidable (unlike several other partial-order logics studied in the literature [LPRT]). The decision procedure for this logic is automata-theoretic, in the style of Vardi and Wolper [VW], except that it makes use of Büchi asychronous automata rather than conventional Büchi automata. The gossip automaton plays a crucial role in this construction as well. In fact, *this* was the original motivation for constructing the gossip automaton.

Acknowledgments P.S. Thiagarajan suggested the gossip problem. We have benefited greatly from discussions with him. The reformulation of the gossip automaton in terms of ideals and frontiers emerged during work with Nils Klarlund on determinizing asynchronous automata. His suggestions have been a great help in cleaning up the presentation. We also thank Robert de Simone for pointing out some subtle technical errors. The first author was partially supported by IFCPAR Project 502-1.

References

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

[BHR] S.D. Brookes, C.A.R. Hoare, A.W. Roscoe: A theory of communicating sequential processes, *J. ACM*, **31** (1984) 560–599.

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

[CS] R. Cori, E. Sopena: Some combinatorial aspects of time-stamp systems, *Europ. J. Combinatorics*, **14** (1993) 95–102.

[Die] V. Diekert: *Combinatorics on traces*, *LNCS* **454** (1990).

[DS] D. Dolev, N. Shavit: Bounded concurrent time-stamps are constructible, *Proc. ACM STOC* (1989) 454–466.

[GP] P. Gastin, A. Petit: Asynchronous cellular automata for infinite traces, *Proc. ICALP '92*, *LNCS* **623** (1992) 583–594.

[HHL] S.M. Hedetniemi, S.T. Hedetniemi, A.L. Liestman: A survey of gossiping and broadcasting in communication networks, *Networks*, **18** (1988) 319–349.

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

[IL] A. Israeli, M. Li: Bounded time-stamps, *Proc. 28th IEEE FOCS* (1987) 371–382.

[KMS] N. Klarlund, M. Mukund, M. Sohoni: Determinizing asynchronous automata, *to appear in Proc. ICALP 1994*. A preliminary version appears as *Report TCS-93-5*, School of Mathematics, SPIC Science Foundation, Madras, India (1993).

[LV] M. Li, P. Vitanyi: How to share concurrent asynchronous wait free variables, *Proc. ICALP '89*, *LNCS* **372** (1989) 488–507.

[LPRT] K. Lodaya, R. Parikh, R. Ramanujam, P.S. Thiagarajan: A logical study of distributed transition systems, *to appear in Inform. and Comput.* Also available as *Report TCS-93-8*, School of Mathematics, SPIC Science Foundation, Madras, India (1993).

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

[MS] M. Mukund, M. Sohoni: Keeping track of the latest gossip: Bounded time-stamps suffice, *Proc. FST&TCS '93*, *LNCS* **761** (1993) 388–399.

[Mus] A. Muscholl: On the complementation of Büchi asynchronous cellular automata, *to appear in Proc. ICALP 1994*.

[Pig] G. Pighizzini: *Recognizable trace languages and asynchronous automata*, Phd Thesis, Dip. Scienze dell'Informazione, Univ. di Milano (1993).

[Sta] R.P. Stanley: *Enumerative combinatorics: Volume I*, Wadsworth & Brookes/Cole (1986).

[Thi] P.S. Thiagarajan: TrPTL: A trace based extension of linear time temporal logic, *to appear in Proc. IEEE LICS 1994*. A preliminary version appears as *Report TCS-93-6*, School of Mathematics, SPIC Science Foundation, Madras (1993).

[VW] M. Vardi, P. Wolper: An automata theoretic approach to automatic program verification, *Proc. 1st IEEE LICS* (1986) 332–345.

[Zie1] W. Zielonka: Notes on finite asynchronous automata, *R.A.I.R.O.—Inform. Théor. Appl.*, **21** (1987) 99–135.

[Zie2] W. Zielonka: Safe executions of recognizable trace languages, *Proc. Logical Foundations of Computer Science*, *LNCS* **363** (1989) 278–289.