A Presentation of Regular Languages in the Assumption-Commitment Framework

Swarup Mohalik

Institute of Mathematical Sciences
C. I. T. Campus
Chennai 600 113, India.

swarup@imsc.ernet.in

R. Ramanujam

Institute of Mathematical Sciences
C. I. T. Campus
Chennai 600 113, India.

jam@imsc.ernet.in

Abstract

A distributed presentation of a regular language L is a system of communicating automata accepting L, where the constraint on distribution is given in the form of a distributed alphabet. We study such presentations in the assumption - commitment *framework, where each process makes assumptions about other processes in the system. Symmetrically, each process makes commitments which the other processes can rely on. In [10], we showed that the class of (Mazurkiewicz) regular trace languages can be presented in this framework. In this paper, we show that* every *regular language can be thus decomposed and present a version of Kleene's theorem using top-level parallelism.*

1 Introduction

A central issue in the theory of distributed systems is that of *local reasoning*: each process in the system has only a partial view of the global states of the system, but makes assumptions about the invisible states and proceeds on its own computation. Global consistency of the system is maintained by ensuring that these assumptions are met.

This brief remark is the essential characteristic of the *assumption - commitment* [6, 9] or *rely - guarantee* [8] paradigm studied in distributed computing. Such frameworks aim at reasoning about the behaviour of each process separately, assuming that others maintain relevant properties and reasoning globally about their compatibility.

In the early eighties, considerable effort went into studying this framework in the context of concurrent programs. (See [14, 11] for detailed accounts of such studies in Hoare logics.) Despite the naturalness of the idea, it was recognized that such compositional reasoning was difficult, particularly for liveness properties. However, the advent of automata-theoretic techniques for model checking temporal properties provides a source of encouragement ([1, 3]) for examining the framework further from an automata-theoretic viewpoint.

A theoretically simple model of a distributed system consists of a static network of finite state machines, the number of components and the channels of communication between them is fixed, and messages are not buffered, so that all communication can be studied as synchronization. It was such a simplification that led to the model of Communicating Sequential Processes [7]. The attractiveness of such a model is that when there are n processes in the system, we can represent the system as a parallel composition $(P_1||P_2||\ldots||P_n)$, and in each of the P_i we can refer to synchronizations with other P_j.

The problem of studying assumptions and commitments in such a simple setup is already interesting. In terms of languages, what kind of compatible parallel composition makes sense ? What is the power of the framework, or in other words, what class of languages admit such a distributed presentation ? In [10], it was shown that by a particular constraint on the structure of assumptions, we can obtain the class of *regular trace languages* of Mazurkiewicz [4]. In this paper, we further investigate the issue, and show that the framework is powerful enough to include `all regular languages` over the given alphabet. The crucial difference is this: regular trace languages are closed under 'commutations' of independent actions, which can be done in any order, and this is typical of concurrent behaviour. When we move to arbitrary regular languages we need the ability to 'force' specific interleavings – for instance, when a and b are independent, the language $(ab)^*$ specifies that a is always preferred over b; coming up with product constructions on automata that achieve such forcing systematically is the difficulty. Hence the problem here is different from that in the construction of, say, asynchronous automata [13] or cellular asynchronous automata [2], where global states are decomposed preserving concurrency.

The basic idea is this: alongwith a distributed alphabet, we also have an assumption (commitment) alphabet, which comes with a partial order $\preceq$ defined on it. At each local state of a process, an assumption is specified about each of the other processes in the system, as well as a commitment offered. For instance, in a system of two processes, P_1 and P_2, at a local state s of P_1, the assumption-commitment pair may be (λ_1, λ_2) and similarly (γ_1, γ_2) for process P_2 in state t. At state s, P_1 commits to maintaining λ_1 assuming that P_2 would commit to λ_2, and at state t, P_2 commits to maintaining γ_2 assuming that P_1 would commit to γ_1. Then the global state (s,t) is compatible iff $\lambda_2 \preceq \gamma_2$ and $\gamma_1 \preceq \lambda_1$. We may think of λ_2 as a logical assertion whose invariance is maintained by P_2-local states in all global states that map to s for P_1. In this sense, the notion studied here bears close resemblance to *knowledge-based programs* [5], where a process makes conditional transitions based on what it *knows* about other processes. (While the connection is intuitive, we do not as yet have a formal result relating knowledge-based programs with the automata studied here.)

As a reasoning framework, what we propose here can be seen as a form of *local* reasoning. Each process in a distributed system has a partial view of the global state of the system and makes assumptions about what the 'current' global state might be, depending on its own local state. Compatibility of these assumptions is ensured by the protocol itself so that when all processes follow the protocol and the system execution is initiated at a compatible global state, all the states reached according to the protocol are compatible. This helps to isolate (and hopefully minimize) the global reasoning required.

This should be distinguished from the forms of *compositional* reasoning advocated in [9] and others, where each component in the system is treated as a blackbox, and from assumptions about invariance properties of the components, we derive those of the entire system. This works very well for safety properties, but becomes difficult for liveness. In our approach, we do not adapt a blackbox approach at all, but look into the internal structure of components. Each process knows the program (or transition system) structure of the other components and uses this knowledge in the assumptions it makes, but is only unaware of what is 'currently' happening at other locations.

This is also different from what is termed *modular* reasoning ([12]), where processes operate in environments about which nothing can be assumed at all. This is necessary in the verification of open systems, but differs from the context we discuss, where the distributed systems discussed are closed systems. The environment for any process is simply the set of other processes in the system and hence it knows quite a bit of useful information about the environment.

The main result in this paper is that for every regular language we can find a commitment alphabet such that the product of a system of assumption-based processes accepts exactly that language. This result can be seen as expressive completeness of the assumption - commitment framework in the context of finite state distributed systems. It leads to a top-level parallel version of Kleene's theorem, and gives us a notion of compatible synchronized shuffle of languages. The results easily generalize to behaviours given as infinite words. This may be of importance, since model checking temporal logics is done using automata over infinite words, and distributed presentations of regular infinite behaviours may lead to localized methods for model checking.

2 Assumption-compatible systems

We begin with some notation. (The preliminaries and notation are as in [MR97].) An automaton over a finite nonempty alphabet Σ is a tuple $M = (Q, \delta, q_0, F)$, where Q is a **finite** set of states, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states of the automaton. As usual, we write $q \xrightarrow{a} q'$ to mean that $(q, a, q') \in \delta$. Note that the automaton is in general nondeterministic, and is deterministic only when δ is a function on $S \times \Sigma$. A *run* from $q_0 \in Q$ to $q_k \in Q$ on $x = a_1 a_2 \ldots a_k$ is a sequence of transitions of the automaton: $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_k} q_k$, where $k \geq 0$. When $q_k \in F$, we say that x is *accepted* by M, and denote the set of all strings by $L(M)$, the (regular) language accepted by M.

Here we consider systems of such automata, for which we define the notion of a **distributed alphabet**. Fix a finite set of *locations* $Loc = \{1, \ldots, n\}$, $n > 0$. A distributed alphabet is a tuple $\widetilde{\Sigma} = (\Sigma_1, \ldots, \Sigma_n)$, where each Σ_i is a finite nonempty set of actions of the i^{th} automaton. When $a \in \Sigma_i \cap \Sigma_j$, $i \neq j$, we think of it as a synchronization action between i and j. Given a distributed alphabet $\widetilde{\Sigma}$, we often speak of the set $\Sigma \stackrel{\text{def}}{=} \Sigma_1 \ \cup \ ... \ \cup \ \Sigma_n$ as the alphabet of the system. We also make implicit use of the associated function $loc : \Sigma \rightarrow 2^{Loc}$ defined by $loc(a) \stackrel{\text{def}}{=} \{i \mid a \in \Sigma_i\}$.

As suggested in the last section, we have another alphabet, which we call the **commitment alphabet**. This is a tuple $\widetilde{\mathcal{C}} = <(\mathcal{C}_1, \preceq_1), \cdots, (\mathcal{C}_n, \preceq_n)>$, where each $\mathcal{C}_i$ is a finite nonempty set, and for all $i \neq j$, $\mathcal{C}_i \cap \mathcal{C}_j = \{\perp\}$, and $\preceq_i$ is a partial order on $\mathcal{C}_i$, with $\perp$ as its least element. The element $\perp$ is the null assumption (or commitment). We call $\mathcal{C} = \mathcal{C}_1 \cup \ldots \cup \mathcal{C}_n$ the *commit set*.

Let $\Phi \stackrel{\text{def}}{=} \{\phi : Loc \mapsto \mathcal{C} \mid \forall i \in Loc, \phi(i) \in \mathcal{C}_i\}$.

We now define *AC-automata*, the class of machines which make assumptions and commitments. We first define individual automata and then systems of such automata.

Definition 2.1 *Let $i \in \{1, 2, \ldots, n\}$.*

1. *An* **AC-automaton** *over $(\widetilde{\Sigma}, \widetilde{\mathcal{C}})$ is a tuple $\mathcal{M}_i = (Q_i, \longrightarrow_i, q_i^0, f_i)$ where, Q_i is a finite set of states, q_i^0 is the initial state, $\longrightarrow_i \subseteq Q_i \times \Sigma_i \times Q_i$ is the transition relation and $f_i : Q_i \rightarrow \Phi$. At a state $p \in Q_i$, $f_i(p)(j)$ gives automaton i's assumption about automaton j.*

2. *An* **Assumption - compatible system (ACS)** *over $(\widetilde{\Sigma}, \widetilde{\mathcal{C}})$ is given by a tuple $\widetilde{\mathcal{M}} = (\mathcal{M}_1, \cdots, \mathcal{M}_n, F)$, where each $\mathcal{M}_i$ is an AC-automaton over $(\widetilde{\Sigma}, \widetilde{\mathcal{C}})$, and $F \subseteq (Q_1 \times \ldots \times Q_n)$.*

It is not necessary that the commit alphabet be fixed universally for the system, as we have done above. We could have defined each AC-automaton with its own commit alphabet and subsequently ensured in the definition of systems that for all $i, j \in Loc$, the j^{th} commit set of automaton i is contained in the j^{th} commit set of automaton j. However, this tends to clutter up notation considerably, so we stick with the (more restricted) notation of globally determined commit alphabet.

Note that we have associated final states only with systems rather than with individual automata. This is natural in the assumption - commitment framework: local reachability of any state for any automaton is dependent on the behaviour of other automata in the system, and hence properties of the system are best given in terms of global states. We might wish to partially specify global states with a "don't care" condition on some components in the system, but this is easily specified using a set of global states as above.

The global behaviour of $\widetilde{\mathcal{M}}$ is given below as that of the product automaton $\widehat{M}$ associated with the system. Note that the system is then a (finite state) machine over Σ, thus hiding away assumptions and commitments as internal details. This fits with the intuition that the behaviour of a distributed system is globally specified on Σ, and the machines in the system are programmed to achieve this, using coördination mechanisms like synchronization and commitments among themselves.

Definition 2.2 *Fix an ACS $\widetilde{\mathcal{M}}$ over $(\widetilde{\Sigma}, \widetilde{\mathcal{C}})$, where $\mathcal{M}_i = (Q_i, \longrightarrow_i, q_i^0, f_i)$. Let $Q = (Q_1 \times \ldots \times Q_n)$.*

1. *$(p_1, p_2, \cdots, p_n) \in Q$ is said to be* **compatible** *iff for all $i, j \in Loc$: $f_i(p_i)(j) \preceq_j f_j(p_j)(j)$. Let $\widehat{Q}$ denote the set of all compatible states in Q.*

2. *The product automaton associated with $\widetilde{\mathcal{M}}$ is given by: $\widehat{M} = (\widehat{Q}, \Longrightarrow, \bar{q}_{in}, F)$ where, $\bar{q}_{in} = (q_1^0, \cdots, q_n^0) \in \widehat{Q}$, $F \subseteq \widehat{Q}$ and $\Longrightarrow \subseteq \widehat{Q} \times \Sigma \times \widehat{Q}$ is defined by: $(p_1, \cdots, p_n) \stackrel{a}{\Longrightarrow} (q_1, \cdots, q_n)$ iff*
   - *(a) $\forall i \notin loc(a), p_i = q_i$, and*
   - *(b) $\forall i \in loc(a), p_i \stackrel{a}{\longrightarrow}_i q_i$.*

We extend the one step transition function $\Longrightarrow$ to words over Σ^*. Then the product accepts a string $x \in \Sigma^*$ if there is a state $\bar{q}$ in F such that $(q_1^0, \cdots, q_n^0) \stackrel{x}{\Longrightarrow} \bar{q}$ and the language accepted by $\widetilde{\mathcal{M}}$ is given as $\{x \in \Sigma^* \mid$ x is accepted by $\widehat{M}$ $\}$.

The class of languages over $\widetilde{\Sigma}$ accepted by ACS's is denoted as $\mathcal{L}(ACS_{\widetilde{\Sigma}})$. Formally, $\mathcal{L}(ACS_{\widetilde{\Sigma}}) = \{L \subseteq \Sigma^* \mid$ there is a commit alphabet $\widetilde{\mathcal{C}}$ and an ACS $\widetilde{\mathcal{M}}$ over $(\widetilde{\Sigma}, \widetilde{\mathcal{C}})$ such that $\widetilde{\mathcal{M}}$ accepts $L\}$.

Example: See Fig. 1 for an example ACS. Here, $\widetilde{\Sigma} = (\{a\}, \{b\})$ and $\widetilde{\mathcal{C}} = <(\mathcal{C}_1, \preceq_1), (\mathcal{C}_2, \preceq_2)>$ where the commit alphabet $\mathcal{C}$ is as shown in the figure. For ease of reference, we have annotated the local states by the respective $f_i(\cdot)$. We see that the only compatible states are those that are shown in the product.

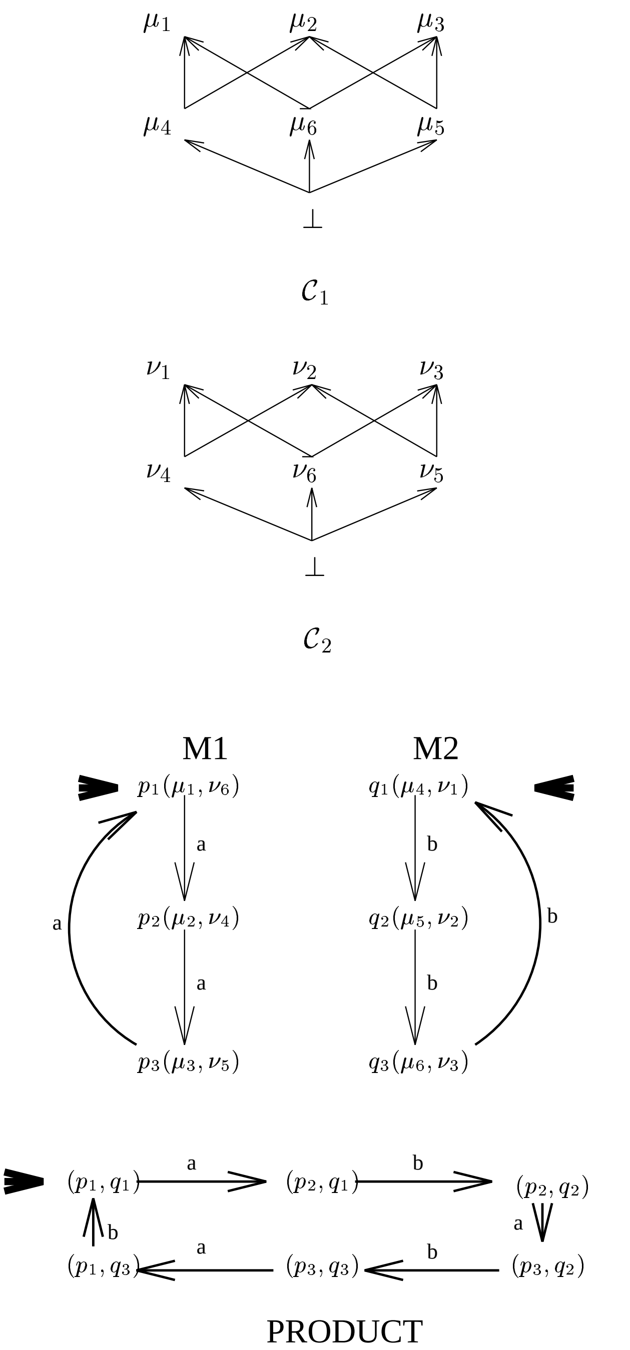
Note that the system accepts the regular language $(ab)^*$. Distributing this language amounts to a protocol where the 1-action a always takes precedence over the 2-action b. In the absence of explicit communication, disabling behaviours of the form $ba, abba$ etc. is the difficulty.

The following theorem constitutes the central result of the paper:

Theorem 2.3 $\mathcal{L}(ACS_{\widetilde{\Sigma}}) = REG_{\Sigma}$.

The inclusion $\mathcal{L}(ACS_{\widetilde{\Sigma}}) \subseteq REG_{\Sigma}$ is easy and follows from the fact that the product of ACS's are finite

Figure 1: An example ACS accepting $(ab)^*$ over the alphabet $(\{a\},\{b\})$.



state automata. The other inclusion, showing that every regular language over Σ is in $\mathcal{L}(ACS_{\tilde{\Sigma}})$, is (understandably) complicated. The proof is in two stages. First, we construct an automaton for the given regular language where the states are distributed but the transitions are globally specified. From this automaton, we compute the assumptions and commitments for the ACS and distribute the transitions as well so that the product of the ACS accepts the same language. Details are given in the Appendix.

3 An illustration

In this section, we show how the formal model of assumption compatible systems helps in reasoning about typical problems in distributed computing. For this we choose a very simple problem, namely, that of **reliable bit transmission**.

There are two processes, a *sender* S and a *receiver* R that communicate by asynchronous message passing over a possibly faulty channel. We assume that message loss in the channel is the only kind of fault in the system and that the number of such faults is finite(but unbounded) in any execution sequence.

The sender wants to send one bit(0 or 1) to the receiver. Since messages may get lost, there is no guarantee that a message sent by either of the processes will be received. The problem is to ensure that till R receives the bit, S has to go on sending the bit to R .

Finite state solutions for the above problem are simple. The main idea is to let R send back an acknowledgment when it receives the bit. When S gets the acknowledgment it stops sending the bit. We illustrate how the design can be done in an assumption commitment framework.

The sender has three states. At the initial state(called p_0), it is yet to send the bit and hence it knows that R cannot have received the bit. So it commits that it has not sent the bit and assumes that R is in a state where it has not received the bit. At the second state(called p_1) it has sent a bit and is waiting for acknowledgment. Now that the bit is sent, S does not know whether R has or has not received the bit and if R has received the bit then whether it has sent any acknowledgment or not. In short, it can assume "nothing" about the states of R but commits to having sent the bit and not having got any acknowledgment. At the third state(called p_2), it commits that it has got an acknowledgment from the receiver. Further this can happen only with the assumption that R has received the bit and has sent an acknowledgment.

The receiver also has three states. At the initial state(called q_0), it is yet to receive the bit, at the second state(called q_1) it has just received the bit and at the third state(called q_2) it has sent acknowledgment to S . At q_0, R commits (naturally) that it has not received the bit. Also, since it is yet to send back an acknowledgment, it assumes that S is in a state where it has not got any acknowledgment. Note that at q_0, the receiver can not assume anything about whether the sender has sent the bit or not. At q_1, R commits that it has received the bit and has not yet sent the acknowledgment. Clearly, the assumption is that S must already have sent the bit and that the latter has not yet got the acknowledgment. At q_2, R commits to having received the bit and sent an acknowledgment. While it can not assume that the sender has received this acknowledgment, R assumes that the bit has been sent(otherwise, R would not have sent any acknowledgment).

The assumptions and commitments of the sender and the receiver at their states can be summarized in the following table. The entries of the table are self-explanatory.

		Commitment
(S)	p_0	*bit_not_sent* and *ack_not_recd* by S
	p_1	*bit_sent* and *ack_not_recd* by S
	p_2	*bit_sent* and *ack_recd* by S
(R)	q_0	*bit_not_recd* and *ack_not_sent* by R
	q_1	*bit_recd* and *ack_not_sent* by R
	q_2	*bit_recd* and *ack_sent* by R

		Assumption
(S)	p_0	*bit_not_recd* by R
	p_1	R may be in any state.
	p_2	*bit_recd* and *ack_sent* by R
(R)	q_0	*ack_not_recd* by S
	q_1	*bit_sent* and *ack_not_recd* by S
	q_2	*bit_sent* by S

Note that this information can be naturally encoded by the ACS's we have just described. Fig. 3 gives a pictorial view such an ACS.

The actions of the processes are as follows; *send_bit*: S sends the bit to R , *rec_bit*: R receives the bit, *send_ack*: R sends the acknowledgment and *get_ack*: S gets the acknowledgment. We abbreviate the actions as s,r,a and g respectively. The natural distributed alphabet associated with the system is is $\widetilde{\Sigma} = (\Sigma_1, \Sigma_2)$ where, $\Sigma_1 = \{s, g\}$ and $\Sigma_2 = \{r, a\}$.

We encode the table of assumption and commitments in the ACS as follows. We have a commit alphabet $\mathcal{C} = < (\mathcal{C}_1, \preceq_1), (\mathcal{C}_2, \preceq_2) >$; $\mathcal{C}_1 = \mathcal{C}_1' \otimes \mathcal{C}_1'$ and $\mathcal{C}_2 = \mathcal{C}_2' \otimes \mathcal{C}_2'$ where $\otimes$ is the usual cartesian product of partial orders with $(\perp, \perp)$ identified with $\perp$. The assumptions and commitments associated with the states are exactly as given in the table above(we put the assumption of S at p_1 as $\perp$).

We see from the product automaton that with the final state $F = (p_2, q_2)$, the behaviour of the ACS is $s^+rs^*a(s+a)^*ga^*$, which captures the desired behaviour of the system, namely, S sends the bit till it receives an acknowledgment and then stops; R starts by receiving the bit and then goes on sending acknowledgments. Note how the global state (p_0, q_1)(which says R has received the bit before it has been sent) is ruled incompatible since the assumption at q_1 about S is *bit_sent* and *ack_not_recd*, the commitment at p_0 is *bit_not_sent* and *ack_not_recd* but *bit_sent* $\not\preceq_1$ *bit_not_sent*. Similar compatibility considerations rule out global states (p_2, q_0), (p_2, q_1) and (p_0, q_2).

4 Compatible shuffle and Kleene's theorem

A product operation on finite state automata corresponds to a shuffle operation on regular languages. Thus we can ask, what manner of shuffle gives us languages in $\mathcal{L}(ACS_{\widetilde{\Sigma}})$? We first answer this question below, and then present a distributed version of Kleene's theorem.

We first extend the distributed alphabet using commitments so that we have actions of the form $< a, \phi >$ where $\phi \in \Phi$. When $\phi(i) = \perp$ for any $i \in Loc$, we treat this as a "don't care" condition.

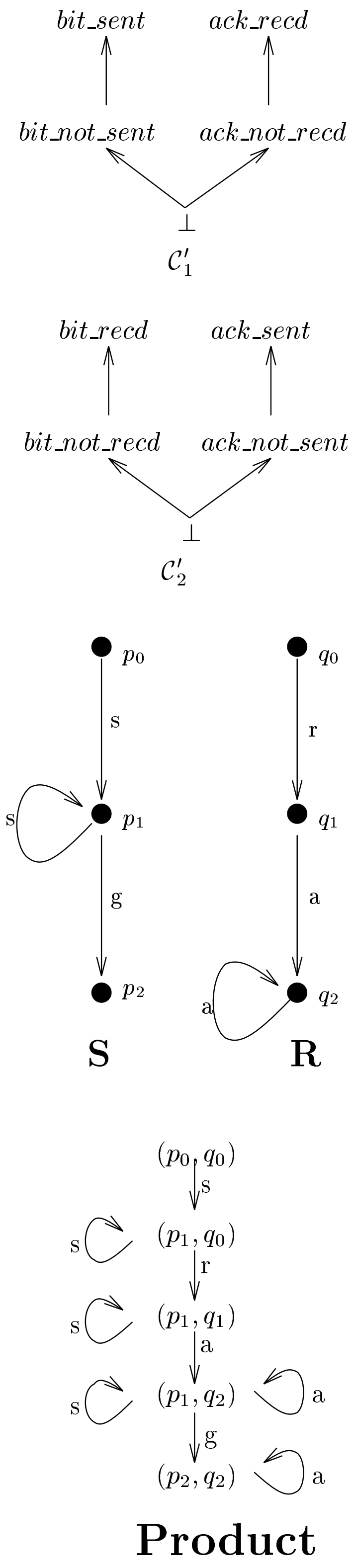
Definition 4.1 *Given a distributed alphabet, $\widetilde{\Sigma}$ and a commit alphabet $\widetilde{\mathcal{C}}$, we define extended alphabets as follows:* $\Sigma_i^c \stackrel{\text{def}}{=} \{< a, \phi > | a \in \Sigma_i,\ \phi \in \Phi\}$; $\widetilde{\Sigma^c} \stackrel{\text{def}}{=} (\Sigma_1^c, \cdots, \Sigma_n^c)$; $\Sigma^c \stackrel{\text{def}}{=} \bigcup_{i \in Loc} \Sigma_i^c$.

We extend loc to Σ^c as well: $loc(< a, \phi >) \stackrel{\text{def}}{=} \{i \mid a \in \Sigma_i\}$.

We will need two kinds of projection maps. The first is the *commit erasure* map: $\sigma : \Sigma^{c*} \to \Sigma^*$ defined as : $\sigma(< a_1, \phi_1 > \cdots < a_k, \phi_k >) \stackrel{\text{def}}{=} a_1 \cdots a_k$. The second is the *component projection* map: $\lceil$: $(\Sigma^{c*} \times Loc) \to \Sigma^{c*}$ defined by: $\epsilon \lceil i \stackrel{\text{def}}{=} \epsilon$ (where ϵ denotes the null string), and $(\lambda x)\lceil i \stackrel{\text{def}}{=} \lambda(x\lceil i)$, if $i \in loc(\lambda)$, and is $x\lceil i$, otherwise.

For this section, fix a distributed alphabet $\widetilde{\Sigma} = (\Sigma_1, \ldots, \Sigma_n)$, a commit alphabet $\widetilde{\mathcal{C}}$, and hence the associated extended alphabet $\widetilde{\Sigma^c}$ over Loc. When $x, y \in \Sigma_i^{c*}$, with $x = < a_1, \phi_1 >, \ldots, < a_k, \phi_k >$, and $y = < b_1, \psi_1 >, \ldots, < b_m, \psi_m >$, we say $x \sqsubseteq_i y$ iff $k = m$, for all $t, 1 \le t \le k$, $a_t = b_t$, $\phi_t(i) \preceq_i \psi_t(i)$ and for all $j \neq i$, $\psi_t(j) \preceq_j \phi_t(j)$. (This is because $a \in \Sigma_i$ implies that in $< a, \phi > \phi(i)$ gives i's commitment and $\phi(j)$ gives i's assumption about j.)

Figure 2: ACS for bit transmission



Let Ξ denote the set of all possible assumption - commitment tuples for all the agents in the system. Formally, $\Xi = \{\xi \mid \xi$ is a map from Loc to $\Phi\}$. For any $i, j \in Loc$, $\xi(i)$ is an n-tuple, and $\xi(i)(j) \in \mathcal{C}_j$ is i's assumption about j. We speak of ξ as an *assumption environment.*

Definition 4.2 *Let $\xi \in \Xi$. ξ is said to be* **feasible** *iff for all $i, j \in Loc$, $\xi(i)(j) \preceq_j \xi(j)(j)$.*

Let $x \in \Sigma^{c*}$ with $x = < a_1, \phi_1 >, \ldots, < a_k, \phi_k >$. and $\xi_0 \in \Xi$. The commitment structure *for x with respect to ξ_0 is a map com_{x,ξ_0}: $\{0, \ldots, m\} \rightarrow \Phi$: defined as follows (leaving the subscripts implicit): $com(0) = \xi_0$. Let $B = loc(a_{k+1})$.*
$com(k+1)(i, j) =$

$$\begin{cases} com(k)(i,j) & i \notin B. \\ \phi_{k+1}(j) \sqcup com(k)(i,j) & i, j \in B. \\ \sqcup_{l \in B} com(k)(l,j) & i \in B, j \notin B. \end{cases}$$

We say x is **good** *with respect to ξ_0 iff for all $k \in \{0, \ldots, m\}$, $com_{x,\xi_0}(k)$ is feasible.*

Let $x_i \in \Sigma_i^{c*}$, $i \in \{1, \ldots, n\}$, let $x \in \Sigma^*$, and $\xi_0 \in \Xi$. We say that x is **generated** *by the tuple $(x_1, x_2, \ldots, x_n)$ under ξ_0 iff there exists $y \in \Sigma^{c*}$ such that: y is good with respect to ξ_0, $\sigma(y) = x$ and for all $i \in Loc$, $y\lceil i \sqsubseteq_i x_i$.*

Let $L_i \subseteq \Sigma_i^{c*}$, $i \in \{1, \ldots, n\}$, and $\xi \in \Xi$. We define the n-ary **compatible shuffle** *of these languages under the assumption environment ξ by: $(\widehat{\|}_{i=1}^{n} L_i)_\xi \stackrel{\text{def}}{=} \{x \in \Sigma^* \mid x$ is generated by a tuple $(x_1, \ldots, x_n)$ under ξ, where for all i, $x_i \in L_i\}$.*

Let $\mathcal{L}(AC\text{-}Shuffle)_{\widetilde{\Sigma}}$ denote the least family of sets which includes the set $\{L \subseteq \Sigma^* \mid$ for some commit alphabet $\widetilde{\mathcal{C}}$, there exist $L_i \subseteq \Sigma_i^{c*}$ and $\xi \in \Xi$ such that $L = (\widehat{\|}_{i=1}^{n} L_i)_\xi\}$ and is closed under union. The following theorem asserts that languages in $\mathcal{L}(AC\text{-}Shuffle)$ are exactly the languages accepted by assumption compatible systems in $\mathcal{L}(ACS_{\widetilde{\Sigma}})$.

Theorem 4.3 $\mathcal{L}(AC\text{-}Shuffle_{\widetilde{\Sigma}}) = REG_\Sigma = \mathcal{L}(ACS_{\widetilde{\Sigma}})$.

Proof Idea: Consider $L \in \mathcal{L}(ACS_{\widetilde{\Sigma}})$. Let $L = L(\widetilde{\mathcal{M}})$, and let $L_i = L(\mathcal{M}_i, F_i)$ for $i \in \{1, \ldots, n\}$, where it suffices to consider $\widetilde{\mathcal{M}}$ such that $\widetilde{\mathcal{M}} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, F)$ and $F = \Pi_{i \in Loc} F_i$. Define $\xi_0 \in \Xi$ as follows: let $(q_1^0, \ldots, q_n^0)$ be the initial state of $\widetilde{\mathcal{M}}$; for all $i \in Loc$, $\xi_0(i) = f_i(q_i^0)$. We claim that $L(\widetilde{\mathcal{M}}) = (\widehat{\|} L_i)_{\xi_0}$. This places L in $\mathcal{L}(AC\text{-}Shuffle_{\widetilde{\Sigma}})$.

To show this in one direction, suppose that $x \in \Sigma^*$ is in $L(\widetilde{\mathcal{M}})$. Let $x = a_1 \dots a_k$. Consider an accepting path of $\widetilde{\mathcal{M}}$ on x, say $(q_1^0, \dots, q_n^0) \stackrel{a_1}{\Longrightarrow} \dots \stackrel{a_k}{\Longrightarrow} (q_1^k, \dots, q_n^k)$. Define $y = < a_1, \phi_1 > \dots < a_k, \phi_k > \in \Sigma^{c*}$ by: for $1 \le l \le k$, and $i \in Loc$, $\phi_l(i) = f_i(q_i^l)(i)$ if $i \in loc(a_l)$, and $\phi_l(i) = \perp$, otherwise. Then it can be checked that y is good with respect to ξ_0 and that $\sigma(y) = x$. Now consider $x_i = y\lceil i, i \in Loc$. It can be seen that $x_i \in L_i$ and that x is generated by the tuple $(x_1, \dots, x_n)$. Therefore $x \in (\widehat{\|} L_i)_{\xi_0}$.

To see the proof in the converse direction, observe that given $x_i \in L_i$, constructing a run for x in the product system depends crucially only on maintaining compatibility, and this follows from the fact that x is good with respect to ξ_0. Note that the definition of com_x ensures that the product system never gets stuck.

Suppose $L \in \mathcal{L}(AC\text{-}Shuffle_{\widetilde{\Sigma}})$. To show that $L = (\widehat{\|} L_i)_\xi$ is in $\mathcal{L}(ACS_{\widetilde{\Sigma}})$, by regularity of L_i, consider finite state machines $< \mathcal{M}_i, F_i >$ accepting L_i. We first obtain equivalent machines $\mathcal{M}_i'$ (whose transitions are Σ_i labelled) so that the initial states of the latter machines have no incoming transitions, and whenever $q_1 \stackrel{<a,\phi>}{\Longrightarrow} q$ and $q_2 \stackrel{<b,\psi>}{\Longrightarrow} q$, we have that $a = b$ and $\phi = \psi$. This can always be done with finite state automata by making sufficiently many copies of states; we need to to do this to define the assumption function for each i in such a way that the AC-system constructed accepts L. For the initial state, define $f_i(q_i^0) = \xi(i)$. When a transition in $\mathcal{M}_i'$ is of the form $q \stackrel{<a,\phi>}{\Longrightarrow} q'$, $f_i(q')$ is defined to be ϕ. We claim that $L = L(\widetilde{\mathcal{M}})$, where $\widetilde{\mathcal{M}}$ is the ACS constructed using the $\mathcal{M}_i'$'s. When $x \in L$, showing that it is accepted by $\widetilde{\mathcal{M}}$ mainly uses the fact that $com_{x,\xi}(j)$ is feasible for all j along x. On the other hand, from the construction it is easy to see that every accepting run of the product is over words which are good with respect to ξ, and which are generated by their projections to components.

We now consider the question of syntax for languages in $\mathcal{L}(ACS_{\widetilde{\Sigma}})$. Below, we write $L_1 \cdot L_2$ to denote language concatenation, and L^* for language (Kleene) iteration. The syntax is given in two layers, one for 'local' expressions and another for parallel composition. Fix a distributed alphabet $\widetilde{\Sigma}$, a commit alphabet $\widetilde{\mathcal{C}}$, and the associated extended alphabet.

$$AC_i ::= \emptyset \mid\ < a, \phi > \in \Sigma_i^c \mid p + q \mid p; q \mid p^*$$
$$AC ::= (\|_{i=0}^n r_i)_\xi, r_i \in AC_i,\ \xi \in \Xi \mid R_1 + R_2, R_i \in AC$$

The semantics of these expressions is given as follows: for $i \in Loc$, we have a map $[]_i : AC_i \to 2^{\Sigma_i^{c*}}$, and globally a map $[] : AC \to 2^{\Sigma^*}$. These maps are defined by structural induction:

- $[\emptyset]_i = \emptyset$.
- $[< a, \phi >]_i = \{< a, \phi >\}$.
- $[p + q]_i = [p]_i \cup [q]_i$.
- $[p; q]_i = [p]_i \cdot [q]_i$.
- $[p^*]_i = ([p]_i)^*$.
- $[(r_1 \| \cdots \| r_n)_\xi] = ([r_1]_1 \widehat{\|} \cdots \widehat{\|} [r_n]_n)_\xi$.
- $[R_1 + R_2] = [R_1] \cup [R_2]$.

The class of regular languages generated by the AC expressions is denoted as $\mathcal{L}(AC)$. Formally, $\mathcal{L}(AC_{\widetilde{\Sigma}})$ $= \{L \subseteq \Sigma^* \mid$ for some commit alphabet $\widetilde{\mathcal{C}}$, there is an $R \in AC$ over $\widetilde{\Sigma^c}$ such that $L = [R]\}$. The following theorem asserts that every regular expression over Σ has an equivalent presentation in AC.

Theorem 4.4 $\mathcal{L}(ACS_{\widetilde{\Sigma}}) = \mathcal{L}(AC_{\widetilde{\Sigma}}) = REG_\Sigma$.

The proof of the theorem is quite straightforward: the fact that languages associated with AC expressions are regular follows from the fact that compatible shuffle of regular languages is regular and that the class $\mathcal{L}(AC\text{-}Shuffle)$ is closed under union. For the other direction, given a regular language L, we consider the AC-system $\widetilde{\mathcal{M}}$ with initial state $(q_1^0, \cdots, q_n^0)$ that accepts it (which exists, by the main theorem): L can be written as a union of languages $\bigcup_q L_q$, where q is a final state of $\widetilde{\mathcal{M}}$. Now, if $q = (q_1, \dots, q_n)$, consider $L_i = L(\mathcal{M}_i, q_i)$, where $\mathcal{M}_i$ are the component automata. We can show that $L = (\widehat{\|}_i L_i)_\xi$, where $\xi = (f_1(q_1^0), f_2(q_2^0), \cdots, f_n(q_n^0))$. Hence we can consider the regular expressions associated with the components, put them in parallel and get the required AC expression.

The syntax presented above is not entirely satisfactory, since every expression in the language necessarily involves assumptions and commitments. Typically we wish to make assumptions only at *some* control points rather than at all control points. Moreover, the ξ parameter in the parallel composition operator is awkward. As it turns out, these are not serious problems. Consider the modified syntax (where FAC stands for assumptions and commitments occurring 'free' within expressions):

$$FAC_i ::= \emptyset \mid a \in \Sigma_i \mid \phi \in \Phi \mid p+q \mid p;q \mid p^*$$
$$FAC ::= (\|_{i=0}^{n} r_i), r_i \in FAC_i, \mid R_1 + R_2, R_i \in FAC$$

The semantic mappings need to be changed now: $[]_i$ now maps FAC_i to $2^{(\Sigma_i \cup \Phi)^*}$. Moreover, we need to define a notion of compatible shuffle of languages, where the strings have both actions from Σ as well as elements of Φ. This can be done on the same lines as we carried out above, by 'computing' the commitment structure *com*. It should be clear that we do not need an environment parameter ξ now, as we can replace any i-expression r by $\xi(i);r$, if required.

Let $FAC_{\widetilde{\Sigma}}$ denote the class of expressions in the new syntax. We can show the following:

Theorem 4.5 $\mathcal{L}(FAC_{\widetilde{\Sigma}}) = \mathcal{L}(AC_{\widetilde{\Sigma}}) = REG_\Sigma$.

We prove this by showing that $\mathcal{L}(FAC)_{\widetilde{\Sigma}} = REG_\Sigma$, in a manner similar to that described above, and then appealing to Theorem 4.4. We do not have a direct translation of programs in FAC into AC.

5 Discussion

We have shown that networks of finite state machines making assumptions about each other capture all finite state behaviours. A reasonable question is whether we can obtain interesting subclasses by placing restrictions on the structure of assumptions. While we do not have space to go into details here, we can show the following: (Fix $\widetilde{\Sigma}$.)

- Suppose for all local states s of the i^{th} automaton, $f_i(s)(j) = \perp$, for all $j \neq i$. This subclass of AC-systems captures the subclass of regular languages obtained by boolean closure of product languages over $\widetilde{\Sigma}$.

- Suppose the product construction is further constrained by the following condition: whenever $(s_1, \ldots, s_n) \stackrel{a}{\Longrightarrow} (t_1, \ldots, t_n)$ in the product, then for all $i \in loc(a)$, for all $j \in Loc$, $f_i(t_i)(j) = \sqcup_j \{f_k(s_k)(j) \mid k \in loc(a)\}$. Then this subclass exactly captures the class of *regular trace languages* over $\widetilde{\Sigma}$. Alternatively, instead of assumptions on local states, we can label i-transitions by (a, ϕ) where ϕ maps $loc(a)$ to $C_a = \bigcup_{j \in loc(a)} C_j$. Then the product construction ensures that during an a-transition, all automata participating in a have compatible ϕ's. The latter approach is the one followed in [10], another way of obtaining regular trace languages over $\widetilde{\Sigma}$.

- Consider an AC-system where assumptions are *monotone*: that is, whenever $s \stackrel{a}{\rightarrow}_i s'$, then for all j, $f_i(s)(j) \preceq_j f_i(s')(j)$. We can show that the class of languages accepted is a proper subclass of REG_Σ, and incomparable with the class of regular trace languages over $\widetilde{\Sigma}$.

These preliminary results suggest that exploring the assumption - commitment structure of finite state automata may give a rich automata theory of concurrency, but this is only a promise waiting to be fulfilled at this stage.

For reasons of brevity, we have not explored infinite behaviours, but would like to remark that the results obtained generalize smoothly. We merely give the relevant formal definitions below. Fix a distributed alphabet $\widetilde{\Sigma} = (\Sigma_1, \ldots, \Sigma_n)$, a commit alphabet $\widetilde{\mathcal{C}} = (\mathcal{C}_1, \cdots, \mathcal{C}_n)$ and the associated extended alphabet $\widetilde{\Sigma^c}$. An ω**AC-system** is a tuple $\mathcal{M} = (<\mathcal{M}_1, \mathcal{M}_2, \cdots, \mathcal{M}_n>, \mathcal{F})$, where $\mathcal{M}_i$ are AC-automata over Σ_i^c, $i \in Loc$, and $\mathcal{F} \subseteq 2^Q$ where $Q = \Pi_{i \in Loc} Q_i$.

Let $\widehat{M}$ be the product of the local AC-machines $\mathcal{M}_i$. The behaviour of the system $\widetilde{\mathcal{M}}$, denoted as $L^\omega(\widetilde{\mathcal{M}})$ is defined as the subset of Σ^ω accepted by the product $\widehat{M}$ with the acceptance table $\mathcal{F}$. The automaton $\widehat{M}$ accepts a string $x = a_0 a_1 \ldots \in \Sigma^\omega$, if there exists an infinite run $\rho = s^0 \stackrel{a_0}{\rightarrow} s^1 \stackrel{a_1}{\rightarrow} \ldots$ of the product system, and $U \in \mathcal{F}$ such that for all $i \in Loc$, $Inf(\rho) = U$ where $Inf(\rho) = \{q \mid \exists^\infty j,\ s^j = q\}$. Then, $L^\omega(\widetilde{\mathcal{M}}) = \{x \in \Sigma^\omega \mid \widehat{M} \text{ accepts } x\}$.

Thus, $\mathcal{L}(\omega ACM_{\widetilde{\Sigma}}) = \{L \subseteq \Sigma^\omega \mid$ there exists a commit alphabet $\widetilde{\mathcal{C}}$ and an ωAC-system $\widetilde{\mathcal{M}}$ over $\widetilde{\Sigma^c}$ such that $L = L^\omega(\widetilde{\mathcal{M}})\}$. We can then show that $\mathcal{L}(\omega ACM_{\widetilde{\Sigma}})$ is exactly the class of ω-regular languages over Σ.

References

[1] Alur, R., and Henzinger, T., "Local liveness for compositional modelling of fair reactive systems", *LNCS 939*, 1995, 166-179.

[2] Cori, R., Metivier, Y. and Zielonka, W., "Asynchronous mappings and asynchronous cellular automata", *Information and Computation*, vol 106, 1993, 159-202.

[3] de Roever, W.P., *Proceedings of COMPOS'97: 'Compositionality – the significant difference'*, Bad Malente-Gremsmühlen, Germany, 1997.

[4] Diekert, V. and Rozenberg, G., *The book of traces*, World Scientific Press, 1995.

[5] Fagin, R., Halpern, J., Moses, Y. and Vardi, M., *Reasoning about knowledge*, M.I.T. Press, 1995.

[6] Francez, N., and Pnueli, A., "A proof method for cyclic programs", *Acta Inf.*, vol 9, 1978, 138-158.

[7] Hoare, C.A.R., *Communicating Sequential processes*, Prentice Hall, 1985.

[8] Jones, C.B., "Specification and design of (parallel) programs", *Proc IFIP 83*, 1983, 321-331.

[9] Misra, J., and Chandy, M., "Proofs of networks of processes", *IEEE Trans. on Soft. Engg.*, vol 7, 1981, 417-426.

[10] Mohalik, S., and Ramanujam, R., "Assumption - Commitment in automata", *Proc FST & TCS 17*, LNCS 1346, 1997, 153-168.

[11] Pandya, P.K., and Joseph, M., "P-A logic: a compositional proof system for distributed programs", *Distributed Computing*, vol 5, 1991, 37-54.

[12] Vardi, M., "Verification of open systems", *Proc FST & TCS 17*, LNCS 1346, 1997, 250-266.

[13] Zielonka, W., "Notes on finite asynchronous automata", RAIRO-Inf. Theor. et Appli., vol. 21, 1987, 99-135.

[14] Zwiers, J., *Compositionality, concurrency and partial correctness*, LNCS 321, 1989.

Appendix

Here we give a sketch of the proof of Theorem 2.3.

Definition 5.1 *A Distributed State Automaton (abbreviated DSA) on $\widetilde{\Sigma}$ is a tuple $\mathcal{A} = (A_1, \ldots, A_n, \Delta, F)$, where*

1. *for every $i \in Loc, A_i = (Q_i, \Delta_i, q_i^0)$ is the i-th TS on Σ_i.*
2. *$Q = \Pi_{i \in Loc} Q_i$ is the state space of $\mathcal{A}$ and $F \subseteq Q$ is the set of final states, and*
3. *$\Delta \subseteq Q \times \Sigma \times Q$.*

The behaviour of $\mathcal{A}$ is given by the product automaton $(\widehat{\mathcal{A}}, F)$ where $\widehat{\mathcal{A}} = (Q, \Delta, (q_1^0, \ldots, q_n^0))$ satisfies the following *asynchrony* condition: if $((p_1, p_2, \ldots, p_n), a, (q_1, q_2, \ldots, q_n)) \in \Delta$ then for all $i \in loc(a), (p_i, a, q_i) \in \Delta_i$ and for all $j \notin loc(a), p_j = q_j$.

When $(\overline{p}, a, \overline{q}) \in \Delta$ we write it as $\overline{p} \overset{a}{\Rightarrow}_{\mathcal{A}} \overline{q}$. If $p = (p_1, \cdots, p_n) \in Q$ then $p[i]$ denotes the i-th local state p_i. $\mathcal{A}$ is *deterministic* if Δ is a function.

Fix a deterministic DSA $\mathcal{A}$ over $\widetilde{\Sigma}$. If $x\lceil i = \epsilon$, set $x \downarrow i = \epsilon$. Otherwise, $x \downarrow i \overset{\text{def}}{=} y$ where, for some $u \in \Sigma^*$ such that $x = yu$, $last(y) \in \Sigma_i$ and $u\lceil i = \epsilon$. $x \downarrow i$ gives the maximal prefix of x ending in an i-action. For example, let $\widetilde{\Sigma} = \{\{a, c\}, \{b, c\}\}$. Then $ab \downarrow 1 = a$, $abcaa \downarrow 2 = abc$ and $abcaac \downarrow 1 = abcaac$.

Definition 5.2 *For all $x \in \Sigma^*$, $\overline{x} \overset{\text{def}}{=} (q_1, \cdots, q_n) \in Q$ such that $(s_1^0, \cdots, s_n^0) \overset{x}{\Rightarrow}_{\mathcal{A}} (q_1, \cdots, q_n)$. Since $\mathcal{A}$ is deterministic, $\overline{x}$ is well-defined.*

Definition 5.3 *The* end-*transitions associated with strings over Σ are defined as follows: $S(\epsilon) \overset{\text{def}}{=} (\overline{\epsilon}, \epsilon, \overline{\epsilon})$ and $S(ua) \overset{\text{def}}{=} (\overline{u}, a, \overline{ua})$.*

$S(u)$ i-$\triangleright$-j $S(v)$ $\overset{\text{def}}{=}$ there is an $r \in \Sigma^*$ such that $r \downarrow i$ is a prefix of $r \downarrow j$, $S(u) = S(r \downarrow i)$ and $S(v) = S(r \downarrow j)$.

We note certain properties of DSA's in terms of the definitions above. An immediate corollary of the asynchrony condition on $\mathcal{A}$ is the following.

Proposition 5.4 *For all $x \in \Sigma^*$, $\overline{x \downarrow i}[i] = \overline{x}[i]$.*

Proposition 5.5 *Let $i, j \in loc(a)$. $S(xa)$ i-$\triangleright$-j $S(ya)$ implies $\overline{x} = \overline{y}$.*

Definition 5.6 *A DSA $\mathcal{A}$ is* **stutter-free** *if $(\overline{p} \overset{a}{\rightarrow} \overline{q}$ implies (for all $i \in loc(a), \overline{p}[i] \neq \overline{q}[i]$)).*

(Informally, this means that in $\mathcal{A}$, consecutive i-states are different).

Proposition 5.7 *Suppose $\mathcal{A}$ is stutter-free. Let $u, w \in \Sigma^*$, $a \in \Sigma_i$ and $k \in Loc$. Then it is never the case that both $S(ua)$ i-$\triangleright$-k $S(w)$ and $S(u \downarrow i)$ i-$\triangleright$-k $S(w)$,*

Definition 5.8 *We say $\mathcal{A}$ is* **four-alternation-free** *if it satisfies the following condition: let $r \in \Sigma^*$ be such that $r \downarrow i = r_1 a, r_1 \downarrow k = r_2 b, r_2 \downarrow i = r_3 c, r_3 \downarrow k = r_4 d, r_4 \downarrow i = r_5$. Further, let $a, c \notin \Sigma_k$ and $b, d \notin \Sigma_i$. Then $\overline{r_5} \neq \overline{r}$.*

Proposition 5.9 *Suppose $\mathcal{A}$ is four-alternation-free. Let $u, y \in \Sigma^*$, $a \in \Sigma$ and $i, k \in Loc$. The, ($S(ua)$ i-$\triangleright$-k $S(y)$ and $S(y)$ k-$\triangleright$-i $S(u \downarrow i)$) implies $\overline{y}[k] = \overline{u}[k]$.*

The following lemma establishes a correspondence between DSA's and ACS's over $\widetilde{\Sigma}$.

Theorem 5.10 *Let $\mathcal{A}$ be a stutter-free and four-alternation-free deterministic DSA on $\widetilde{\Sigma}$. Then, there exists an ACS $\widetilde{\mathcal{M}}$ on $\widetilde{\Sigma}$ such that $L(\mathcal{A}) = L(\widetilde{\mathcal{M}})$.*

From the given DSA $\mathcal{A}$, we construct an ACS $\widetilde{\mathcal{M}}$ $= (\mathcal{M}_1, \cdots, \mathcal{M}_n, < f_1, \cdots, f_n >, F)$ as follows.

For each $i \in Loc$, let the i-local TS be $\mathcal{M}_i = (Q_i, \rightarrow_i, q_i^0)$ where
$Q_i = \{S(x) | x \in \Sigma^*, x \downarrow i = x\}$,
$q_i^0 = S(\epsilon)$, and
$p \xrightarrow{a}_i q$ iff there is a $u \in \Sigma^*$ such that $p = S(u \downarrow i)$ and $q = S(ua)$.

The set of final states is defined to be $F = \{(S(x \downarrow 1), \ldots, S(x \downarrow n)) | x \in L(\mathcal{A})\}$.

The commit alphabet $\mathcal{C}$ is defined as follows: for all $i \in Loc$, $\mathcal{C}_i = 2^{Q_i}$ and $\lambda \preceq_i \mu$ iff $\lambda \supseteq \mu$. Local assumption maps f_i are defined as follows. For every $q \in Q_i$, $f_i(q)(j) = \{S(x \downarrow j) | S(x \downarrow i) = q\}$.

The following proposition gives an equivalent and more useful definition of the assumption maps.

Proposition 5.11 *For all $i, j \in Loc$ and $q \in Q_i$,*

$$f_i(q)(j) = \{p \in Q_j | \text{ either } p \text{ } j\text{-}\triangleright\text{-}i \text{ } q \text{ or } q \text{ } i\text{-}\triangleright\text{-}j \text{ } p\}.$$

Let $(\widehat{M} = (\widehat{Q}, \rightarrow, (q_1^0, \cdots, q_n^0)), F)$ be the product automaton of $\widetilde{\mathcal{M}}$ where $\widehat{Q}$ is the set of all compatible states. We observe a few properties of the product in the following.

Observation 5.12 **1:** *For all $x \in \Sigma^*$, $(S(x \downarrow 1), \ldots, S(x \downarrow n))$ is compatible.*

2: *For all $x \in \Sigma^*$, $(S(x \downarrow 1), \ldots, S(x \downarrow n)) \stackrel{a}{\Longrightarrow} (S(xa \downarrow 1), \ldots, S(xa \downarrow n))$.*

In the light of the above, by a simple induction on the length of strings we get for all $x \in \Sigma^*$, $(S(\epsilon), \ldots, S(\epsilon)) \stackrel{x}{\Longrightarrow} (S(x \downarrow 1), \ldots, S(x \downarrow n))$.

These observations immediately give us the following lemma proving one direction of the Theorem 5.10

Lemma 5.13 $L(\mathcal{A}) \subseteq L(\widetilde{\mathcal{M}})$.

We now prove the more difficult direction.

Lemma 5.14 $L(\widetilde{\mathcal{M}}) \subseteq L(\mathcal{A})$.

Proof: Let $x \in L(\widetilde{\mathcal{M}})$. Then there is a $u \in L(\mathcal{A})$, such that there is a path $(S(\epsilon), \ldots, S(\epsilon)) \stackrel{x}{\Longrightarrow} (S(u \downarrow 1), \ldots, S(u \downarrow n))$.

Claim 5.15 *Let $x \in \Sigma^*$. If $(S(\epsilon), \ldots, S(\epsilon)) \stackrel{x}{\Longrightarrow} (S(z_1), \ldots, S(z_n))$, then for all $i \in Loc$, $S(z_i) = S(x \downarrow i)$.*

Assuming the claim, for all $i \in Loc, S(x \downarrow i) = S(u \downarrow i)$. This implies for all $i \in Loc$, $\overline{x \downarrow i} = \overline{u \downarrow i}$, and hence $\overline{x}[i] = \overline{u}[i]$. So $\overline{x} = \overline{u}$. This means both x and u lead to the same state in $\mathcal{A}$. Since $u \in L(\mathcal{A})$, and $\mathcal{A}$ is deterministic, $\overline{u} \in F_{\mathcal{A}}$. Therefore, $\overline{x} \in F_{\mathcal{A}}$ which means $x \in L(\mathcal{A})$.

Proof of claim: The proof is by induction on $|x|$. The base case is trivial. Assume that the claim holds for all strings of length k. Take $x = ya$.

Suppose $(S(\epsilon), \ldots, S(\epsilon)) \stackrel{ya}{\Longrightarrow} (S(z_1), \ldots, S(z_n))$. Then there is a state $(S(y_1), \ldots, S(y_n)) \in \widehat{Q}$ such that

$$(S(\epsilon), \ldots, S(\epsilon)) \stackrel{y}{\Longrightarrow} (S(y_1), \ldots, S(y_n)), \text{ and}$$
$$(S(y_1), \ldots, S(y_n)) \stackrel{a}{\rightarrow} (S(z_1), \ldots, S(z_n)).$$

By induction hypothesis

$$S(y_i) \quad = \quad S(y \downarrow i) \text{ for all } i \in Loc \,. \tag{1}$$

We have to show that $S(z_i) = S(ya \downarrow i)$ for all $i \in Loc$.

Case 1: $i \notin loc(a)$. Then, $S(z_i) = S(y \downarrow i)$, by asynchrony and induction hypothesis. Since $S(y \downarrow i) = S(ya \downarrow i)$ for $i \notin loc(a)$, we are done.

Case 2. $i \in loc(a)$. Since $ya \downarrow i = ya$, we have to show $S(z_i) = S(ya)$.

By asynchrony, for all $j \in loc(a)$, $S(y_i) \xrightarrow{a}_i S(z_i)$. Hence, from the construction of $\rightarrow_i$, we have for all $j \in loc(a)$ a $u_j \in \Sigma^*$ such that

$$S(y \downarrow j) \quad = \quad S(u_j \downarrow j), \text{and} \tag{2}$$
$$S(u_j a) \quad = \quad S(z_j). \tag{3}$$

So it suffices to show that $S(u_i a) = S(ya)$. From the definition of S, it suffices to show that $\overline{u_i} = \overline{y}$ or for all $k \in Loc, \overline{u_i}[k] = \overline{y}[k]$. We do this separately for two cases: $k \notin loc(a)$ and $k \in loc(a)$.

Note that, by Equation 2,

$$\text{for all } m \in loc(a), \overline{u_m}[m] = \overline{y}[m]. \tag{4}$$

Let $k \in loc(a)$. By compatibility, we have $S(z_i) \in f_k(S(z_k))(i)$. Hence, $S(u_i a) \in f_k(S(u_k a))(i)$. By Prop. 5.11 either $S(u_i a)$ i-$\triangleright$-k $S(u_k a)$ or $S(u_k a)$ k-$\triangleright$-i $S(u_i a)$. In both the cases, from Prop. 5.5, $\overline{u_i} = \overline{u_k}$. Hence $\overline{u_i}[k] = \overline{u_k}[k]$ and from Equation 4 $\overline{u_i}[k] = \overline{y}[k]$.

Let $k \notin loc(a)$. Then $S(z_k) = S(y_k)$. By compatibility, $S(z_i) = S(u_i a) \in f_k(S(z_k))(i)$ and $S(y_i) = S(u_i \downarrow i) \in f_k(S(z_k))(i)$. By Prop. 5.11 [either $S(z_i)$ i-$\triangleright$-k $S(z_k)$ or $S(z_k)$ k-$\triangleright$-i $S(z_i)$] and [either $S(y_i)$ i-$\triangleright$-k $S(z_k)$ or $S(z_k)$ k-$\triangleright$-i $S(y_i)$]. We consider the possible cases. Remember that we are considering the case when $i \in loc(a)$ and $k \notin loc(a)$. Hence, in the following, $S(z_k) = S(y_k)$ and $S(z_i) = S(u_i a)$. Recall also that from Equations 1 and 2, $S(y_i) = S(u_i \downarrow i)$.

Case $S(z_k)$ k-$\triangleright$-i **$S(z_i)$**: There is an $r \in \Sigma^*$ such that $r \downarrow i = r$, $S(z_i) = S(u_i a) = S(r)$ and $S(z_k) = S(y_k) =$

$S(r \downarrow k)$. Then $\overline{y}[k] = \overline{y_k}[k] = \overline{z_k}[k] = \overline{r \downarrow k}[k] = \overline{r}[k] = \overline{u_i a}[k] = \overline{u_i}[k]$.

Case $S(z_i)$ i-⊳-k $S(z_k)$ and $S(y_i)$ i-⊳-k $S(z_k)$:

Since we assume that $\mathcal{A}$ is *stutter-free*, by Prop. 5.7 this case is not possible.

Case $S(z_i)$ i-⊳-k $S(z_k)$ and $S(z_k)$ k-⊳-i $S(y_i)$:

Assume the claim. Since for the case under consideration, $S(z_i) = S(u_i a)$, $S(z_k) = S(y_k)$ and $S(y_i) = S(u_i \downarrow i)$, we have $\overline{u_i}[k] = \overline{y_k}[k] = \overline{y}[k]$.

Thus we prove that for all locations $k \in Loc$, $\overline{u_i}[k] = \overline{y}[k]$. Thereby we prove the claim and the lemma.

Lemmas 5.13 and 5.14 prove Theorem 5.10.

Because of Theorem 5.10, in order to show that ACS's over $\widetilde{\Sigma}$ characterize the class of all regular languages over Σ, it suffices to prove the following theorem.

Theorem 5.16 *Let the class of deterministic DSA's over $\widetilde{\Sigma}$ that are stutter-free and four-alternation-free be denoted as FDSA. Then $\mathcal{L}(FDSA_{\widetilde{\Sigma}}) = REG_{\Sigma}$.*

Let $L \subseteq \Sigma^*$ be regular. We construct a deterministic FDSA $\mathcal{A}$ such that $L(\mathcal{A}) = L$.

Definition 5.17 (A labeling scheme) *Let $l : \Sigma^* \rightarrow (Loc \times Loc \rightarrow \{0, 1, 2\})$ be a labeling function defined inductively as follows: $l(\epsilon) \stackrel{\text{def}}{=} C$ where $C(i,j) = 0$ for every $i, j \in Loc$. Let $l(x) = C$. Then $l(xa) \stackrel{\text{def}}{=} C'$ where $C'(i,j) =$*

$$\begin{cases} C(i,j) & \forall i \notin loc(a) \\ 0 & \forall i,j \in loc(a), \text{ and} \\ (C(j,i)+1) \ mod \ 3 & \forall i \in loc(a), j \notin loc(a). \end{cases}$$

Note that, by definition, $l(x)(i,i) = 0$ for all x and $i \in Loc$.

Definition 5.18 (i-parity) *Define i-parity : $\Sigma^* \rightarrow \{0,1\}$ as: for all $x \in \Sigma^*$, i-parity$(x) \stackrel{\text{def}}{=} |x\lceil i| \ mod \ 2$.*

Proposition 5.19 *1. For all $x \in \Sigma^*, i, j \in Loc$, $l(x)(i,j) = l(x \downarrow i)(i,j)$.*

2. i-parity$(x) = i$-parity$(x \downarrow i)$ for all $x \in \Sigma^*$ and $i \in Loc$.

Let L be a regular language over Σ. We define the following relations on Σ^*. It can be easily checked that these are equivalence relations of finite index.

Definition 5.20 *$x \equiv y$ iff $l(x) = l(y)$ and $x \backslash L = y \backslash L$, where $x \backslash L \stackrel{\text{def}}{=} \{y \in \Sigma^* \mid xy \in L\}$. For all $i \in Loc$, $x \equiv_i y$ iff $x \downarrow i \equiv y \downarrow i$.*

Lemma 5.21 *If for all $x, y \in \Sigma^*$ and for all $i \in Loc$ $x \equiv_i y$ then $x \backslash L = y \backslash L$.*

Definition 5.22 *For all $x, y \in \Sigma^*, i \in Loc, x \sim_i y$ iff $x \equiv_i y$ and i-parity$(x) = i$-parity(y).*

Proposition 5.23 *1. $[xa]_i = [x]_i$ for all $i \notin loc(a)$.*

2. For all $x, y \in \Sigma^*$, if for all $i \in Loc, x \sim_i y$ then $x \backslash L = y \backslash L$.

The local automata of the constructed DSA $\mathcal{A}$ are defined as $A_i = (Q_i, \Delta_i, q_i^0)$, where:
$Q_i = \{[x]_i | x \in \Sigma^*\}$,
$q_i^0 = ([\epsilon]_i, 0)$ and
$\Delta_i = \{([x]_i, a, [xa]_i) | x \in \Sigma^*\}$.

Finally, $\mathcal{A} = (A_1, A_2, \ldots, A_n, \Delta, F)$, where $\Delta = \{(([x]_1, \ldots, [x]_n), a, ([xa]_1, \ldots, [xa]_n)) | x \in \Sigma^*\}$ and $F = \{([x]_1, [x]_2, \ldots, [x]_n) | x \in L\}$.

We observe that by construction, Δ is deterministic, and $\mathcal{A}$ is stutter-free as well as four-alternation-free.

Lemma 5.24
$\forall x \in \Sigma^* : (([\epsilon]_1, 0), \ldots, ([\epsilon]_n, 0)) \stackrel{x}{\Rightarrow}_{\mathcal{A}} ([x]_1, \ldots, [x]_n)$.

From this lemma, it is easy to see that $L(\mathcal{A}) = L$, proving Theorem 5.16, and completing the proof of Theorem 2.3.