Distributed automata in an Assumption - Commitment framework *

Swarup Mohalik[1] and R. Ramanujam[2]

[1]Sasken Communications Technology Ltd
Bangalore, India.
E-mail: swarup@sasken.com

[2]The Institute of Mathematical Sciences
Chennai, India.
E-mail: jam@imsc.ernet.in

Abstract. We propose a class of finite state systems of synchronizing distributed processes, where processes make assumptions at local states about the state of other processes in the system. This constrains the global states of the system to those where assumptions made by a process about another are compatible with the commitments offered by the other at that state. We model examples like reliable bit transmission and sequence transmission protocols in this framework and discuss how assumption - commitment structure facilitates compositional design of such protocols. We prove a decomposition theorem which states that every protocol specified globally as a finite state system can be decomposed into such an assumption compatible system. We also present a syntactic characterization of this class using top level parallel composition.

1 The assumption - commitment framework

Compositionality is a desired criterion for verification methodologies, particularly for the development and analysis of large systems. The idea is to decompose a system into smaller subsystems; then the specification for the system is verified (with respect to its implementation) using only the specifications of the subsystems without referring to their internal structure. This idea is formalized in [Flo] where properties of a *sequential* program are derived from the properties of its atomic actions and in [Dij] which improves upon the former by hierarchical decomposition and verification of a given program.

Compositional verification of *parallel* programs, on the other hand, adds substantially many complications, mainly because of the complex interaction of independently executing entities. The first proof systems for parallel programs of the form $P_1 \parallel \cdots \parallel P_n$ were suggested by [OG] and [AFR]. In the former, the central idea was that of *interference freedom test* and in the latter it was *cooperation test*. Both these needed to probe into the body of the component programs

* The main results here were first reported in [MR2]. We thank the anonymous reviewers for detailed comments that helped to improve the presentation.

to verify these tests. In this sense, they were not compositional proof systems. [MC] provided the foundation for the much-studied *assumption - commitment* framework(AC-framework) for compositional verification. The main idea is to specify a system as a module such that if some assumptions about the external environment are satisfied then the module commits to some desired behaviour. When one has a number of such modules acting together, then each module is effectively in the environment created by the other modules. If this environment satisfies the assumptions then the module delivers the right behaviour. Thus for the desired behaviour of the global system, assumptions and commitments of the components must mutually satisfy each other. This facilitates compositional reasoning: we can reason about the behaviour of each component separately, assuming that others maintain relevant properties and reason globally about their compatibility.

In [MC], in order to capture the assumptions of the environment and commitments of the modules, one has predicates over *communication histories*. Communication histories encode the kind of interaction a module undergoes with the environment. Assumptions on communication histories are essentially constraints on the environment (in case of systems of modules running in parallel, they are constraints on the communication behaviour of other modules). But the framework itself is very general and can be applied in various ways to prove global system properties [AL1,AL2,BKP,Jon,PJ,QM].

In this paper, we consider a kind of reasoning for distributed systems that is somewhat different in spirit from the classical compositional reasoning mentioned above. In the latter, a component is looked upon as a black-box that maintains some invariant when the environment guarantees some properties. Then, one hopes to derive system properties from the compatibility of assumptions and commitments of components, without looking into internal structure.

This black-box approach works very well as long as we are composing safety properties of the system [AL1,AL2,AAF,MP,OL]. On the other hand, it is generally agreed that composing liveness is hard. This is because, in distributed systems, local enabling of actions does not ensure global enabling. It depends crucially on the internal structure of components in the system. Therefore, for liveness it looks as if one needs to *look into* the black-boxes and keep some global information as part of *local structures* (either states or local transitions). If one does this judiciously, then all necessary global information has been distributed so that global behaviour can be obtained by a product of component processes. Our thesis is that this global information can be distributed in the local structures by suitable assumptions about other processes and commitments. Thus, a process does not have to know the detailed structure of other processes in so far as it can make suitable assumptions about others, and rely on the protocol to ensure that in the global execution these assumptions are met by the commitments of other processes.

This kind of distribution through assumption and commitment is not novel. This happens routinely when one develops subsystems without having access to a global view of the system, for example, when different groups develop parts of a large program. For instance, suppose we are designing a *receiver* that receives a bit from a sender and processes it. Then independent of the sender, the receiver can be designed as follows:

⟨Assume there is a bit in the channel⟩ Receive the bit; Process the bit.

The internal actions of components change their state or the state of the environment. These effects can be said to be commitments of the components. Thus the components go on making assumptions about environments and make commitments as well. When such components are put together, one gets compatible behaviours where, as intended, the assumptions of a process are met by commitments of the other processes that constitute its environment. Consider the design of the sender too.

Send the bit; ⟨Commit that a bit is in the channel⟩

When we put the sender and receiver together, we expect the normal sequence of transfer of the bit. Notice that the receiver cannot receive the bit before it is sent because then its assumption about the bit in the channel can not be met. In this way, causal dependence can be *encoded* by assumptions.

(Sender) Send the bit; (Receiver)Receive the bit; (Receiver)Process the bit.

We call this way of reasoning *local reasoning* since each component reasons "locally" about the environment (other processes in the system) to make appropriate assumptions. Observe that this view is different from the compositionality principle since here one looks into the internal design of components. Our concern is to model one aspect of system design that occurs in many situations, notably in distributed algorithms and program development, as we tried to illustrate in the example above.

Note also that *local reasoning* in the sense described above is very different from what researchers call *modular reasoning*, employed in modular model checking [KV,Var]. In modular reasoning there is no constraint at all on the environment of the module. This makes perfect sense because one is concerned about design of modules as open systems, systems that can potentially be embedded in *any* environment. On the other hand, in local reasoning we are interested in closed systems where the environment of a process is the set of other processes in the system and the processes know the protocol of interaction. Thus, the environments here are very much constrained and the processes know a lot about the environment. Therefore, while modular reasoning is hard [KV,PL,Var], one can expect local reasoning to be simpler.

While a number of researchers seem to have studied the AC-framework in the context of programming methodology, process algebras or temporal logics, there seems to have been little effort in formulating it from an automata-theoretic viewpoint. Implicitly, these models assume each process to be a machine of some sort, but studying formally the implications of each process being a finite-state machine is a different exercise altogether. There have been efforts in modular model checking [KV,Var], but the kind of complex interaction and compatibility that is reflected in the behaviour of parallel systems is not quite transparent.
Why should one look for an automata theoretic account of the AC-framework? An important reason is that these automata can serve as natural models for temporal logics based on local reasoning (for instance the one in [Ram3]). Compositional model checking is one of the major goals of computer-aided verification [AH], and we believe that local reasoning with automata as the component processes (particularly over infinite words) may help.
In order to "locally reason" about the process model, we attach assumptions and commitments to local states and stipulate that only those global states of the system are valid where assumptions and commitments of local states are mutually compatible. A valid behaviour of the system is determined only by compatible global states. In the next section we show that such modelling is natural, by describing systems like the sequence transmission protocol. We then go on to formally define the class of automata and show that the modelling is expressive enough to capture *all regular behaviours*; that is, every (globally specified) finite state system can be decomposed into an assumption compatible system suitably. We refine this observation further to show that the decomposition can be carried out at a *syntactic* level as well. We close the presentation with a discussion.
The paper presents only automata-theoretic results on the assumption compatible systems defined here. While the definition of the model is clearly inspired by compositional verification, this paper does not address the related logical issues, except for a minor discussion.

2 System examples

In order to express assumptions and commitments at local states of processes, alongwith a distributed alphabet of actions in the system, we have an alphabet we call a *commit alphabet*. It is a tuple $\mathcal{C} = ((\mathcal{C}_1, \preceq_1), \cdots, (\mathcal{C}_n, \preceq_n))$. where $\mathcal{C}_i$'s are nonempty alphabets called local commit alphabets and $\preceq_i$ is an order on $\mathcal{C}_i$. The intention is to have boolean formulas over some finite set of propositions as commitment alphabet and logical implication as the partial order $P \preceq_i Q$ iff $Q \Rightarrow P$. Thus we can think of a commitment at a state as saying which propositions hold at that state. Moreover, suppose process i has an assumption c_j about process j at a local state p_i. Since $c_j \in \mathcal{C}_j$ can be seen as a boolean formula, i assumes that j is in some state which satisfies c_j. If actually j is in such a state p_j, then it will be compatible with p_i; in other words, the commitments at p_j should logically imply c_j for compatibility. The commit alphabet is essentially an abstraction of this idea. We may also think of commit alphabets as denoting subsets of local states with compatibility defined by set-inclusion.

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.

Our formulation is partly inspired by *knowledge-based programs* [FHMV], where atomic statements of a process are of the form $K_i\varphi \rightarrow a$. These statements are called *knowledge statements* and they are read as "if i knows φ then execute a". The usual semantics of K_i is on Kripke models of global states: $K_i\varphi$ is true at a global state s if φ holds at all the global states i considers possible at s. However, for local reasoning, we need a notion of knowledge based on local states. In [Ram2] semantics of K_i is given at local states of i: $K_i\varphi$ is true at an i-local state p if φ holds at all the global states i considers possible at p. While the connection is intuitive, we do not have a formal result relating knowledge-based programs with the automata studied here.

There is one relevant observation regarding the commit alphabet to be made here. It is not necessary that the commit alphabet be fixed universally for the system, as we have done above. This is because different processes may have access to different variables of a process i, hence their assumptions about the states of i will vary from each other. We can define each process with its own n-tuple of assumption alphabets and subsequently ensure in the definition of systems that for all $i, j \in Loc$, the j^{th} assumption set of automaton i is contained in the j^{th} commit set of automaton j. But such fine structure plays no technical role and clutters up notation considerably. Hence, we stick with the (more restricted) notation of a globally determined commit alphabet.

A theoretically simple framework for assumption-commitment in automata is when processes make assumptions only about those other processes that they communicate with. This is naturally modelled by having assumption and commitment on synchronization transitions. Here an automaton A_1 may synchronize with another automaton A_2 on an action $(a, \lambda_1, \lambda_2), a \in \Sigma_1 \cap \Sigma_2$ where we see A_1 as committing to λ_1 provided A_2 commits to λ_2. Symmetrically, for such a synchronization to occur, A_2 must have a transition on a where it commits to λ_2. (In this case, A_2 may not require the λ_1 commitment from A_1.)

2.1 A mutual exclusion example

In order to motivate the kind of framework we are leading to, we give an example of a simple two-process mutual exclusion problem. For simplicity, we abstract away the internal computational states and assume that the processors are always requesting for or executing in the critical section. We model this as follows. Each process i can either be in state W_i (process i waiting to enter the critical section) or in state C_i (process i is in the critical section) state. In order to gain access to the critical section from the wait state, the processes do a joint action c. Actions a and b are actions taken by processes in the critical section. When an internal action is taken, the process in the critical section goes back to the waiting state.

We set the commitment alphabet $\mathcal{C} =< \mathcal{C}_1 = (\{p_1, np_1\}, \preceq_1), \mathcal{C}_2 = (\{p_2, np_2\}, \preceq_2) >$. Here, p_i, $i = 1, 2$ denotes that process i is permitted access to critical section and np_i denotes that it is not permitted to enter the critical section. The commit alphabet is shown in Fig. 1.

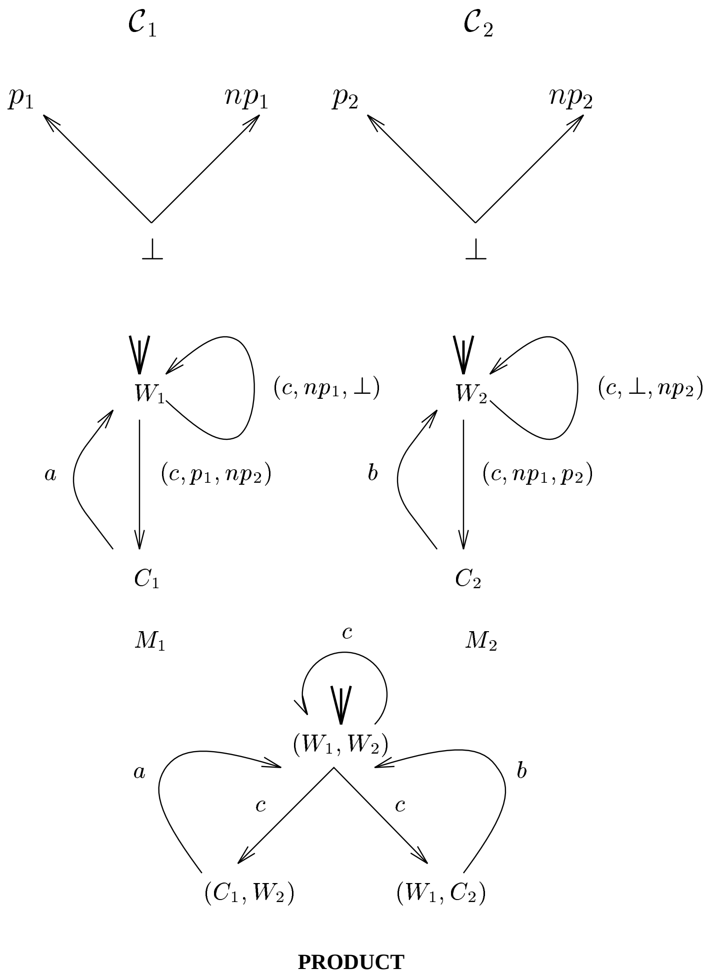


Fig. 1. Two-processor mutual exclusion.

The design of process 1 can then be as follows: when 1 is in the state W_1, it stays in the same state if it is not permitted entry to critical section. When it is permitted entry, assuming that process 2 is not permitted entry, it can go to the state C_1 denoting access to critical section. Process 2 is designed in a symmetric way. Figure 1 shows the two processes and also the product showing the global behaviour. See that the assumptions at the local transition $W_1 \xrightarrow{c}_1 C_1$ are (p_1, np_2) and those at $W_2 \xrightarrow{c}_2 C_2$ are (np_1, p_2). These assumptions are not compatible because $np_2 \not\preceq_2 p_2$ and $np_1 \not\preceq_1 p_1$. Hence the global transition $(W_1, W_2) \stackrel{c}{\Longrightarrow} (C_1, C_2)$ is not possible; so that at no point both the processes can be in the critical section, thus satisfying the safety requirement.
It is possible to formally define a class of automata where assumptions and commitments are associated with synchronizing transitions like in this example, along the lines followed in the next section. It then turns out that the class of languages accepted by such systems is the same as the so-called class of *regular consistent languages* ([MR1]). However, we will not do so, but define a larger class of automata from which these can be obtained by suitable restrictions.
In general, a process may make assumptions about other processes in the system even in the absence of any communication from them. This leads us to a type of systems where at a local state a process makes assumptions about the states in which other processes may be, and in the product only mutually compatible states are admissible. We call these systems Assumption-Compatible Systems(ACS).
We now show how the formal model of assumption compatible systems helps in reasoning about typical problems in distributed computing. For this we choose simplified versions of two well-known problems, namely, that of **reliable bit transmission** and **sequence transmission**.

2.2 Bit transmission problem

There are two processes, a *sender* S and a *receiver* R. Assume that they communicate by asynchronous message passing over a possibly faulty channel. Further, 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 (See Fig. 2). The alphabet Σ_1 of sender S is $\{s, g\}$, where

– s: S sends the bit to R , and

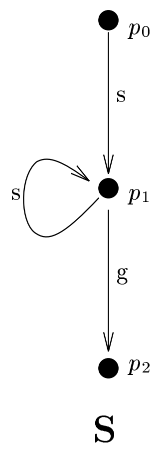


Fig. 2. The sender.

– g: S receives the acknowledgment.

S 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 received the bit or not 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 (See Fig. 3) The alphabet Σ_2 of sender R is $\{r, a\}$, where

– r: R receives the bit from S , and
– a: R sends the 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

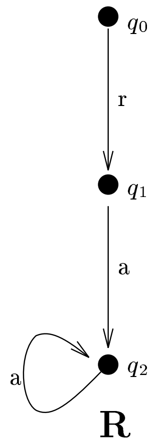


Fig. 3. The receiver.

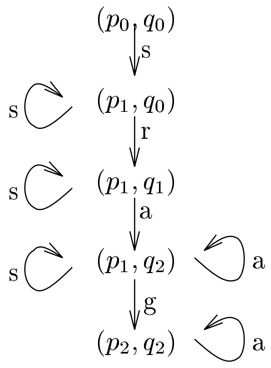
got the acknowledgment. At q_2, R commits to having received the bit and sent an acknowledgment. While it can not assume that the sender has received this acknowledgment, R assumes that the bit has been sent(otherwise, R would not have sent any acknowledgment).
The commitment alphabet of S can then be taken as $(\mathcal{C}_1, \preceq_1)$ where $\mathcal{C}_1$ is the boolean closure of the set of propositions $\{ack\_recd, bit\_sent\}$, where the meaning of the propositions is obvious. Similarly, the commitment alphabet of S can, then be taken as $(\mathcal{C}_2, \preceq_2)$ where $\mathcal{C}_2$ is the boolean closure of the set of propositions $\{bit\_rec, ack\_recd\}$, In both cases the orders $\preceq_i$ are logical implication.
With this the assumptions and commitments of the sender at its states can be summarized in the following table. ($\neg bit\_sent$ means "bit is not sent" etc.)

		Commitment	Assumption
$(S\ )$	p_0	$\neg bit\_sent$ and $\neg ack\_recd$ by S	$\neg bit\_recd$ by R
	p_1	bit_sent and $\neg ack\_recd$ by S	R may be in any state.
	p_2	bit_sent and ack_recd by S	bit_recd and ack_sent by R

Similarly, the assumptions and commitments of the sender at its states are given in the following table.

		Commitment	Assumption
$(R\ )$	q_0	$\neg bit\_recd$ and $\neg ack\_sent$ by R	$\neg ack\_recd$ by S
	q_1	bit_recd and ack_not_sent by R	bit_sent and $\neg ack\_recd$ by S
	q_2	bit_recd and ack_sent by R	bit_sent by S

Product automaton and global behaviour Fig. 4 gives the product automaton of the ACS consisting of S and R . Notice that 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 $\lambda = bit\_sent \wedge \neg ack\_recd$, the commitment at



Product

Fig. 4. ACS for bit transmission.

p_0 is $\mu = \neg \mathit{bit\_sent} \wedge \neg \mathit{ack\_recd}$ but $\mu \not\Rightarrow \lambda$. Similar compatibility considerations rule out global states (p_2, q_0), (p_2, q_1) and (p_0, q_2).
Then, 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.

2.3 Sequence transmission problem

We now study a slightly more involved protocol, namely the **Sequence Transmission Problem** and discuss how it can be modelled naturally in the assumption - commitment framework. The problem is as follows.
As before, there is a sender S and a receiver R . The sender now wants to send a bit-stream to the receiver. For each message bit, the bit transmission protocol is used. Essentially, S goes on sending the i-th message bit till it receives an acknowledgment and then it starts sending the $i+1$-th bit and so on. On the other hand, R initially waits till it gets the first bit. After this, for each bit (say it is the i-th bit), it goes on sending the acknowledgment till it receives the $i+1$-th bit. There are, of course, some other requirements of the problem that makes the design slightly harder.

- *Totality. All* the bits of the stream are delivered.
- *Sequentiality.* The bits are delivered in the order in which they occur in the stream.
- *Non-duplication.* A particular bit may be delivered many times over the channel because S might not have got any acknowledgment for the bit, but once R receives the bit and sends acknowledgment, it does not receive the same bit from the channel.

From the description above, one can design the protocol as a series of bit transmission protocols for each bit in the stream. For each bit, if we label the actions and also the assumptions and commitments with the position of the bit, we get an infinite state protocol which is given in Fig. 5 with the assumption and commitments from Fig. 6. (Note that there are now infinite number of elements of the commitment alphabet; in the table a state $p_1 i$ denotes the i-th p_1 state. In the figure it is given as the state p_1 with i quotes.)
The product shows that the requirements are actually satisfied, namely, for every $i \geq 0$, the action r_i (receive bit i) takes place (totality), it occurs only once (non-duplication) and r_j occurs strictly before r_k when $j < k$ (sequentiality).
Now, we want to *fold* this protocol so that the sender and receiver actually have finite number of states. Since the states are determined by their assumptions and commitments and the actions, this folding would have to bound the labels attached to the letters of the commitment alphabet and also the actions. As a first attempt, if we banish the labels altogether, we get a protocol as in Fig. 7. Immediately we see that this protocol does not satisfy the requirements.

- Since there is no distinction between consecutive messages, S might be sending the i-th message even after it gets an acknowledgment from R . But this is a minor difficulty because we can always ensure that S moves to the i + 1-th bit after it gets an acknowledgment for the i-th message. Hence, sequentiality is ensured.
- But one can see that in the product, between a *rec_bit* by R and *rec_ack* by S , there are other *rec_ack*'s, which means duplication takes place.
- Also, there is loss of message bits in this design for the following reason: S receives an *ack* and sends i-th message, R does not receive it and sends another *ack* corresponding to $(i - 1)$-th message, S receives this *ack* and assumes that this is an acknowledgment of the i-th message. Hence it then starts sending the $(i + 1)$-th message. Thus the message i is never received by R . The problem arises because when S sends message $(i+1)$, it assumes that R has already received message i, but then in the protocol assumptions and commitments do not reflect this.

In order to remedy this deficiency, we have one bit (0 or 1) attached to the actions of sending and receiving message and acknowledgments and also to the assumptions and commitments. This is essentially to distinguish between consecutive messages and acknowledgments. Thus, we get a protocol as in Fig. 9 with the assumptions and commitments as in the table in Fig. 8 and the product shows that this satisfies all the requirements of sequence transmission problem. This, in fact, is the **Alternating Bit Protocol(ABP)** for the sequence transmission problem.
The above analysis does not give any clue as to why only two distinguishing sets of states were sufficient for the sequence transmission problem. In fact it looks a bit like an accident that we hit upon the ABP. But, at the least, this guides us to have more states to capture relevant assumptions and to design the protocol correctly. While it is not argued here that this transformation presented here makes it easier to design the components (it does not), we remark that it

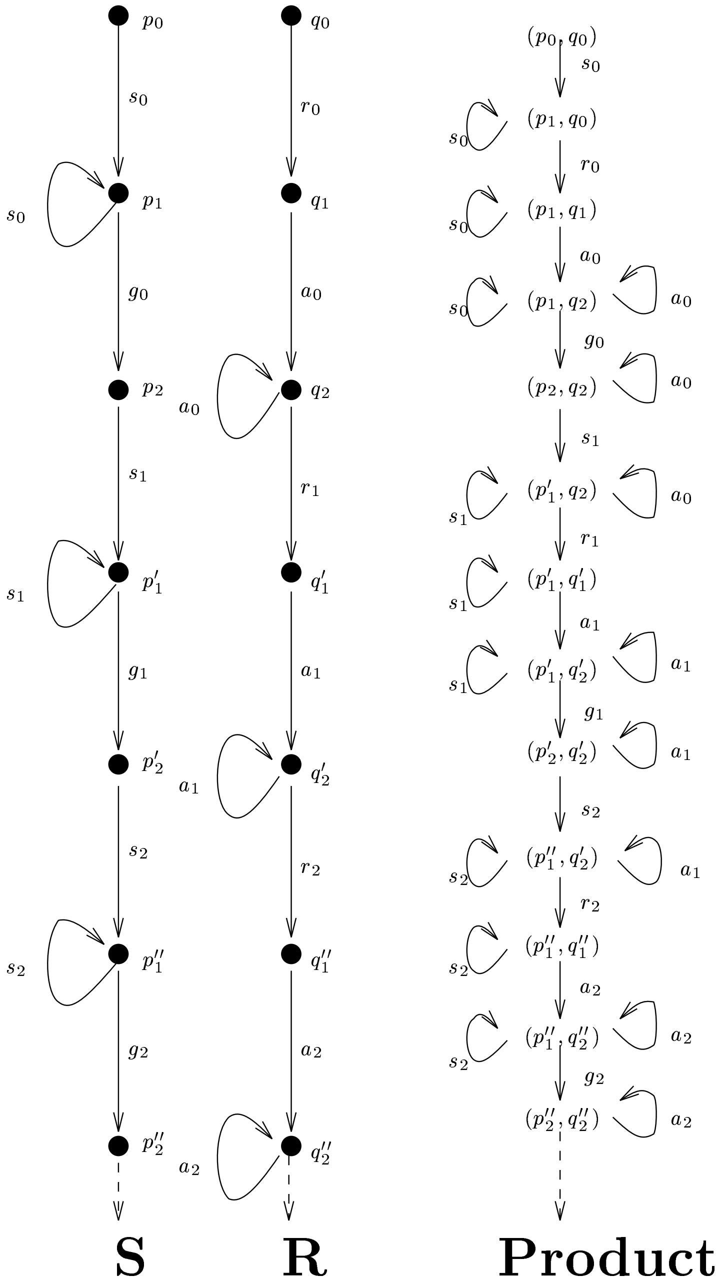


Fig. 5. An infinite state protocol for STP.

		Commitment	Assumption
(S)	p_0	$\neg bit_0\_sent$ and $\neg ack_0\_recd$ by S	$\neg bit_0\_recd$ by R
	$p_1 i$	bit_i_sent and $\neg ack_i\_recd$ by S	R may be in any state.
	$p_2 i$	bit_i_sent and ack_i_recd by S $\neg bit_{i+1}\_sent$ and $\neg ack_{i+1}\_recd$ by S	bit_i_recd and ack_i_sent by R $\neg bit_{i+1}\_recd$ by R
(R)	q_0	$\neg bit_0\_recd$ and $\neg ack_0\_sent$ by R	$\neg ack_0\_recd$ by S
	$q_1 i$	bit_i_recd and $ack_i\_not\_sent$ by R	bit_i_sent and $\neg ack_i\_recd$ by S
	$q_2 i$	bit_i_recd and ack_i_sent by R $\neg bit_{i+1}\_recd$ and $\neg ack_{i+1}\_sent$ by S	bit_i_sent by S $\neg bit_{i+1}\_recd$ by R

Fig. 6. Infinitely many assumptions and commitments for the infinite state protocol.

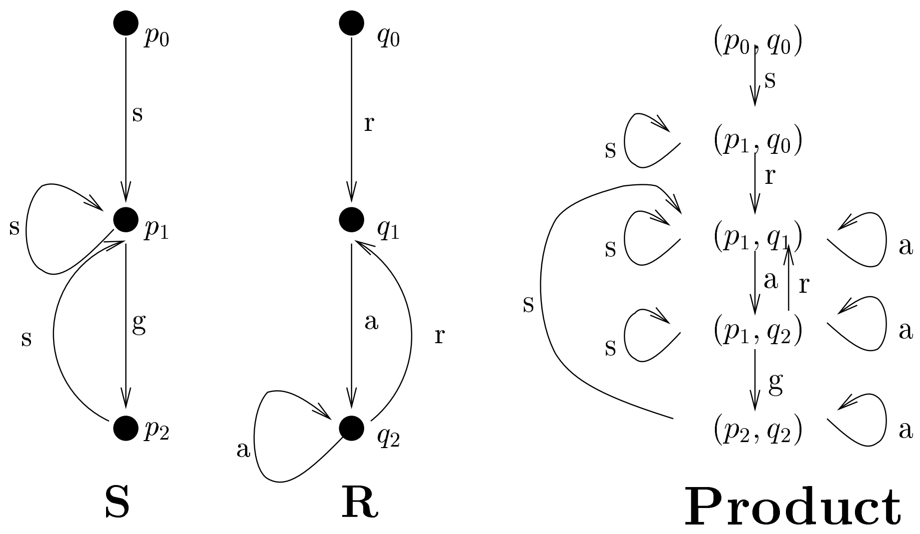


Fig. 7. An incorrect folding of the infinite state protocol for STP.

		Commitment	Assumption
(S)	p_0	$\neg bit_0\_sent$ and $\neg ack_0\_recd$ by S	$\neg bit_0\_recd$ by R
	p_1	bit_0_sent and $\neg ack_0\_recd$ by S	R may be in any state.
	p_2	bit_0_sent and ack_0_recd by S $\neg bit_1\_sent$ and $\neg ack_1\_recd$ by S	bit_0_recd and ack_0_sent by R $\neg bit_1\_recd$ by R
	p_1'	bit_1_sent and $\neg ack_1\_recd$ by S	R may be in any state.
	p_2'	bit_1_sent and ack_1_recd by S $\neg bit_0\_sent$ and $\neg ack_0\_recd$ by S	bit_1_recd and ack_1_sent by R $\neg bit_0\_recd$ by R
(R)	q_0	$\neg bit_0\_recd$ and $\neg ack_0\_sent$ by R	$\neg ack_0\_recd$ by S
	q_1	bit_0_recd and $\neg ack_0\_sent$ by R	bit_0_sent and $\neg ack_0\_recd$ by S
	q_2	bit_0_recd and ack_0_sent by R $\neg bit_1\_recd$ and $\neg ack_1\_sent$ by S	bit_0_sent by S $\neg bit_1\_recd$ by R
	q_1'	bit_1_recd and $\neg ack_1\_sent$ by R	bit_1_sent and $\neg ack_1\_recd$ by S
	q_2'	bit_1_recd and ack_1_sent by R $\neg bit_0\_recd$ and $\neg ack_0\_sent$ by S	bit_1_sent by S $\neg bit_0\_recd$ by R

Fig. 8. Assumptions and commitments for a correct finite state protocol of Fig. 4.10.

illustrates a paradigm: design the components loally assuming an infinite number of assumptions first and then fold it. The structure of ACS's offers the possibility of designing components 'bearing in mind' global requirements, and translating them into local dependencies. In general, this may involve the construction of an infinite state space (which may be conceptually easy) and then folding it to obtain a finite system.

3 Assumption-compatible systems

We now formally define the class of assumption compatible systems that we have discussed intuitively so far.

Notation: Let Σ be a finite and nonempty alphabet. A *transition system* (TS) over Σ is a tuple $M = (Q, \longrightarrow, q^0)$, where Q is a *finite* set of states, $\longrightarrow \subseteq (Q \times \Sigma \times Q)$ is the transition relation and $q^0 \in Q$ is the initial state. The tuple $A = (M, F)$, where M is a TS and $F \subseteq Q$, is called a *finite state automaton* (FA). When $q_0 \stackrel{x}{\Longrightarrow} q_k$ and $q_k \in F$, we say that the string x is accepted by A (where $\Longrightarrow$ is the transition relation extended to Σ^*). The set of all strings accepted by A (also called the language of A) is denoted as $L(A)$ or $L(M, F)$. Let Reg_Σ denote the class of languages $L(A)$ accepted by finite state automata.

We model the distribution of actions among the processes by a *distributed alphabet* $\widetilde{\Sigma}$. It is a tuple $(\Sigma_1, \cdots, \Sigma_n)$, where each Σ_i is a finite nonempty set of actions and is called a *local alphabet.* The local alphabets are not required to be disjoint. In fact, when $a \in \Sigma_i \cap \Sigma_j, i \neq j$, we think of it as a potential 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 since the overall behaviour of the system is expressed by strings from Σ^*. For any action in Σ, we have the notion of processes participating in this action. Let $loc : \Sigma \rightarrow 2^{\{1,\dots,n\}}$ be defined by

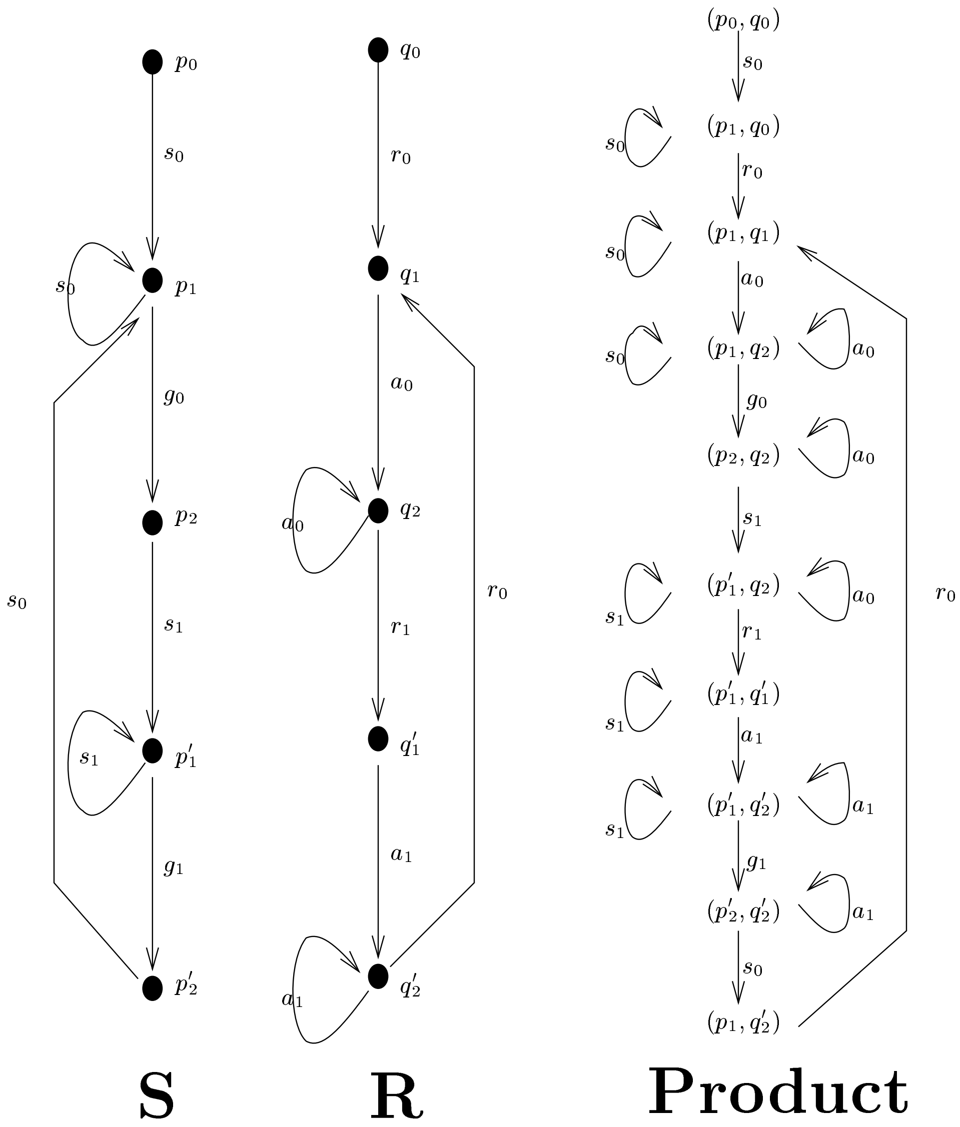


Fig. 9. A correct folding of the infinite state protocol for STP.

$loc(a) \stackrel{\rm def}{=} \{i \mid a \in \Sigma_i\}$. So $loc(a)$(called "locations of a") gives the set of processes that participate (or, synchronize) in the action a. By definition, for all $a \in \Sigma$, $loc(a) \neq \emptyset$. When $loc(a) \cap loc(b) = \emptyset$, we think of a and b as *independent* actions, as they can be performed concurrently by disjoint sets of processes.
A *distributed system* over $\widetilde{\Sigma}$ is a tuple $\widetilde{M} = (M_1, \ldots, M_n)$, where for all i in $Loc, M_i = (Q_i, \longrightarrow_i, q_i^0)$ is a TS over Σ_i. Global transitions of $\widetilde{M}$ are given by a *product transition system* (product TS): let $Q \stackrel{\rm def}{=} Q_1 \times \cdots \times Q_n$. A TS $\widehat{M} = (\widehat{Q}, \longrightarrow, (q_1^0, \cdots, q_n^0))$, where $\widehat{Q} \subseteq Q$, $\longrightarrow \subseteq (\widehat{Q} \times \Sigma \times \widehat{Q})$ and $(q_1^0, \cdots, q_n^0) \in \widehat{Q}$, is called a **product TS** of $\widetilde{M}$ on Σ if it satisfies the **asynchrony** condition: $(p_1, \ldots, p_n) \stackrel{a}{\longrightarrow} (q_1, \ldots, q_n)$ iff $\forall i \in loc(a), p_i \stackrel{a}{\longrightarrow}_i q_i$, and $\forall j \notin loc(a), p_j = q_j$. The language accepted by $\widetilde{M}$ is $L(\widetilde{M}) \stackrel{\rm def}{=} L(\widehat{M}, F)$. We use $\overline{p}, \overline{q}$ etc. to denote elements of Q. For any global state $\overline{s} = (p_1, p_2, \ldots, p_n)$, $\overline{s}[i]$ denotes the i-th component p_i.
Fix a distributed alphabet $\widetilde{\Sigma}=(\Sigma_1, \cdots, \Sigma_n)$. We also have a **commitment alphabet** $\widetilde{\mathcal{C}} = ((\mathcal{C}_1, \preceq_1), \cdots, (\mathcal{C}_n, \preceq_n))$. Each $\mathcal{C}_i$ is a nonempty set and for all $i \neq j$, $\mathcal{C}_i \cap \mathcal{C}_j = \{\perp\}$. $\preceq_i$ is a binary relation on $\mathcal{C}_i$ such that $\perp \preceq_i c$ for each $c \in \mathcal{C}_i$. 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*. Since we work with finite state systems, it suffices to consider finite $\mathcal{C}_i$. Let $\Phi \stackrel{\rm def}{=} \{\phi : Loc \mapsto \mathcal{C} \mid \forall i \in Loc, \phi(i) \in \mathcal{C}_i\}$.
In specific systems, the relations $\preceq_i$ may have more structure. For example, if the commitment alphabet is constructed from boolean formulae with implication as the ordering, we get a partial order on the alphabets. In fact, in all our figures of commitment alphabets, we treat them as pre-orders (reflexive and transitive relations) for ease of depiction, but such ordering is not mandatory by definition.

Definition 3.1 *Let* $i \in \{1, 2, \ldots, n\}$. *An* **AC transition system***(AC-TS) over* $(\Sigma_i, \widetilde{\mathcal{C}})$ *is a tuple* (M_i, f_i) *where* $M_i = (Q_i, \longrightarrow_i, q_i^0)$ *is a TS over* Σ_i *and* $f_i : Q_i \to \Phi$ *is called an* **assumption map**.

At a state $p \in Q_i$, if $f_i(p) = \phi$ then $\phi(i)$ is the commitment of M_i at p and $\phi(j), j \neq i$, is the assumption of M_i about M_j at p.

Definition 3.2 *An* **Assumption-compatible system(ACS)** *over* $(\widetilde{\Sigma}, \widetilde{\mathcal{C}})$ *is given by a tuple*

$$\widetilde{M} = (M_1, M_2, \cdots, M_n, < f_1, f_2, \cdots, f_n >, F),$$

where for each $i \in Loc, (M_i = (Q_i, \longrightarrow_i, q_i^0), f_i)$ *is an AC-TS over* $(\Sigma_i, \widetilde{\mathcal{C}})$, *and* $F \subseteq (Q_1 \times \ldots \times Q_n)$.

Global behaviour of $\widetilde{M}$ is given below as that of the product automaton $\widehat{M}$ associated with the system.

Definition 3.3 *Given an ACS* $\widetilde{M} = (M_1, M_2, \cdots, M_n, < f_1, f_2, \cdots, f_n >, F)$, *a global state* $(p_1, p_2, \cdots, p_n) \in Q$ *is called* **compatible** *iff*

$$\text{for all } i, j \in Loc : \; f_i(p_i)(j) \preceq_j f_j(p_j)(j).$$

Definition 3.4 *The product automaton of $\widetilde{M}$ is defined to be $(\widehat{M}, F)$ where*

1. $\widehat{M} = (\widehat{Q}, \longrightarrow, (q_1^0, \cdots, q_n^0))$ *is a product TS of* $\widetilde{M}$ *over* Σ *with* $\widehat{Q}$ *as the set of all compatible global states,*
2. $(q_1^0, \cdots, q_n^0) \in \widehat{Q}$, *and*
3. $F \subseteq \widehat{Q}$.

The class of languages over $\widetilde{\Sigma}$ accepted by ACS's is denoted as $\mathcal{L}(ACS_{\widetilde{\Sigma}})$.

$$\mathcal{L}(ACS_{\widetilde{\Sigma}}) = \{L \subseteq \Sigma^* \mid \exists \widetilde{\mathcal{C}} \text{ and an ACS } \widetilde{M} \text{ over } (\widetilde{\Sigma}, \widetilde{\mathcal{C}}) \text{ s.t. } L = L(\widehat{M}, F)\}.$$

Thus the behaviour of ACS's is given by a product construction. Note that the product contains compatible global states, and the rest are eliminated.
It is worth noting here how an apparent circularity between assumptions and commitments is resolved. If we were to consider a commitment as an 'offer', we could consider the following situation. Process 1, at local state s commits to c assuming d to be available from process 2. If, simultaneously, 2 in local state t offers d' assuming c' to be available from 1, we can have circularity or 'deadlock' when c and c', as well as d and d' are incompatible. However, the way compatible global states have been defined here, a global state consisting of local states s and t would simply be declared incompatible. Operationally, this can be seen as both processes making their offers first, and then checking whether what is on offer meets with the assumptions made.
The use of compatible states adds a great deal of modelling power. For instance, define the relation $\sim$ on Σ^* as: for all $x, y \in \Sigma^*$, $x \sim y \overset{\text{def}}{=} x\lceil i = y\lceil i$ for all $i \in Loc$, where $\lceil : (\Sigma^* \times Loc) \to \Sigma^*$ is the component projection map giving the maximal subsequence over any local alphabet. It is easy to see that $\sim$ is an equivalence, and it can be verified that $x \sim y$ holds exactly when x can be obtained by commuting independent actions in y. This is a relation of crucial interest in concurrency theory [Maz]. Languages accepted by product TS's (defined earlier) are closed under $\sim$, but $\mathcal{L}(ACS_{\widetilde{\Sigma}})$ need not be closed under $\sim$.
As an example, we describe an ACS in Fig. 10 that accepts the language $(ab)^*$, where $n = 2$ and $\widetilde{\Sigma} = (\{a\}, \{b\})$ (hence a and b are independent). Note that this language is not closed under $\sim$.

Example 3.1. Let $\widetilde{\mathcal{C}} = (\mathcal{C}_1, \mathcal{C}_2)$ where the commit alphabet $\mathcal{C}_i$ are as shown in Fig. 10. For ease of reference, we have annotated the local states by the respective $f_i(\cdot)$. We see that of the possible 9 global states, only 6 are compatible. For example, the global state (p_2, q_2) is compatible because at p_2, agent 1's assumption about agent 2 is $f_1(p_2)(2) = \nu_4$, at q_2 agent 2's commitment is $f_2(q_2)(2) = \nu_2$ and $\nu_4 \preceq_2 \nu_2$. Also, at q_2, agent 2's assumption about agent 1 is $f_2(q_2)(1) = \mu_5$, agent 1's commitment at p_2 is $f_1(p_2)(1) = \mu_2$ and $\mu_5 \preceq_1 \mu_2$. On the other hand the global state (p_1, q_2) is not compatible because $f_1(p_1)(2) = \nu_6 \not\preceq_2 \nu_2 = f_2(q_2)(2)$. The final states are: $\{(p_1, q_1), (p_2, q_2), (p_3, q_3)\}$.

In the design of ACS's, a crucial decision relates to the choice of commit alphabets. In effect, elements of commit alphabets code up sets of local states or

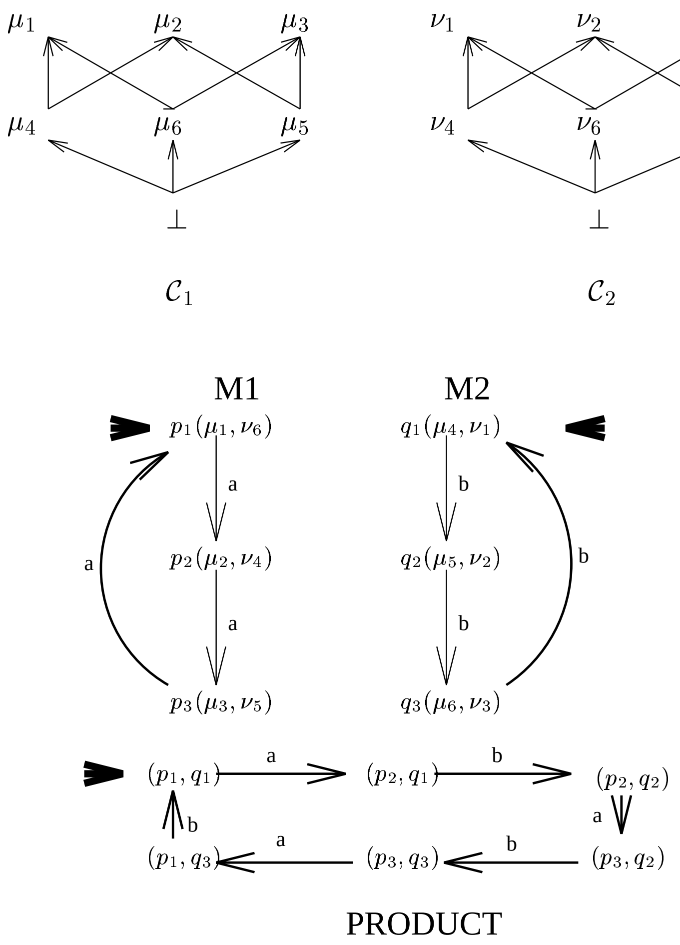


Fig. 10. An example ACS accepting $(ab)^*$ over the alphabet $(\{a\}, \{b\})$.

regions of transition systems. That is, when we write $f_1(p_2)(2) = \nu_4$ above, it is an assumption by process 1 at state p_2 that process 2 is in a set of states coded by ν_4. This approach, as opposed to the 'black-box' way of defining assumptions and commitments over the entire system behaviours, makes it possible for us to reason with a limited form of liveness. We can use commit alphabets and the assumption map to code up situations whereby process 1 eventually makes a condition p true, assuming that until then, 2 is in a region of its transition system, where it eventually gets to a state making a condition q true. We can symmetrically build a similar condition for process 2, and then the product eventually reaches a global state where 1 makes p true and 2 makes q true. However, formalizing such reasoning as a liveness composition rule in a formal logic is left to future work.
The following theorem constitutes the central result of the paper:

Theorem 3.1. $\mathcal{L}(ACS_{\widetilde{\Sigma}}) = Reg_{\Sigma}$.

The inclusion $\mathcal{L}(ACS_{\widetilde{\Sigma}}) \subseteq Reg_{\Sigma}$ is easy and follows from the fact that the product automata of ACS's are just finite state automata. The other inclusion, showing that every regular language over Σ is in $\mathcal{L}(ACS_{\widetilde{\Sigma}})$, is (understandably) complicated because when we want to accept arbitrary regular languages we need the ability to 'force' specific interleavings. For instance, when a and b are independent actions 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 [Zie] or cellular asynchronous automata [CMZ], where global states are decomposed preserving concurrency.
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 hence it is not locally presented. ¿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. The detailed proof is given in the Appendix.

4 Behavioural analogue of compatible products

A product operation on finite state automata corresponds to a shuffle operation on regular languages. Thus we can ask, what manner of shuffle corresponds to the compatible product of automata with assumption and commitment on states? We first answer this question below, and then present a distributed version of Kleene's theorem. For this section, fix a distributed alphabet $\widetilde{\Sigma} = (\Sigma_1, \ldots, \Sigma_n)$ and a finite commit alphabet $\widetilde{\mathcal{C}}$. Recall that $\Phi \stackrel{\text{def}}{=} \{\phi : Loc \mapsto \mathcal{C} \mid \forall i \in Loc, \phi(i) \in \mathcal{C}_i\}$. Since each $\mathcal{C}_i$ is finite, $|\Phi|$ is finite.
Note that in AC-transition systems local transitions are labelled by letters from the local alphabet, but the product of these TS's crucially depends on the assumptions and commitments assigned to local states through assumption maps.

Hence, if we want to have a shuffle operation corresponding to the product, local languages must encode assumptions and commitments. Then the shuffle operation on these languages should generate strings over Σ from local ones by using this information.
In order to bring assumptions and commitments into languages over distributed alphabets, we extend any given alphabet using assumption maps so that we have actions of the form $< a, \phi >$ where $\phi \in \Phi$.

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

4.1 Compatible shuffle

We want to define n-way shuffle for strings $x_1, \cdots, x_n$, where $x_i \in \Sigma_i^{c*}, i \in Loc$. By their structure, each x_i has an assumption map at every point essentially denoting the assumptions of the local state corresponding to x_i. Hence, globally, at every point one has n assumption maps, one for each string, denoting a global state. For a valid shuffle of the given strings, the global states that occur at each point have to be compatible. In terms of strings, the collection of local assumption maps have to be compatible.
Let Ξ denote the set of all possible assumption-commitment tuples for all the agents in the system. Formally, $\Xi = \{\xi \mid \xi : Loc \to \Phi\}$. For any $i, j \in Loc$, $\xi(i)$ is an assumption map, 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 $\forall\ i, j \in Loc$, $\xi(i)(j) \preceq_j \xi(j)(j)$.*

Definition 4.3 $\Sigma^{\Xi} = \{< a, \xi > \mid a \in \Sigma$ *and* $\xi \in \Xi\}$.

We use $\widehat{x}, \widehat{y}, \ldots$ to denote strings over Σ^{Ξ} which, intuitively, stand for sequences of global states. Since our goal is to establish a correspondence between the global strings and the runs of a compatible product automaton, these strings must somehow capture the compatibility condition internally. We call these strings *good.* Formally,

Definition 4.4 *Let $\widehat{x} =< a_1, \xi_1 >< a_2, \xi_2 >, \ldots, < a_k, \xi_k >\in \Sigma^{\Xi*}$. Then, $\widehat{x}$ is* **good** *w.r.t. an* initial environment ξ_0 *iff*

1. *for all $1 \leq l \leq k$, for all $j \notin loc(a_l)$, $\xi_{l-1}(j) = \xi_l(j)$, and*
2. *for all $0 \leq l \leq k$, ξ_l is feasible.*

As a minor observation, note that ϵ (the null string) is good w.r.t. some initial environment ξ iff ξ is feasible.
Now we define the notion of when strings over local alphabets can generate global strings. For this we need two projection maps. The *commit erasure* map: $\sigma : \Sigma^{c*} \to \Sigma^*$ is defined as : $\sigma(< a_1, \phi_1 > \cdots < a_k, \phi_k >) \stackrel{\text{def}}{=} a_1 \cdots a_k$. We use σ as a commit erasure map on strings over Σ^{Ξ} as well since there is no scope of confusion here.
The second projection map is the *component projection* map: $\widehat{\lceil}: (\Sigma^{\Xi*} \times Loc) \to \Sigma^{c*}$ defined by:

$$\widehat{x}\widehat{\lceil}i = \begin{cases} \epsilon & \text{if } x = \epsilon, \\ \widehat{y}\widehat{\lceil}i & \text{if } \widehat{x} = \widehat{y}\cdot < a, \xi > \text{ and } i \notin loc(a), \text{ and} \\ (\widehat{y}\widehat{\lceil}i)\cdot < a, \xi(i) > & \text{if } \widehat{x} = \widehat{y}\cdot < a, \xi > \text{ and } i \in loc(a). \end{cases}$$

Definition 4.5 *A string $\widehat{x} \in \Sigma^{\Xi*}$ is called a* **witness** *for $x \in \Sigma^*$ under $\xi \in \Xi$ if $\widehat{x}$ is good w.r.t. ξ and $\sigma(\widehat{x}) = x$.*
Let $x_i \in \Sigma_i^{c*}$, $i \in \{1, \ldots, n\}$. A string $x \in \Sigma^*$ is said to be **generated** *by $(x_1, x_2, \ldots, x_n)$ under ξ if there is a witness $\widehat{x}$ for x under ξ such that for all $i \in Loc, x_i = \widehat{x}\widehat{\lceil}i$.*

Notice that when $x \in \Sigma^*$ is generated by $(x_1, x_2, \ldots, x_n)$ under ξ, $x\lceil i = \sigma(x_i)$. This is because, $\sigma(x_i) = \sigma(\widehat{x}\widehat{\lceil}i) = \sigma(\widehat{x})\lceil i = x\lceil i$.
We now define the compatible shuffle of languages over local extended alphabets using the definition of generation.

Definition 4.6 *For all $i \in Loc$, let $L_i \subseteq \Sigma_i^{c*}$, and let $\xi \in \Xi$. We define the n-ary* **compatible shuffle** *of these languages under the assumption environment ξ by:*
$(L_1||L_2||\ldots L_n)_\xi \stackrel{\text{def}}{=} \{x \in \Sigma^* \mid x$ *is generated by a tuple* $(x_1, x_2, \ldots, x_n)$ *under* ξ, *where for all* i, $x_i \in L_i\}$.

We are interested in the compatible shuffle of regular languages over the local alphabets. So we define the following class of languages.

Definition 4.7 *Let $\mathcal{L}(AC - shuffle)_{\widetilde{\Sigma}}$ denote the least class that includes the set $\{L \subseteq \Sigma^* \mid$ for some commit alphabet $\widetilde{\mathcal{C}}$ and ξ over $\widetilde{\mathcal{C}}$, there exist regular languages $L_i \subseteq \Sigma_i^{c*}$ such that $L = (L_1 \parallel \ldots \parallel L_n)_\xi\}$ and is closed under union.*

A language in $\mathcal{L}(AC - shuffle)_{\widetilde{\Sigma}}$ will be referred to as an AC-shuffle language (with $\widetilde{\Sigma}$ implicit).

4.2 ACS's and commitment structure

There is a fairly obvious association between states and runs of an ACS and good strings over Σ^{Ξ}. We make it explicit in the following. Let $\widetilde{M} = (M_1, \ldots, M_n, < f_1, \cdots, f_n >, F)$ be an ACS and $\widehat{M} = (\widehat{Q}, \longrightarrow, < q_0^1, \cdots, q_0^n >, F)$ be the compatible product of $\widetilde{M}$.

Definition 4.8 *Let* $(q_1, \cdots, q_n) \in \widehat{q}$. *Then,* $\mathrm{env}(q_1, \cdots, q_n) \stackrel{\mathrm{def}}{=} \xi$, *where* $\xi(i) = f_i(q_i)$, *for all* $i \in Loc$.

Let $(q_1, \cdots, q_n)$ be a compatible state and $\xi = env(q_1, \cdots, q_n)$. Then ξ is feasible. This is because of the following reason. Since $(q_1, \cdots, q_n)$ is compatible, for all $i, j \in Loc, f_i(q_i)(j) \preceq_j f_j(q_j)(j)$. Then by the construction, for all i, j in $Loc, \xi(i)(j) \preceq_j \xi(j)(j)$. This implies feasibility of ξ.
We can associate strings over Σ^{c*} with runs of $\widehat{M}$ in a canonical fashion. Fix $x = a_1 a_2 \ldots a_k$, and a run $\rho = (q_1^0, \ldots, q_n^0) \xrightarrow{a_1} (q_1^1, \ldots, q_n^1) \ldots \xrightarrow{a_k} (q_1^k, \ldots, q_n^k)$ on x in $\widehat{M}$. Define $c(\rho) =< a_1, \xi_1 > \ldots < a_k, \xi_k > \in \Sigma^{\Xi*}$ by: for $1 \leq l \leq k$, $\xi_l = env(q_1^l, \ldots, q_n^l)$.
Note that when $k = 0$, that is, when $x = \epsilon$, $c(\rho) = \epsilon$. The initial environment associated with the run is defined as $env(q_1^0, \ldots, q_n^0)$. Note that $env(q_1^0, \ldots, q_n^0)$ is a feasible assumption environment (because $(q_1^0, \ldots, q_n^0)$ is compatible.)

Proposition 4.9 *Let* ρ *be a run in* $(q_1, \cdots, q_n) \stackrel{x}{\Longrightarrow} (p_1, \cdots, p_n)$. *Then,* $c(\rho)$ *is a witness for* x *under* $\mathrm{env}(q_1, \cdots, q_n)$.

Proof. By definition, $\sigma(c(\rho)) = x$. Hence it suffices to show that $c(\rho)$ is good w.r.t. $env(q_1, \cdots, q_n)$.
Let the run ρ be $(q_1^0, \ldots, q_n^0) \xrightarrow{a_1} (q_1^1, \ldots, q_n^1) \ldots \xrightarrow{a_k} (q_1^k, \ldots, q_n^k)$.
Fix l such that $1 \leq l \leq k$. For all $j \notin loc(a_l)$, $q_j^{l-1} = q_j^l$ by asynchrony. Hence, by definition of $c(\rho)$, $\xi_{l-1}(j) = f_j(q_j^{l-1}) = f_j(q_j^l) = \xi_l(j)$.
Also, since all the states on the run are compatible, as we have observed before, the assigned assumption environments are feasible. This proves that $c(\rho)$ is good w.r.t. $env(q_1, \cdots, q_n)$. □

Proposition 4.10 *Let* ρ *denote a run* $(q_1, \cdots, q_n) \stackrel{x}{\Longrightarrow} (p_1, \cdots, p_n)$ *in* $\widehat{M}$. *Then for all* $i \in Loc$, *there exist* $x_i \in \Sigma_i^{c*}$ *such that* $q_i \stackrel{x_i}{\Longrightarrow}{}_i^c p_i$ *and* x *is generated by the tuple* $(x_1, x_2, \cdots, x_n)$ *under* $\mathrm{env}(q_1, \cdots, q_n)$.

Proof. By the previous proposition $c(\rho)$ is a witness for x under $env(q_1, \cdots, q_n)$. Take $x_i = c(\rho)\widehat{\lceil} i$, for all $i \in Loc$. Then, x is generated by the tuple $(x_1, x_2, \cdots, x_n)$ under $env(q_1, \cdots, q_n)$. From the definition of $\widehat{\lceil}$ and $\longrightarrow_i^c$, one can carry out an induction argument on the length of x to show that $q_i \stackrel{x_i}{\Longrightarrow}{}_i^c p_i$. □

Proposition 4.11 *Let* $\widehat{x} =< a_1, \xi_1 >< a_2, \xi_2 >, \ldots, < a_k, \xi_k > \in \Sigma^{\Xi*}$ *be good w.r.t.* $\xi_0 = \mathrm{env}(q_1, \cdots, q_n)$ *such that* $q_i \stackrel{\widehat{x}\widehat{\lceil} i}{\Longrightarrow}{}_i^c p_i$. *Then,* $\xi_k(i) = f_i(p_i)$. *(Since* ξ_k *is feasible this implies that* $(p_1, \cdots, p_n)$ *is compatible.)*

Proof. Let l be the last i-action in $\widehat{x}$. If $l = 0$ (meaning $\widehat{x}\lceil i$ is empty), then $p_i = q_i$ and by goodness of $\widehat{x}$, $\xi_k(i) = \xi_0(i) = f_i(p_i)$.

If $l \neq 0$, then there is an $r_i \in Q_i$ such that $q_i \stackrel{y}{\Longrightarrow}{}_i^c r_i \xrightarrow{< a_l, \xi_l(i) >}{}_i^c p_i$, where $\widehat{x}\widehat{\lceil} i = y \cdot < a_l, \xi_l(i) >$. Hence, $f_i(p_i) = \xi_l(i)$. By goodness of $\widehat{x}$, $\xi_l(i) = \xi_k(i)$. Therefore, $f_i(p_i) = \xi_k(i)$ and we are done. □

Proposition 4.12 *Suppose $x \in \Sigma^*$ and for all $i \in Loc$, there exist $x_i \in \Sigma_i^{c*}$ such that $q_i \overset{x_i}{\Longrightarrow}_i^c p_i$ and x is generated by the tuple $(x_1, x_2, \cdots, x_n)$ under* $\mathrm{env}(q_1, \cdots, q_n)$. *Then $(q_1, \cdots, q_n) \overset{x}{\Longrightarrow} (p_1, \cdots, p_n)$ in $\widehat{M}$.*

Proof. (by induction on length of x) The assumptions of the claim are rephrased as follows: there exist $x_i \in \Sigma_i^{c*}$ such that $q_i \overset{x_i}{\Longrightarrow}_i^c p_i$ and there exists a witness $\widehat{x} \in \Sigma^{\Xi *}$ of x under $env(q_1, \cdots, q_n)$ such that $\widehat{x}\lceil i = x_i$. We have to show that $(q_1, \cdots, q_n) \overset{x}{\Longrightarrow} (p_1, \cdots, p_n)$. Note that since $\widehat{x}$ is good w.r.t. $env(q_1, \cdots, q_n)$, $(q_1, \cdots, q_n)$ is compatible from the definition of goodness. From Proposition 4.11 $(p_1, \cdots, p_n)$ is also compatible, so both $(q_1, \cdots, q_n)$ and $(p_1, \cdots, p_n)$ are in $\widehat{M}$.
Let $x = \epsilon$. Then the witness $\widehat{x}$ must be ϵ since $\sigma(\widehat{x}) = x$. This implies, for all $i \in Loc$ $x_i = \epsilon$. Hence, $q_i = p_i$ for all i. Then, it is obvious that $(q_1, \cdots, q_n) \overset{x}{\Longrightarrow} (p_1, \cdots, p_n)$.
For the induction step, let $x = ya$. Then the witness $\widehat{x}$ for x must be of the form $\widehat{x} = \widehat{y} \cdot < a, \xi >$, where $\sigma(\widehat{y}) = y$. Since $\widehat{x}$ is a witness under $env(q_1, \cdots, q_n)$ it is good w.r.t. $env(q_1, \cdots, q_n)$. Hence, $\widehat{y}$ must also be good w.r.t $env(q_1, \cdots, q_n)$. Therefore, $\widehat{y}$ is a witness of y under $env(q_1, \cdots, q_n)$. Let $\widehat{y}\lceil i = y_i$ for all $i \in Loc$. Then, $\widehat{y}$ is generated by $(y_1, \cdots, y_n)$ under $env(q_1, \cdots, q_n)$.
It is given that for all $i \in Loc, \widehat{x}\lceil i = x_i$. Hence for every $i \notin loc(a), x_i = \widehat{x}\lceil i = \widehat{y}\lceil i = y_i$ and for every $i \in loc(a)$, $x_i = y_i \cdot < a, \xi(i) >$.
It is also given that for all $i \in Loc, q_i \overset{x_i}{\Longrightarrow}_i^c p_i$. Hence, for all $i \notin loc(a)$, $q_i \overset{y_i}{\Longrightarrow}_i^c p_i$ and for all $i \in loc(a)$, there is an $r_i \in Q_i$ such that $q_i \overset{y_i}{\Longrightarrow}_i^c r_i \overset{< a, \xi(i) >}{\longrightarrow}_i^c p_i$. Therefore, $f_i(p_i) = \xi(i)$.
Since we have:

1. for all $i \notin loc(a)$, $q_i \overset{y_i}{\Longrightarrow}_i^c p_i$ and
2. for all $i \in loc(a)$, $q_i \overset{y_i}{\Longrightarrow}_i^c r_i$,

by induction hypothesis, $(q_1, \cdots, q_n) \overset{y}{\Longrightarrow} (s_1, \cdots, s_n)$, where for $i \notin loc(a)$, $s_i = p_i$ and for $i \in loc(a)$ $s_i = r_i$.
Since $\widehat{M}$ satisfies the asynchrony condition (definition of Product TS), from the fact that for $i \in loc(a)$, $s_i = r_i \overset{< a, \xi(i) >}{\longrightarrow}_i^c p_i$, it follows that $(s_1, \cdots, s_n) \overset{a}{\longrightarrow} (p_1, \cdots, p_n)$. Therefore, finally, $(q_1, \cdots, q_n) \overset{ya}{\Longrightarrow} (p_1, \cdots, p_n)$, as required. □

4.3 Equivalence of AC-Shuffle and AC-systems

We now establish a correspondence between AC-shuffle of languages over Σ_i^c and languages accepted by ACS's. Note that the local automata are over Σ_i. How does one relate languages over Σ_i^c and local automata over Σ_i? This is simple: since the states are annotated with assumption maps from Φ, we take these into account (like in Moore machines) alongwith the labels of transitions so that languages accepted by local automata are actually over Σ_i^c. For example,

if $q_0 \xrightarrow{a_1}_i q_1 \xrightarrow{a_2}_i q_2$, q_0 is the initial state and q_2 is the final state, then we say that this is an accepting path for $< a_1, f_i(q_1) > \cdot < a_2, f_i(q_2) >$.
Since the local automata are FSA's, languages thus accepted by these are regular over Σ_i^c. But, is the reverse true? In other words, for any regular language L over Σ_i^c, is there an AC-automaton over Σ_i that accepts L? We show in the following that this is indeed the case.
We first define language acceptance (in the above sense) of AC-automata. Let $(M_i = (Q_i, \longrightarrow_i, q_i^0), f_i)$ be an AC-transition system over Σ_i. We transform M_i into a TS $M_i' = (Q_i', \longrightarrow_i^c, q_i^{c0})$ over Σ_i^c where, $Q_i^c = Q_i$, $q_i^{c0} = q_i^0$ and $\longrightarrow_i^c \subseteq Q_i^c \times \Sigma_i^c \times Q_i^c$ is defined as follows: $p \overset{<a,\phi>}{\longrightarrow_i^c} q$ iff $p \xrightarrow{a}_i q$ and $\phi = f_i(q)$.
This one step transition can be extended to $\Longrightarrow_i^c$ for strings from Σ_i^{c*}.
Then given an AC-automaton $((M_i, f_i), F_i)$, we define:
$L_m((M_i, f_i), F_i) \stackrel{\text{def}}{=} L(M_i', F_i)$.

Proposition 4.13 *$L \in Reg_{\Sigma_i^c}$ iff there is an AC-automaton (M, F) over Σ_i such that $L = L_m(M, F)$.*

Proof. One direction of the proof follows immediately from the definition of L_m.
For the other direction, suppose $L \in Reg_{\Sigma_i^c}$. Since L is regular over Σ_i^c, there exist some FSA $(A = (Q, \longrightarrow, q^0), F)$ such that $L = L(A, F)$.
Note that in this FSA, two transitions with different assumption maps may be pointing to the same state. So we refine the states so that transitions that point to a state have the same assumption map. This is possible because the number of assumption maps is finite. We do this as follows.
Define the AC-automaton $((M, f), G)$ as follows. Let $M = (P, \Longrightarrow, p^0)$ where,

- $P = \{q^0\} \cup \bigcup_{q \in Q} \{(q, \phi) \mid$ there is a transition $p \xrightarrow{<a,\phi>} q$ in $A\}$,
- $p^0 = q^0$,
- $(p, \phi_1) \xRightarrow{a} (q, \phi_2)$ iff $p \xrightarrow{<a,\phi>} q$ and $\phi_2 = \phi$, and
- for all states (p, ϕ), $f((p, \phi)) = \phi$.

Finally, $G = \{(p, \phi) | p \in F\}$.
In order to check that $L_m((M, f), G) = L$, we transform M in to the following TS $(M^c = (Q^c, \longrightarrow^c, q^{c0}), F^c)$ over Σ_i^c, where

- $Q^c = P$,
- $q^{c0} = p^0$,
- $(p, \phi_1) \overset{<a,\phi>}{\longrightarrow^c} (q, \phi_2)$ iff $(p, \phi_1) \xRightarrow{a} (q, \phi_2)$ and $\phi = f((q, \phi_2)) = \phi_2$,
- Finally, $F^c = G$.

By definition, $L_m(M, G) = L(M^c, F^c)$. So it suffices to show that $L(M^c, F^c) = L(A, F)$.
We note that M^c *simulates* A via the map Θ, where $\Theta((p, \phi)) = p$ and $\Theta(q^0) = q^0$; that is, states of M^c map to states of A so that transitions of M^c on an

action map to transitions on the same action, and initial and final states map to initial and final states respectively. It is then routine to show language equality and the proposition follows. □

We prove the following proposition which is used later to show that languages obtained by AC-systems and shuffle coincide.

Proposition 4.14 *Suppose, for all* $i \in Loc$, $L_i = L_m((M_i, f_i), F_i)$. *Take an AC-system* $\widetilde{M} = (M_1, M_2, \ldots, M_n, < f_1, \cdots, f_n >, \Pi_{i \in Loc} F_i)$ *with the initial state* $(q_1^0, \cdots, q_n^0)$. *Let* ξ_0 *denote* $\mathrm{env}(q_1^0, \cdots, q_n^0)$. *Then* $L(\widetilde{M}) = (\| L_i)_{\xi_0}$.

Proof. ($\subseteq$:) Let $x \in L(\widetilde{M})$. Then there is a final state $(q_1^f, \cdots, q_n^f) \in \Pi_{i \in Loc} F_i$ such that $(q_1^0, \cdots, q_n^0) \stackrel{x}{\Longrightarrow} (q_1^f, \cdots, q_n^f)$. By Proposition 4.10,there exist $x_i \in \Sigma_i^{c*}$ such that $q_i^0 \stackrel{x_i}{\Longrightarrow}{}_i^c q_i^f$ and x is generated by the tuple $(x_1, x_2, \cdots, x_n)$ under $\xi_0 = env(q_1^0, \cdots, q_n^0)$. Since each q_i^f is in F_i, each x_i is in L_i. Hence by definition of shuffle, $x \in (\| L_i)_{\xi_0}$. Therefore, $L(\widetilde{M}) \subseteq (\| L_i)_{\xi_0}$.
($\supseteq$:) Let $x \in (\| L_i)_{\xi_0}$. By definition, there exist $x_i \in \Sigma_i^{c*}$ such that $x_i \in L_i$ and x is generated by the tuple $(x_1, x_2, \cdots, x_n)$ under ξ_0.

Since $x_i \in L_i$, for all $i \in Loc$, there is some $q_i^f \in F_i$ such that $q_i^0 \stackrel{x_i}{\Longrightarrow}{}_i^c q_i^f$. By Proposition 4.12 we have $(q_1^0, \cdots, q_n^0) \stackrel{x}{\Longrightarrow} (f_1, \ldots, f_n) \in \Pi_{i \in Loc} F_i)$. Hence $x \in L(\widetilde{M})$. Therefore, $(\| L_i)_{\xi_0} \subseteq L(\widetilde{M})$. □

We state the main theorem connecting AC-shuffle and languages accepted by AC-systems.

Theorem 4.15 $\mathcal{L}(AC\text{-}Shuffle_{\widetilde{\Sigma}}) = Reg_{\Sigma} = \mathcal{L}(ACS_{\widetilde{\Sigma}})$.

Proof. Consider $L \in \mathcal{L}(ACS_{\widetilde{\Sigma}})$. Then for some ACS $\widetilde{M} = (M_1, M_2, \cdots, M_n, < f_1, \cdots, f_n >, F)$, $L = L(\widetilde{M})$. One can then write L as $L = \bigcup_{q^f \in F} \mathcal{L}(\widetilde{M}_{q^f})$ where $\widetilde{M}_{q^f} = (M_1, \cdots, M_n), < f_1, \cdots, f_n >, \{q^f\})$. Let $q^f = (q_1^f, q_2^f, \cdots, q_n^f)$ and $L_i = L_m((M_i, f_i), \{q_i^f\})$. By the previous proposition there is an assumption environment ξ_0 such that $L(\widetilde{M}_{q^f}) = (\| L_i)_{\xi_0}$. This places $L(\widetilde{M}_{q^f})$ in $\mathcal{L}(AC\text{-}Shuffle_{\widetilde{\Sigma}})$. Then, since $\mathcal{L}(AC\text{-}Shuffle_{\widetilde{\Sigma}})$ is closed under union, L also is placed in it.
Let $L = (\| L_i)_{\xi_0}$. Since L_i's are regular over Σ_i^c, by Proposition 4.13, there are AC-automata $((M_i, f_i), F_i)$ such that $L_i = L_m((M_i, f_i), F_i)$. We show that $L \in Reg_{\Sigma}$. By Proposition 4.14, the ACS $M = (M_1, \cdots, M_n, < f_1, \cdots, f_n >, \Pi_{i \in Loc} F_i)$ accepts L. Since M is an FA over Σ, $L \in Reg_{\Sigma}$. Since any language in $\mathcal{L}(AC\text{-}Shuffle_{\widetilde{\Sigma}})$ is either an AC-shuffle language or a union of AC-shuffle languages, $\mathcal{L}(AC\text{-}Shuffle_{\widetilde{\Sigma}}) \subseteq Reg_{\Sigma}$. The theorem then follows from Theorem 3.1. □

5 A Kleene theorem for ACS's

We now consider the question of syntax for languages in $\mathcal{L}(ACS_{\widetilde{\Sigma}})$. 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.

$$\begin{aligned} ACREG_i &::= \emptyset \mid \ < a, \phi > \in \Sigma_i^c \mid p + q \mid p; q \mid p^* \\ ACREG &::= (\|_{i=0}^n r_i)_\xi, r_i \in ACREG_i,\ \xi \in \Xi \\ &\quad \mid\ R_1 + R_2, R_i \in ACREG. \end{aligned}$$

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

- $[\emptyset]_i = \emptyset$.
- $[< a, \phi >]_i = \{< a, \phi >\}$.
- $[p + q]_i = [p]_i \cup [q]_i$.
- $[p; q]_i = [p]_i \cdot [q]_i$.
- $[p^*]_i = ([p]_i)^*$.
- $[(r_1 \| r_2 \| \cdots \| r_n)_\xi] = ([r_1]_1 \ \widehat{\|}\ [r_2]_2 \ \widehat{\|}\ \cdots \ \widehat{\|}\ [r_n]_n)_\xi$.
- $[R_1 + R_2] = [R_1] \cup [R_2]$.

Thus, the $ACREG_i$ expressions give languages over Σ_i^c and then $ACREG$ expressions are given semantics via AC-shuffle of these local languages and their unions.
The class of regular languages generated by the $ACREG$ expressions is denoted as $\mathcal{L}(ACREG)$. Formally, $\mathcal{L}(ACREG)_{\widetilde{\Sigma}} = \{L \subseteq \Sigma^* \mid$ for some commit alphabet $\widetilde{\mathcal{C}}$, there is an $R \in ACREG$ over $\widetilde{\Sigma^c}$ such that $L = [R]\}$.
Then, from the semantics and Theorem 4.15, we get the following characterization.

Theorem 5.1 $\mathcal{L}(ACS)_{\widetilde{\Sigma}} = \mathcal{L}(ACREG)_{\widetilde{\Sigma}} = Reg_\Sigma$.

5.1 A different syntax

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 $FACREG$ stands for assumptions and commitments occurring 'free' within expressions):

$$\begin{aligned} FACREG_i &::= \emptyset \mid a \in \Sigma_i \mid \phi \in \Phi \mid p + q \mid p; q \mid p^* \\ FACREG &::= r_1 \| r_2 \| \cdots \| r_n, r_i \in FACREG_i \mid R_1 + R_2, R_i \in FACREG \end{aligned}$$

We wish to map $FACREG_i$ expressions via a function $\langle\rangle_i$ to languages over Σ_i^c so that at the global level, $FACREG$ expressions can be given semantics by AC-shuffle of local languages as before. The natural semantics of $FACREG_i$ expressions give regular languages over $\Sigma \cup \Phi$ (call the semantic function $[]_i$). We translate these languages to languages over Σ_i^c in a systematic way, preserving regularity.

We describe a translation scheme that uniformly translates each string of the given language. We illustrate this with a running example to make the basic ideas clear. Let $x = a_1\phi_1\phi_2a_2a_3\phi_3a_4 \in (\Sigma \cup \Phi)^*$. The basic idea is to first convert every string over $(\Sigma \cup \Phi)$ to one where the letters and assumption maps alternate.

1. if there are consecutive letters we insert a $\perp$ in between them. Thus we translate x to $a_1\phi_1\phi_2a_2\perp a_3\phi_3a_4$.
2. if there are consecutive assumption maps then retain only the last one in the sequence. Thus we get $a_1\phi_2a_2\perp a_3\phi_3a_4$.
3. ensure that there is an assumption map at the beginning and at the and. If there are not any, we put $\perp$. Thus, we get $\perp a_1\phi_2a_2\perp a_3\phi_3a_4\perp$.
4. from the first letter onwards, pair up consecutive letter and assumption map. Thus, finally, $\perp\cdot < a_1, \phi_2 > \cdot < a_2, \perp > \cdot < a_3, \phi_3 > \cdot < a_4, \perp >$.

For some more examples, note the following:

1. $a_1\phi_1a_2\phi_2$ is translated to $\perp\cdot < a_1, \phi_1 > \cdot < a_2, \phi_2 >$,
2. $a_1a_2\phi_1\phi_2$ is translated to $\perp\cdot < a_1, \perp > \cdot < a_2, \phi_2 >$, and
3. $\phi_0a_1\phi_1a_2$ is translated to $\phi_0\cdot < a_1, \phi_1 > \cdot < a_2, \perp >$.

It should be clear that this translation is actually a function, call it h such that $h : (\Sigma_i \cup \Phi)^* \to \Phi \cdot \Sigma_i^{c*}$. Also following the steps described above, one can show that h can be expressed as a composition of homomorphisms, hence h is itself a homomorphism.

Suppose $L \subseteq (\Sigma \cup \Phi)^*$ is regular. Then, $L' = h(L)$ is a regular language over $\Phi \cup \Sigma_i^c$. Moreover, since for any string in L', only the first element is in Φ and $|\Phi|$ is finite, L' can be expressed as $\bigcup_{\phi\in\Phi} \phi \cdot L'_\phi$ where L'_ϕ is regular over Σ_i^c.

Definition 5.2 *Let $r \in FACREG_i$. Then* $\langle r\rangle_i \stackrel{\text{def}}{=} h([r]_i)$.

Consider the languages $\phi_i \cdot L_i, i \in Loc$. The shuffle of these languages is given as: $(L_1 \mathbin{\widehat{\|}} \cdots \mathbin{\widehat{\|}} L_n)_{(\phi_1,\cdots,\phi_n)}$ which is the AC-shuffle of the L_i's with the initial environment $(\phi_1, \cdots, \phi_n)$.

The $FACREG$ expressions are now given semantics via AC-shuffle.

Definition 5.3 *Let $R = (r_1 \widehat{\|}_F \cdots \widehat{\|}_F r_n) \in FACREG$ and $\langle r_i\rangle_i = \bigcup_{k\in\Gamma_i} \phi_{ik}L_{ik}$. Then,*

$$\langle R\rangle \stackrel{\text{def}}{=} \bigcup_{j_l\in\Gamma_l, l\in Loc} (L_{1j_1} \mathbin{\widehat{\|}} \cdots \mathbin{\widehat{\|}} L_{nj_n})_{(\phi_{1j_1},\cdots,\phi_{nj_n})}.$$

By the very definition, $\mathcal{L}(FACREG_{\widetilde{\Sigma}}) \subseteq \mathcal{L}(ACREG_{\widetilde{\Sigma}})$.
In order to show that the other inclusion also holds, we need to prove that for any $ACREG$ expression r, there is a $FACREG$ expression r' such that $\langle r' \rangle = [r]$. From the way h is described above, it is easy to see that the alphabetic homomorphism $d : \Sigma_i^c \to (\Sigma_i \cdot \Phi)$ defined as $d(< a, \phi >) = a\phi$ suffices for the proof, because then $h(\phi \cdot d(x)) = \phi \cdot x$. We omit the monotonous technical details and summarize the result below.

Theorem 5.1. $\mathcal{L}(ACREG_{\widetilde{\Sigma}}) = \mathcal{L}(FACREG_{\widetilde{\Sigma}})$.

6 Discussion

We have proved that ACS's over $\widetilde{\Sigma}$ characterize all regular languages over Σ. Hence, languages accepted by ACS's are closed under complementation, union and concatenation. Indeed, the construction gives a *deterministic* ACS for any regular language, hence we have determinization as well. We can also directly define constructions on ACSs to show closure under ACSs though we do not as yet have a direct proof of determinization.
A natural extension of the notions here relates to **infinite** behaviour of ACS's. We can indeed show that using appropriate Muller acceptance conditions, the results here extend to ω-regular languages as well ([Moh]). This is of importance in the context of verifying liveness properties of systems, which are specified on infinite system computations.
An interesting consideration on ACSs relates to **sub-regular** languages. It is easily seen that the class of languages obtained by shuffle or synchronized shuffle is but a small subclass of regular languages. A larger class of sub-regular languages are those regular languages closed under the equivalence $\sim$ defined in Section 3. We can show that these classes can be obtained by simple restrictions on the structure of the commit alphabets of ACS's. Proceeding along these lines, we can study ACSs with a specific commit structure and we find new classes of sub-regular languages ([Moh]).
It should be clear that ACS's have strong relationships with structures studied in concurrency theory. In fact, the techniques that we use while constructing distributed state automata in the process of decomposing regular behaviours arise mainly from partial order models of concurrency. It is also worth noting here that the study of ACSs arose from the initial work in [Ram1] where we studied such a decomposition theorem in a categorical framework, obtaining local presentations for a subclass of event structures. In [MR1], we first proved a decomposition theorem only for the class of regular trace languages, and only later generalized it to the results here. Thus, it seems evident that ACS's bear a formal relationship to automata over partial orders, but a precise elucidation of such a relationship seems difficult.
These remarks are intended here to emphasize the fact that the automata thory of ACS's does seem to be rich in interesting notions. The study of these, particularly the complexity of decision questions on ACS's, is important for the design and verification methodology that we outline here.

We conclude the paper with some remarks on the relevance of ACS's in the context of compositional verification introduced in Section 1. Note that each process i in an ACS over alphabet Σ_i is describing a regular subset of Σ_i^*. Consider a linear time temporal logic defined as follows (along the lines of [Ram3]): Let P_i be a set of atomic propositions, $i \in Loc$. The formulas of Φ_i are defined by:

$$\alpha \in \Phi_i ::= p \in P_i \mid \neg\alpha \mid \alpha_1 \vee \alpha_2 \mid \ < a > \alpha,\ a \in \Sigma_i \mid \alpha_1 \mathbf{U} \alpha_2$$

$$\mid [now](\alpha @ j),\ j \in Loc,\ \alpha \in \Phi_j$$

The semantics of these formulas can be defined easily on runs of ACS's. Note that formulas of Φ_i are interpreted on time instants on the 'local clock' of process i, and $[now]\alpha @ j$ constrains any global state to be such that the local instant for process j satisfies α. Thus, this modality acts like an assumption specification; the commitment made by a process is simply whatever is true at that state. Now let Ψ define the syntax of global formulas:

$$\phi \in \Psi ::= \alpha @ j,\ j \in Loc,\ \alpha \in \Phi_j \mid \neg\phi \mid \phi_1 \vee \phi_2$$

Now, we can associate an ACS with every formula ϕ such that the language it accepts exactly corresponds to the models of the formula ϕ. Such a *formula automaton construction* is at the heart of automata based model checking procedures. We are currently investigating how we can exploit the structure of the constructed formula automaton for compositional model checking.

References

[AFR] Apt, K.R., Francez, N. and de Roever, W.P., "A proof system for communicating sequential processes", *ACM TOPLAS*, vol. 2, 1980, 359-385.

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

[AL1] Abadi, M. and Lamport, L., "Composing Specifications", *TOPLAS*, vol. 15, 1993, 73-132.

[AL2] Abadi, M. and Lamport, L., "Conjoining Specifications", *TOPLAS*, vol. 17, 1995, 507-534.

[AAF] Abadi, M., Alpern,B., Apt, K.R., Francez, N., Katz, S., Lamport, L. and Schneider, F.B., "Preserving liveness: comments on "Safety and Liveness from a Methodological Point of View"". *IPL*, vol. 40, 1991, 141-142.

[BKP] Barringer, H., Kuiper, R. and Pnueli. A., "Now you may compose temporal logic specifications", *Proc. of STOC*, 1984, 51-63.

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

[Dij] Dijkstra, E.W., "Programming considered as a human activity", *Proc. IFIP 65*, 1965, 213-217.

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

[Flo] Floyd, W.F., "Assigning meanings to programs", *Proc. of Symp. in Appl. Math. XIX*, Americal Mathematical Society, 1967, 19-32.

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

[KV] Kupferman, O. and Vardi, M., "An automata-theoretic approach to modular model checking", *Proc. COMPOS 97*, Malente-Gremsmühlen, Germany, 1997.

[Maz] Mazurkiewicz, A., "Basic notions of trace theory", *LNCS 354*, 1989, 285-363.

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

[MP] Manna, Z. and Amir Pnueli, A., *Temporal logic of reactive systems: specification*, Springer-Verlag, New York, 1992.

[Moh] Mohalik S.K., *Local presentations for finite state distributed systems*, Ph.D. thesis, University of Madras, 1999.

[MR1] Mohalik S.K. and Ramanujam, R., "Assumption - Commitment in automata", *Proc. of FST & TCS '97 in LNCS 1346*, Kharagpur, 1997, 153-168.

[MR2] Mohalik S.K. and Ramanujam, R., "A presentation of regular languages in the assumption - commitment framework", *Proc. of CSD'98*, Aizu-Wakamatsu, Japan, 1998, 250-260 .

[OG] Owicki, S. and Gries, D., "Verifying properties of parallel programs: an axiomatic approach", *CACM*, vol. 19, 1976, 279-285.

[OL] Owicki, S. and Lamport, L., "Proving liveness properties of concurrent programs." *TOPLAS*, vol. 4, 1982, 455-495.

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

[PL] Pnueli, A. and Rosner, R., "Distributed reactive systems are hard to synthesize". *FOCS*, 1990, 746-757.

[QM] Quiwen Xu and Mohalik. S.K., "Compositional reasoning using the assumption - commitment paradigm", *Proc. COMPOS 97*, Malente-Gremsmühlen, Germany, 1997.

[Ram1] Ramanujam, R., "A local presentation of synchronizing systems", in *Structures in Concurrency Theory*, ed: Jörg Desel, Springer workshops in computing, 1995, 264-278.

[Ram2] Ramanujam, R., "Local knowledge assertions in a changing world", in *Proc. Theoretical Aspects of Rationality and Knowledge*, Morgan Kaufmann, 1996, 1-17.

[Ram3] Ramanujam, R., "Locally linear time temporal logic", in *Proc. IEEE Logic in Computer Science*, 1996, 118-127.

[Var] Vardi, M.Y., "Verification of open systems", Invited talk in *Proc. of FST & TCS '97 in LNCS 1346*, Kharagpur, 1997, 250-266.

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

Appendix: Proof of Completeness

Here we prove the main result, that all regular behaviours on a distributed alphabet can be accepted by Assumption-Compatible systems.

Theorem 6.1. $Reg_\Sigma \subseteq \mathcal{L}(ACS_{\widetilde{\Sigma}})$.

We first define an automaton for the given regular language where the states are distributed but the transitions are globally specified hence it is not locally presented. 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.

6.1 Distributed state automaton

A **D**istributed **S**tate **A**utomaton(**DSA**) on $\widetilde{\Sigma}$ is a tuple $\mathcal{A} = (A_1, \ldots, A_n, \longrightarrow_{\mathcal{A}}, F)$, where

1. for every $i \in Loc, A_i = (Q_i, \longrightarrow_{\mathcal{A}i}, s_i^0)$ is the i-th TS on Σ_i,
2. $\widehat{Q} = \Pi_{i \in Loc} Q_i$ is the state space of $\mathcal{A}$,
3. $(s_1^0, \cdots, s_n^0)$ is the initial state,
4. $F \subseteq Q$ is the set of final states, and
5. $\longrightarrow_{\mathcal{A}} \subseteq (Q \times \Sigma \times Q)$ satisfies the following condition:
   if $((p_1, p_2, \ldots, p_n), a, (q_1, q_2, \ldots, q_n)) \in \longrightarrow_{\mathcal{A}}$ then
   (a) for all $i \in loc(a), (p_i, a, q_i) \in \longrightarrow_{\mathcal{A}i}$, and
   (b) for all $j \notin loc(a), p_j = q_j$.

Notice that the transition relation $\longrightarrow_{\mathcal{A}}$ satisfies only one half of the asynchrony condition. This is the crucial difference between DSA's and locally presented systems. In DSA's global transitions are given *a priori* while in locally presented systems, global transitions for the product are *constructed* by the asynchrony condition. We emphasize that DSA's are intended only as intermediate representation for a given regular language. We believe that this gives some structure to both the construction of the ultimate locally presented automaton and the proofs needed to show language equivalence. (We will continue refer to condition (5) above as the asynchrony condition.)
When $(\overline{p}, a, \overline{q}) \in \longrightarrow_{\mathcal{A}}$ we write it as $\overline{p} \xrightarrow{a}_{\mathcal{A}} \overline{q}$. $\mathcal{A}$ is *deterministic* if $\longrightarrow_{\mathcal{A}}$ is a deterministic relation.

6.2 Properties of DSA

Fix a *deterministic* DSA $\mathcal{A}$ over $\widetilde{\Sigma}$. Recall the projection operator $\lceil : (\Sigma^* \times Loc) \to \Sigma^*$.

$$x\lceil i = \begin{cases} \epsilon & \text{if } x = \epsilon \\ y\lceil i & \text{if } x = ya \text{ and } i \notin loc(a) \\ (y\lceil i)a & \text{if } x = ya \text{ and } i \in loc(a) \end{cases}$$

Recall also that for a non-empty string $x = a_1 \cdots a_k$, $last(x) \stackrel{\text{def}}{=} a_k$.
Now we define $\Downarrow : (\Sigma^* \times Loc) \to \Sigma^*$ as follows.

Definition 6.1 $x \Downarrow i \stackrel{\text{def}}{=} \begin{cases} \epsilon \textit{ if } x\lceil i = \epsilon \\ y \textit{ such that } \exists u \in \Sigma^* : x = yu, \\ \quad last(y) \in \Sigma_i \textit{ and } u\lceil i = \epsilon \\ \qquad \textit{if } x\lceil i \neq \epsilon. \end{cases}$

$x \Downarrow i$ 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 $abcaac \Downarrow 1 = abcaac$.
We make a couple of simple observations about $\Downarrow$. Both of them follow from the maximality of $x \Downarrow i$ in x.

1. For all $x \in \Sigma^*, i,j \in Loc$, if $x \Downarrow i$ is a prefix of $x \Downarrow j$ then $x \Downarrow i = (x \Downarrow j) \Downarrow i$.
2. For all $x \in \Sigma^*, i,j \in Loc$, if $x \Downarrow i = ya$ and $x \Downarrow j = y'a$, then $x \Downarrow j = x \Downarrow i$ and hence $y = y'$.

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

Definition 6.2 *The* events *associated with strings over* Σ *are defined inductively as follows:* $event(x) \stackrel{\text{def}}{=} \begin{cases} ((\epsilon)_{\mathcal{A}}, \epsilon, (\epsilon)_{\mathcal{A}}) & \textit{if } x = \epsilon \\ ((y)_{\mathcal{A}}, a, (ya)_{\mathcal{A}}) & \textit{if } x = ya \end{cases}$

An event, say $event(x)$, is called an i-event if either $x = \epsilon$ or $x = ya$ and $a \in \Sigma_i$. In the DSA $\mathcal{A}$ there is a precedence relation among the events associated with strings. The definition below says when an i-event precedes a j-event.

Definition 6.3 *$event(u)$ i-$\triangleright$-j $event(v)$ iff there is an $r \in \Sigma^*$ such that $r \Downarrow i$ is a prefix of $r \Downarrow j$, $event(u) = event(r \Downarrow i)$ and $event(v) = event(r \Downarrow j)$.*

It is clear from this and definition of $\Downarrow$ that $event(u)$ i-$\triangleright$-j $event(v)$ iff there is an $r \in \Sigma^*$ such that $r = r \Downarrow j$, $event(v) = event(r)$ and $event(u) = event(r \Downarrow i)$.
We prove 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 6.4 *For all $x \in \Sigma^*$, $(x \Downarrow i)_{\mathcal{A}}[i] = (x)_{\mathcal{A}}[i]$.*

Proposition 6.5 *Let $i, j \in loc(a)$. $event(xa)$ i-$\triangleright$-j $event(ya)$ implies $(x)_{\mathcal{A}} = (y)_{\mathcal{A}}$.*

Proof: Suppose $event(xa)$ i-$\triangleright$-j $event(ya)$. By definition, there exists an $r \in \Sigma^*$ such that $event(r \Downarrow i) = event(xa)$ and $event(r \Downarrow j) = event(ya)$ and $r \Downarrow i$ is a prefix of $r \Downarrow j$. (See Fig. 11).
From the definition of *event*, we have

- $((x)_{\mathcal{A}}, a, (xa)_{\mathcal{A}}) = ((r')_{\mathcal{A}}, c, (r'c)_{\mathcal{A}})$, where $r \Downarrow i = r'c$, and
- $((y)_{\mathcal{A}}, a, (ya)_{\mathcal{A}}) = ((r'')_{\mathcal{A}}, d, (r''d)_{\mathcal{A}})$, where $r \Downarrow j = r''d$.

From these conditions, we get $c = d = a$. From the observation about $\Downarrow$, we immediately get that $r \Downarrow i = r \Downarrow j$, or, equivalently, $r'a = r''a$. This means $r' = r''$ and therefore $(x)_{\mathcal{A}} = (r')_{\mathcal{A}} = (r'')_{\mathcal{A}} = (y)_{\mathcal{A}}$. ∎

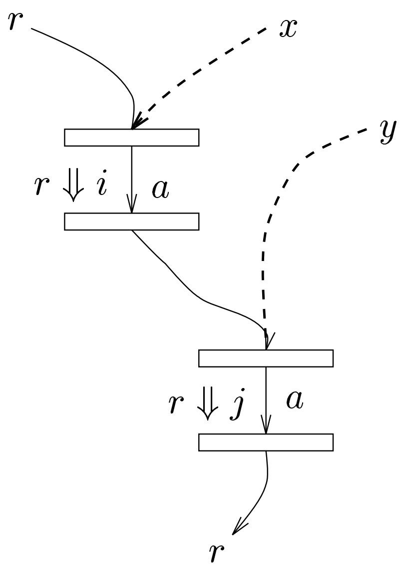


Fig. 11. $event(xa)$ i-⊳-j $event(ya)$

Definition 6.6 *A DSA $\mathcal{A}$ is* **stutter-free** *if for every $\overline{p}, \overline{q} \in \widehat{Q}$,*
$$\overline{p} \xrightarrow{a}_{\mathcal{A}} \overline{q} \text{ implies for all } i \in loc(a), \overline{p}[i] \neq \overline{q}[i].$$

Informally, this means that transitions change the local states of all participating agents.

Proposition 6.7 *Suppose $\mathcal{A}$ is stutter-free. Let $u, w \in \Sigma^*$, $a \in \Sigma_i$ and $k \in Loc$. Then $event(ua)$ i-⊳-k $event(w)$ and $event(u \Downarrow i)$ i-⊳-k $event(w)$ can not both be true simultaneously.*

Proof. Suppose that $event(ua)$ i-⊳-k $event(w)$ and $event(u \Downarrow i)$ i-⊳-k $event(w)$ simultaneously.

Claim. Let $x, y, z \in \Sigma^*$, $i, k \in Loc$ and suppose both $event(x)$ i-⊳-k $event(z)$ and $event(y)$ i-⊳-k $event(z)$. Then, $(x)_{\mathcal{A}}\,[i] = (y)_{\mathcal{A}}\,[i]$.

Assume the claim. Then, $(ua)_{\mathcal{A}}\,[i] = (u \Downarrow i)_{\mathcal{A}}\,[i] = (u)_{\mathcal{A}}\,[i]$, thus violating the stutter-free condition. But $\mathcal{A}$ is given to be stutter-free, hence we get a contradiction, thus proving the proposition.
Proof of claim: Suppose $event(x)$ i-⊳-k $event(z)$. Then, there is an $r \in \Sigma^*$ such that $r = r \Downarrow k$, $event(z) = event(r)$ and $event(x) = event(r \Downarrow i)$. Hence, from definition of event, we have $(r)_{\mathcal{A}} = (z)_{\mathcal{A}}$ and $(x)_{\mathcal{A}} = (r \Downarrow i)_{\mathcal{A}}$.
Therefore, $(x)_{\mathcal{A}}\,[i] = (r \Downarrow i)_{\mathcal{A}}\,[i]$ and using Proposition 6.4 we get $(x)_{\mathcal{A}}\,[i] = (r)_{\mathcal{A}}\,[i] = (z)_{\mathcal{A}}\,[i]$.
Arguing similarly, we have $(y)_{\mathcal{A}}\,[i] = (z)_{\mathcal{A}}\,[i]$. Therefore, $(x)_{\mathcal{A}}\,[i] = (y)_{\mathcal{A}}\,[i]$.
This proves the claim and the proposition. □

Definition 6.8 $\mathcal{A}$ *is* **four-alternation-free** *if for all* $i, k \in Loc$*, for* a, c *in* $\Sigma_i \setminus \Sigma_k, b, d \in \Sigma_k \setminus \Sigma_i$ *and* $r \in \Sigma^*$*,* $(r \Downarrow i = r_1 a,\ r_1 \Downarrow k = r_2 b,\ r_2 \Downarrow i = r_3 c,\ r_3 \Downarrow k = r_4 d$ *and* $r_4 \Downarrow i = r_5)$ *implies* $(r_5)_{\mathcal{A}} \neq (r)_{\mathcal{A}}$ *.*

The *four* in the definition is because i-transitions and k-transitions alternate four times in the loop. (See Fig. 12).
This definition says that a particular kind of "bad" loops are not present in a four-alternation-free DSA. This characteristic of a DSA helps in synthesizing a behaviour preserving ACS as we will see in the next section.[1]

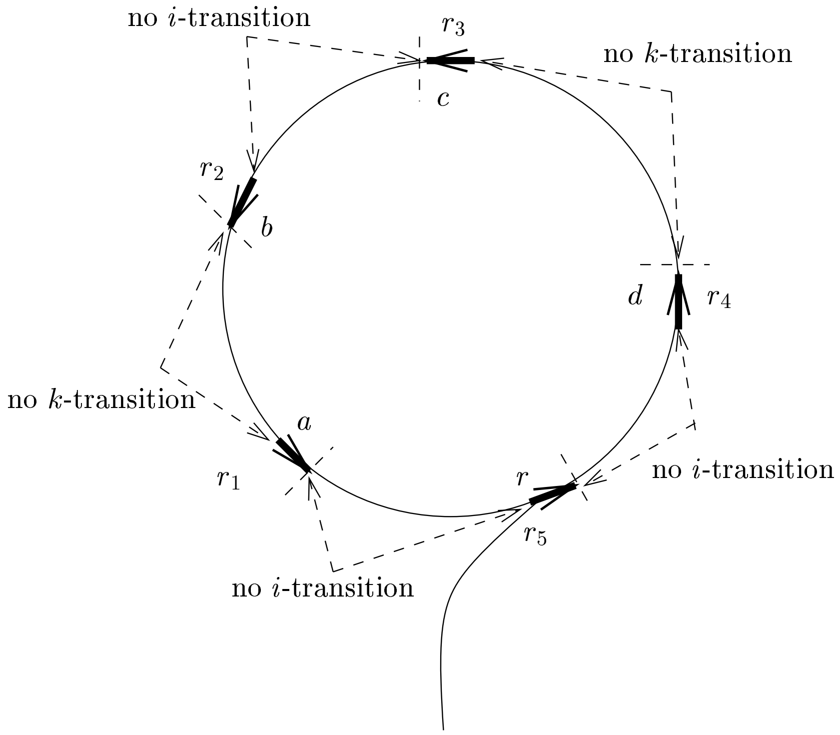


Fig. 12. When $\mathcal{A}$ is four-alternation-free, such loops are not present in $\mathcal{A}$.

Proposition 6.9 *Suppose DSA* $\mathcal{A}$ *is four-alternation-free. Let* $u, y \in \Sigma^*$*,* $a \in \Sigma$*,* $i \in loc(a)$*,* $k \notin loc(a)$ *such that*

$$(event(ua)\ i\text{-}\triangleright\text{-}k\ event(y)\ and\ event(y)\ k\text{-}\triangleright\text{-}i\ event(u \Downarrow i)).$$

[1] We really do not have any intuition to offer as to why only "four" works. This seems to be connected to some kind of minimal unfolding of the DSA necessary for the synthesis.

Then, $(y)_{\mathcal{A}}[k] = (u)_{\mathcal{A}}[k]$.

Proof. By definition of $\dashrightarrow$, we have

1. $\exists r \in \Sigma^*$ such that $event(ua) = event(r \Downarrow i)$, $event(y) = event(r \Downarrow k)$ and $r \Downarrow i$ is a prefix of $r \Downarrow k$. Hence,

$$r \Downarrow i = (r \Downarrow k) \Downarrow i, (ua)_{\mathcal{A}} = (r \Downarrow i)_{\mathcal{A}}, \text{ and } (y)_{\mathcal{A}} = (r \Downarrow k)_{\mathcal{A}}. \quad (1)$$

2. $\exists r' \in \Sigma^*$ such that $event(y) = event(r' \Downarrow k)$, $event(u \Downarrow i) = event(r' \Downarrow i)$ and $r' \Downarrow k$ is a prefix of $r' \Downarrow i$. Hence,

$$r' \Downarrow k = (r' \Downarrow i) \Downarrow k,\ (y)_{\mathcal{A}} = (r' \Downarrow k)_{\mathcal{A}} \text{ and} (u \Downarrow i)_{\mathcal{A}} = (r' \Downarrow i)_{\mathcal{A}}. \quad (2)$$

Claim. $u \Downarrow k$ is a prefix of $u \Downarrow i$.

Assume the claim. Then, $u \Downarrow k = (u \Downarrow i) \Downarrow k$. (recall the simple observation we made about $\Downarrow$). Then,

$$\begin{aligned}
(u)_{\mathcal{A}}[k] &= (u \Downarrow k)_{\mathcal{A}}[k] && \text{from Proposition 6.4}\\
&= ((u \Downarrow i) \Downarrow k)_{\mathcal{A}}[k] && \text{by the claim}\\
&= (u \Downarrow i)_{\mathcal{A}}[k] && \text{from Proposition 6.4 again}\\
&= (r' \Downarrow i)_{\mathcal{A}}[k] && \text{from (2) above}\\
&= ((r' \Downarrow i) \Downarrow k)_{\mathcal{A}}[k] && \text{Proposition 6.4 yet again}\\
&= (r' \Downarrow k)_{\mathcal{A}}[k] && \text{from (2) above}\\
&= (y)_{\mathcal{A}}[k] && \text{from (2) above}
\end{aligned}$$

And thus we prove the proposition.
Proof of claim: Suppose, $u \Downarrow k$ is not a prefix of $u \Downarrow i$. This implies $(u \Downarrow i)$ is a prefix of $u \Downarrow k$ or, in other words,

$$u \Downarrow i = (u \Downarrow k) \Downarrow i. \quad (3)$$

We show that this leads to a contradiction. See Fig. 13.

– Because of (3), u can be written as $x_1 c x_2 d x_3$, for some $x_1 \in \Sigma^*$, x_2 in $(\Sigma \setminus \Sigma_i)^*, x_3 \in (\Sigma \setminus (\Sigma_i \cup \Sigma_k))^*$ where $u \Downarrow i = x_1 c$, $c \in \Sigma_i$, $d \in \Sigma_k \backslash \Sigma_i$.
– Since $r \Downarrow i = (r \Downarrow k) \Downarrow i$, we can write $r \Downarrow k$ as $(r \Downarrow i) \cdot x_4 b$, for some x_4 in $(\Sigma \setminus \Sigma_i)^*$ and $b \in \Sigma_k \backslash \Sigma_i$, by (1) above.
– Finally, since $r' \Downarrow k = (r' \Downarrow i) \Downarrow k$, we can write $r' \Downarrow i$ as $(r' \Downarrow k) \cdot x_5 c$, for some $x_5 \in (\Sigma \setminus \Sigma_k)^*$ and $c \notin \Sigma_i$, by (2) above.

Note that

$$\begin{aligned}
(x_1 c)_{\mathcal{A}} &= (u \Downarrow i)_{\mathcal{A}} && \text{from item 1 above}\\
&= (r' \Downarrow i)_{\mathcal{A}} && \text{since } event(u \Downarrow i) = event(r' \Downarrow i)\\
&= (r' \Downarrow k \cdot x_5 c)_{\mathcal{A}} && \text{from item 3 above.}
\end{aligned}$$

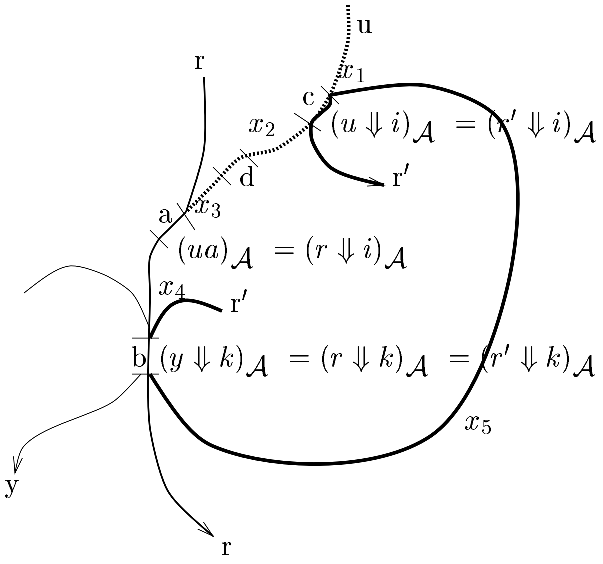


Fig. 13. The case $u \Downarrow k \neq (u \Downarrow i) \Downarrow k$.

By (1) and (2) above, $(r \Downarrow k)_{\mathcal{A}} = (y \Downarrow k)_{\mathcal{A}} = (r' \Downarrow k)_{\mathcal{A}}$.
Hence, we get

$$\begin{aligned}
(x_1 c)_{\mathcal{A}} &= (r \Downarrow k \cdot x_5 c)_{\mathcal{A}} && \text{from preceding observation} \\
&= ((r \Downarrow i \cdot x_4 b) \cdot x_5 c)_{\mathcal{A}} && \text{from item 2} \\
&= (((ua) \cdot x_4 b) \cdot x_5 c)_{\mathcal{A}} && \text{from (1) above} \\
&= (x_1 c \cdot x_2 d \cdot x_3 a \cdot x_4 b \cdot x_5 c)_{\mathcal{A}} && \text{from item 1.}
\end{aligned}$$

Let $y_5 = x_1 c$ and $y = x_1 c \cdot x_2 d \cdot x_3 a \cdot x_4 b \cdot x_5 c$. Then what we have got here is

$$(y)_{\mathcal{A}} = (y_5)_{\mathcal{A}} . \tag{4}$$

Also, one verifies that

1. $y \Downarrow i = x_1 c \cdot x_2 d \cdot x_3 a \cdot x_4 b \cdot x_5 c$, since $c \in \Sigma_i$. Let $y_1 = x_1 c \cdot x_2 d \cdot x_3 a \cdot x_4 b \cdot x_5$.
2. $y_1 \Downarrow k = x_1 c \cdot x_2 d \cdot x_3 a \cdot x_4 b$, since $b \in \Sigma_k$ and $x_5 \in (\Sigma \setminus \Sigma_k)^*$. Let $y_2 = x_1 c \cdot x_2 d \cdot x_3 a \cdot x_4$.
3. $y_2 \Downarrow i = x_1 c \cdot x_2 d \cdot x_3 a$, since $a \in \Sigma_i$ and $x_4 \in (\Sigma \setminus \Sigma_i)^*$. Let $y_3 = x_1 c \cdot x_2 d \cdot x_3$.
4. $y_3 \Downarrow k = x_1 c \cdot x_2 d$, since $d \in \Sigma_k$ and $x_3 \in (\Sigma \setminus \Sigma_k)^*$. Let $y_4 = x_1 c \cdot x_2$.
5. Finally, $y_4 \Downarrow i = x_1 c = y_5$, since $c \in \Sigma_i$ and $x_2 \in (\Sigma \setminus \Sigma_i)^*$.

But since $\mathcal{A}$ is four-alternation-free and y and y_5 satisfy the assumptions of for four-alternation-freeness, $(y)_{\mathcal{A}} \neq (y_5)_{\mathcal{A}}$. Hence, from (4) we get a contradiction. Thus we settle the claim and hence the proposition. $\square$

6.3 From DSA to ACS

The idea behind introducing the notions of *stutter-freeness* and *four-alternation-freeness* is that DSA's with these properties facilitate construction of equivalent ACS's. We demonstrate this construction first. Then, given a regular language, we will show how to construct a DSA with the above-mentioned properties. Thus we will get an ACS for the given regular language.
The following theorem establishes the required correspondence between DSA's and ACS's over $\widetilde{\Sigma}$.

Theorem 6.10 *Let $\mathcal{A}$ be a stutter-free and four-alternation-free deterministic DSA on $\widetilde{\Sigma}$. Then, there exists an ACS $\widetilde{M}$ on $\widetilde{\Sigma}$ such that $L(\mathcal{A}) = L(\widetilde{M})$.*

Proof. We first define some sets that will be used to construct the commit alphabet and local states of the ACS.

Definition 6.11 *For $i \in Loc, S_i \stackrel{\text{def}}{=} \{event(x) \mid x \in \Sigma^*, x \Downarrow i = x\}$.*

The commit alphabet $\mathcal{C}$ is defined as follows: for all $i \in Loc$, $\mathcal{C}_i = 2^{S_i}$ and $\lambda \preceq_i \mu$ iff $\lambda \supseteq \mu$.
Now we construct an ACS $\widetilde{M} = (M_1, \cdots, M_n, < f_1, \cdots, f_n >, F)$ over $(\widetilde{\Sigma}, \widetilde{\mathcal{C}})$ from the DSA $\mathcal{A}$ as follows.
For each $i \in Loc$, i-local TS $M_i = (Q_i, \stackrel{i}{\longrightarrow}, q_i^0)$ where

- $Q_i = S_i$,
- $q_i^0 = event(\epsilon)$, and
- $p \stackrel{a}{\longrightarrow}_i q$ iff there is a $u \in \Sigma^*$ such that $p = event(u \Downarrow i)$ and $q = event(ua \Downarrow i)$.
- Local assumption maps f_i are defined as follows.
  For every $event(x) \in Q_i$, $f_i(event(x))(j) = \{event(y \Downarrow j) \mid event(y \Downarrow i) = event(x)\}$.

Lastly, the set of final states is defined to be
$F = \{(event(x \Downarrow 1), \ldots, event(x \Downarrow n)) \mid x \in L(\mathcal{A})\}$.
Let $(\widehat{M} = (\widehat{Q}, \longrightarrow, (q_1^0, \cdots, q_n^0)), F)$ be the compatible product automaton of $\widetilde{M}$ where $\widehat{Q}$ is the set of all compatible states. Since $L(\widetilde{M}) \stackrel{\text{def}}{=} L(\widehat{M}, F)$, we show in the following that $L(\mathcal{A}) = L(\widehat{M}, F)$.
We first observe some properties of the assumption maps.
1. For all $i \in Loc$, $f_i(event(x))(i) = \{event(x)\}$.

Proof. For $event(x) \in Q_i$, $x \Downarrow i = x$. Hence, by construction,

$$f_i(event(x))(i) = \{event(y \Downarrow i) \mid event(y \Downarrow i) = event(x)\} = \{event(x)\}.$$

□

An immediate consequence is the following.
2. For all $i, j \in Loc$,

$$f_i(event(x))(j) \preceq_j f_j(event(y))(j) \text{ iff } \{event(y)\} \subseteq f_i(event(x))(j).$$

3. For all $i, j \in Loc$ and $q \in Q_i$,

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

Proof. Let $s \in f_i(q)(j)$. Then, there is an $x \in \Sigma^*$ such that $s = event(x \Downarrow j)$ and $q = event(x \Downarrow i)$. Since either $x \Downarrow j \preceq x \Downarrow i$ or vice-versa, we have either s j-$\triangleright$-i q or q i-$\triangleright$-j s and hence $s \in RHS$.
The other inclusion follows from the definition of assumption maps. □
We now observe some properties of the compatible product.
1. For all $x \in \Sigma^*$, $(event(x \Downarrow 1), \ldots, event(x \Downarrow n))$ is compatible.

Proof. We need to show that $f_i(event(x \Downarrow i))(j) \preceq_j f_j(event(x \Downarrow j))(j)$ for all $i, j \in Loc$. This is equivalent to showing $event(x \Downarrow j) \in f_i(event(x \Downarrow i))(j)$ (by the observation about assumption maps above).
By definition,

$$f_i(event(x \Downarrow i))(j) = \{event(y \Downarrow j) \mid event(y \Downarrow i) = event(x \Downarrow i)\}.$$

Therefore, the property is proved by setting y to be x. □
2. For all $x \in \Sigma^*$,

$$(event(x \Downarrow 1), \ldots, event(x \Downarrow n)) \stackrel{a}{\Longrightarrow} (event(xa \Downarrow 1), \ldots, event(xa \Downarrow n)).$$

Proof. Notice first that the given states are compatible by the preceding result. One just has to check that the asynchrony condition holds for the transition.

1. For $i \notin loc(a), x \Downarrow i = xa \Downarrow i$, hence, $event(x \Downarrow i) = event(xa \Downarrow i)$.
2. From the definition of $\stackrel{i}{\longrightarrow}$, it directly follows that for all $i \in loc(a)$, $event(x \Downarrow i) \stackrel{a}{\longrightarrow}_i event(xa \Downarrow i)$, and we are done. □

In the light of the above, by a simple induction on the length of strings we get for all $x \in \Sigma^*$, $(event(\epsilon), \ldots, event(\epsilon)) \stackrel{x}{\Longrightarrow} (event(x \Downarrow 1), \ldots, event(x \Downarrow n))$.
These observations immediately give us the following lemma proving one direction of Theorem 6.10.

Lemma 6.12 $L(\mathcal{A}) \subseteq L(\widetilde{M})$.

Proof. Let $x \in L(\mathcal{A})$. Then, $(event(x \Downarrow 1), \ldots, event(x \Downarrow n)) \in F$ by construction. Also, from the observation above,

$$event(\epsilon), \ldots, event(\epsilon)) \stackrel{x}{\Longrightarrow} (event(x \Downarrow 1), \ldots, event(x \Downarrow n)).$$

Hence, $x \in L(\widetilde{M})$.

We now prove the more difficult direction, which uses the special properties of the given DSA.

Lemma 6.13 $L(\widetilde{M}) \subseteq L(\mathcal{A})$.

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

Claim. $\forall x \in \Sigma^*, (event(\epsilon), \ldots, event(\epsilon)) \stackrel{x}{\Longrightarrow} (event(z_1), \ldots, event(z_n))$ implies for all $i \in Loc$, $event(z_i) = event(x \Downarrow i)$.

Assuming the claim, for all $i \in Loc, event(x \Downarrow i) = event(u \Downarrow i)$. This implies for all $i \in Loc$, $(x \Downarrow i)_{\mathcal{A}} = (u \Downarrow i)_{\mathcal{A}}$, and hence $(x)_{\mathcal{A}}[i] = (u)_{\mathcal{A}}[i]$. So $(x)_{\mathcal{A}} = (u)_{\mathcal{A}}$. This means that both x and u lead to the same state in $\mathcal{A}$. Since $u \in L(\mathcal{A})$, and $\mathcal{A}$ is deterministic, $(u)_{\mathcal{A}} \in F_{\mathcal{A}}$. Therefore, $(x)_{\mathcal{A}} \in F_{\mathcal{A}}$ 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 $(event(\epsilon), \ldots, event(\epsilon)) \stackrel{ya}{\Longrightarrow} (event(z_1), \ldots, event(z_n))$. Then there is a state $(event(y_1), \ldots, event(y_n)) \in \widehat{Q}$ such that

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

By induction hypothesis

$$event(y_i) = event(y \Downarrow i) \text{ for all } i \in Loc\,. \tag{5}$$

We have to show that $event(z_i) = event(ya \Downarrow i)$ for all $i \in Loc$.
Case 1: $\underline{i \notin loc(a)}$. Then, $event(z_i) = event(y \Downarrow i)$, by asynchrony and induction hypothesis. Since $event(y \Downarrow i) = event(ya \Downarrow i)$ for $i \notin loc(a)$, we are done.
Case 2: $\underline{i \in loc(a)}$. Since $ya \Downarrow i = ya$, we have to show $event(z_i) = event(ya)$.
By asynchrony, for all $j \in loc(a)$, $event(y_i) \stackrel{a}{\longrightarrow}_i event(z_i)$. Hence, from the construction of $\stackrel{i}{\longrightarrow}$, we have, for all $j \in loc(a)$ a $u_j \in \Sigma^*$ such that

$$event(y \Downarrow j) = event(u_j \Downarrow j), \text{and} \tag{6}$$
$$event(u_j a) = event(z_j). \tag{7}$$

So it suffices to show that $event(u_i a) = event(ya)$. From the definition of *event*, it suffices to show that $(u_i)_{\mathcal{A}} = (y)_{\mathcal{A}}$ or equivalently,

$$\boxed{\text{(to show) for all } k \in Loc, (u_i)_{\mathcal{A}}[k] = (y)_{\mathcal{A}}[k].}$$

We do this separately for two cases: $k \notin loc(a)$ and $k \in loc(a)$.
Let $\underline{k \in loc(a)}$. Because the state $(event(z_1), \ldots, event(z_n))$ is compatible, we know that $event(z_i) \in f_k(event(z_k))(i)$. Hence, $event(u_i a) \in f_k(event(u_k a))(i)$ by (7).
From the property of assumption maps, it follows that either

$$event(u_i a) \text{ i-}\triangleright\text{-k } event(u_k a) \text{ or } event(u_k a) \text{ k-}\triangleright\text{-i } event(u_i a).$$

In both the cases, from Prop. 6.5, $(u_i)_{\mathcal{A}} = (u_k)_{\mathcal{A}}$. Hence $(u_i)_{\mathcal{A}}[k] = (u_k)_{\mathcal{A}}[k]$. By (6) and using Proposition 6.4 we get, for all $m \in loc(a), (u_m)_{\mathcal{A}}[m] = (y)_{\mathcal{A}}[m]$. Therefore, $(u_i)_{\mathcal{A}}[k] = (y)_{\mathcal{A}}[k]$.
Now, let $\underline{k \notin loc(a)}$.

1. By compatibility of $(event(z_1), \ldots, event(z_n))$, $event(z_i) \in f_k(event(z_k))(i)$. Therefore, by Property of assumption maps,
   [either $event(z_i)$ i-▷-k $event(z_k)$ or $event(z_k)$ k-▷-i $event(z_i)$].
2. By compatibility of $(event(y_1), \ldots, event(y_n))$, $event(y_i) \in f_k(event(y_k))(i)$. Therefore, by Property of assumption maps
   [either $event(y_i)$ i-▷-k $event(z_k)$ or $event(z_k)$ k-▷-i $event(y_i)$].

We consider the possible cases. Remember that we are considering the case when $\underline{i \in loc(a)}$ and $\underline{k \notin loc(a)}$. Hence, in the following, $event(z_k) = event(y_k)$.
Case event$(\mathbf{z_k})$ k-▷-i event$(\mathbf{z_i})$: There is an $r \in \Sigma^*$ such that $r \Downarrow i = r$, $event(z_i) = event(r)$ and $event(z_k) = event(r \Downarrow k)$.
From this and (7), we get $event(u_i a) = event(z_i) = event(r)$ and from asynchrony, we get $event(y_k) = event(z_k) = event(r \Downarrow k)$.
Then

$$\begin{aligned}
(y)_{\mathcal{A}}\ [k] &= (y \Downarrow k)_{\mathcal{A}}\ [k] && \text{by Proposition 6.4} \\
&= (y_k)_{\mathcal{A}}\ [k] && \text{by induction hypothesis (5)} \\
&= (r \Downarrow k)_{\mathcal{A}}\ [k] && \text{since } event(y_k) = event(r \Downarrow k) \\
&= (r)_{\mathcal{A}}\ [k] && \text{by Proposition 6.4} \\
&= (u_i a)_{\mathcal{A}}\ [k] && \text{since } event(u_i a) = event(r) \\
&= (u_i)_{\mathcal{A}}\ [k] && \text{since } k \notin loc(a),
\end{aligned}$$

which is the required result.

Case e$vent(z_i)$ i-▷-k $event(z_k)$ and **e**$vent(y_i)$ i-▷-k $event(z_k)$:
We have $event(z_i) = event(u_i a)$ by (7) and $event(y_i) = event(u_i \Downarrow i)$ by (5) and (6). Hence, for this case,

$$event(u_i a) \text{ i-▷-k } event(z_k) \text{ and } event(u_i \Downarrow i) \text{ i-▷-k } event(z_k).$$

Then, since we assume that $\mathcal{A}$ is *stutter-free*, by Prop. 6.7 this case is not possible.
Case e$vent(z_i)$ i-▷-k $event(z_k)$ and **e**$vent(z_k)$ k-▷-i $event(y_i)$
Recall that for the case under consideration(namely, $i \in loc(a), k \notin loc(a)$),

- $event(z_i) = event(u_i a)$(from (7)),
- $event(z_k) = event(y_k)$(by asynchrony) and
- $event(y_i) = event(u_i \Downarrow i)$(from (5) and (6)).

Hence, $event(u_i a)$ i-▷-k $event(y_k)$ and $event(y_k)$ k-▷-i $event(u_i \Downarrow i)$. Then, since the DSA $\mathcal{A}$ is four-alternation-free, from Proposition 6.9, we have $(u_i)_{\mathcal{A}}\ [k] = (y_k)_{\mathcal{A}}\ [k]$. Then, from (5), applying Proposition 6.4, we finally get $(u_i)_{\mathcal{A}}\ [k] = (y)_{\mathcal{A}}\ [k]$.
Thus we have proved that for all locations $k \in Loc$, $(u_i)_{\mathcal{A}}\ [k] = (y)_{\mathcal{A}}\ [k]$. Thereby we prove the claim and the lemma. □
Lemmas 6.12 and 6.13 prove Theorem 6.10, stated below again.
Theorem 6.10 *Let $\mathcal{A}$ be a stutter-free and four-alternation-free deterministic DSA on $\widetilde{\Sigma}$. Then, there exists an ACS $\widetilde{M}$ on $\widetilde{\Sigma}$ such that $L(\mathcal{A}) = L(\widetilde{M})$.*

6.4 Regular languages and DSA

Because of Theorem 6.10, 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 6.14 *Let the class of deterministic DSA's over $\widetilde{\Sigma}$ that are stutter-free and four-alternation-free be denoted as FDSA. Then $\mathcal{L}(FDSA_{\widetilde{\Sigma}}) = Reg_{\Sigma}$.*

Proof. Since the global automaton of a DSA (hence that of an FDSA) is a finite state automaton over Σ, languages accepted by DSA's are regular over Σ. Thus we get the easy direction.
For the other direction, let $L \in Reg_{\Sigma}$. We construct a deterministic FDSA $\mathcal{A}$ $\widetilde{\Sigma}$ such that $L(\mathcal{A}\ ) = L$ and $\mathcal{A}$ has the required properties.

Definition 6.15 (A labeling scheme) *Let $l : \Sigma^* \to (Loc \times Loc \to \{0,1,2\})$ 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*

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

(When for some $x \in \Sigma^*$, $l(x) = C$, we use the notation C_{ij} to denote $C(i,j)$). Note that, by definition, $l(x)(i,i) = 0$ for all x and $i \in Loc$.

Example 6.1. Let $\widetilde{\Sigma} = \{\{a,c\},\{b,c\}\}$. Then,

$$l(\epsilon) = \begin{pmatrix} 0\,0 \\ 0\,0 \end{pmatrix},\ l(a) = \begin{pmatrix} 0\,1 \\ 0\,0 \end{pmatrix},\ l(ac) = \begin{pmatrix} 0\,0 \\ 0\,0 \end{pmatrix},\ l(ab) = \begin{pmatrix} 0\,1 \\ 2\,0 \end{pmatrix},\ l(aba) = \begin{pmatrix} 0\,0 \\ 2\,0 \end{pmatrix},$$
$$l(abab) = \begin{pmatrix} 0\,0 \\ 1\,0 \end{pmatrix}.$$

Proposition 6.16 *For all $x \in \Sigma^*, i,j \in Loc$, $l(x)(i,j) = l(x \Downarrow i)(i,j)$.*

Proof: (by induction on $|x|$)
When $x = \epsilon$, $l(x)(i,j) = l(\epsilon)(i,j) = l(x \Downarrow i)(i,j)$. Suppose $x = ya$. By induction hypothesis, for all $i, j \in Loc, l(y)(i,j) = l(y \Downarrow i)(i,j)$.

Case 6.1. $a \in \Sigma_i$. $x = x \Downarrow i$ and hence the proposition holds.

Case 6.2. $a \notin \Sigma_i$. Then, by definition of l, $l(y)(i,j) = l(ya)(i,j) = l(x)(i,j)$. Since $y \Downarrow i = ya \Downarrow i = x \Downarrow i$, using induction hypothesis we have $l(x)(i,j) = l(y)(i,j) = l(y \Downarrow i)(i,j) = l(x \Downarrow i)(i,j)$. ■

Proposition 6.17 *If $x \Downarrow j$ is a proper prefix of $x \Downarrow i$ then $l(x)(i,j) = l(x)(j,i) + 1\ mod\ 3$.*

Proof: By the preceding result,

$$l(x)(i,j) = l(x \Downarrow i)(i,j) \text{ and } l(x)(j,i) = l(x \Downarrow j)(j,i).$$

By the given condition, $x \Downarrow i = (x \Downarrow j) \cdot ua$, for some $a \in (\Sigma_i \setminus \Sigma_j)$ and $u \in \Sigma^*$ such that $u\lceil j = \epsilon$. Hence, $x \Downarrow j = (x \Downarrow j \cdot u) \Downarrow j$. Then from previous observation we can derive:

$$\begin{aligned} l(x)(i,j) &= l(x \Downarrow i)(i,j), \\ &= l(x \Downarrow j \cdot u)(j,i) + 1 \ mod\ 3 \\ &= l((x \Downarrow j \cdot u) \Downarrow j)(j,i) + 1 \ mod\ 3 \\ &= l(x \Downarrow j)(j,i) + 1 \ mod\ 3 \\ &= l(x)(j,i) + 1 \ mod\ 3 \end{aligned}$$

■

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 6.18 *Let* $x\backslash L \stackrel{\text{def}}{=} \{y \in \Sigma^* \mid xy \in L\}$. *For all* $i \in Loc$, $x \equiv_i y$ *iff* $l(x \Downarrow i) = l(y \Downarrow i)$ *and* $(x \Downarrow i)\backslash L = y \Downarrow i \backslash L$.

Lemma 6.1. *If (for all* $x, y \in \Sigma^*$ *and for all* $i \in Loc$, $x \equiv_i y$*) then* $x\backslash L = y\backslash L$.

Proof:

Case 6.3. $x = \epsilon$ and $y = \epsilon$ Lemma holds trivially.

Case 6.4. $x = \epsilon$ and $y \neq \epsilon$ Let $j \in loc(last(y))$. Then $y \Downarrow j = y$. Also $x \Downarrow j = \epsilon = x$. By the assumption of the lemma, $x \equiv_j y$. By definition of $\equiv_j$, $x\backslash L = y\backslash L$.

Case 6.5. $x \neq \epsilon$ and $y \neq \epsilon$ Consider the following claim.

Claim. Let x, y be non-empty strings over Σ^*such that for all $k \in Loc$ $x \equiv_k y$. Let $last(x) = a$ and $last(y) = b$. Then $loc(a) \cap loc(b) \neq \emptyset$.

Assuming the claim, there is a $k \in Loc$ such that $k \in loc(last(x)) \cap loc(last(y))$. Hence, $x \Downarrow k = x$ and $y \Downarrow k = y$. Then, since $x \equiv_k y$ (by the antecedent of the lemma), by definition of $\equiv_k$, then, $x\backslash L = y\backslash L$.
Proof of claim: Suppose $loc(a) \cap loc(b) = \emptyset$. Take $i \in loc(a)$ and $j \in loc(b)$. Then, $x \Downarrow i = x$ and $y \Downarrow j = y$. Let $x \Downarrow j = x'$ and $y \Downarrow i = y'$. By our assumption, x'(resp y') is a proper prefix of x(resp. y).
Since $x \equiv_i y, l(x) = l(x \Downarrow i) = l(y \Downarrow i) = l(y')$. Let $l(x) = C = l(y')$. Then, by Proposition 6.17, $l(y) = C'$, where $C'_{ji} = C_{ij} + 1 \ mod\ 3$.
Further, since $x \equiv_j y, l(x') = l(x \Downarrow j) = l(y \Downarrow j) = l(y)$. Now $l(x') = C' = l(y)$. Again, by Proposition 6.17, $C_{ij} = C'_{ji} + 1 \ mod\ 3$.
From the previous paragraph, we then get, $C_{ij} = C_{ij} + 2 \ mod\ 3$, which is a contradiction. Thus we prove the claim and the lemma. ■
The labeling above is so designed as to construct a DSA that is four-alternation-free. In order to ensure that the DSA is also stutter-free, we introduce an obvious notion as below.

Definition 6.19 (i-parity) *Define i-parity : $\Sigma^* \to \{0,1\}$ as: for all $x \in \Sigma^*$,*

$$i\text{-}parity(x) \stackrel{\text{def}}{=} |x \lceil i| \ mod\ 2.$$

For example, let $\widetilde{\Sigma} = \{\{a,c\},\{b,c\}\}$. Then, 1-parity($a$) = 1, 2-parity($bab$) = 0 and 1-parity($abab$) = 0.
The following proposition regarding i-parity of strings follows easily from the definition of $\Downarrow$.

Proposition 6.20 *For all $x \in \Sigma^*$ and $i \in Loc$, i-parity(x) = i-parity($x \Downarrow i$). Therefore, for all $i \notin loc(a)$. i-parity(x) = i-parity(xa).*

Now we define, for each $i \in Loc$, equivalence classes that are to be the local states of the DSA.

Definition 6.21 *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).*

Let $[x]_i$ denote the equivalence class under $\sim_i$ containing x.
The following results follow directly from definition of $\sim_i$ and Lemma 6.1.

Corollary 6.22 *1. $[xa]_i = [x]_i$ for all $i \notin loc(a)$.*
2. If (for all $x, y \in \Sigma^*$ and for all $i \in Loc, x \sim_i y$) then $x \backslash L = y \backslash L$.

The local automata A_i of the constructed DSA $\mathcal{A}$ are defined as $(Q_i, \longrightarrow_{\mathcal{A}i}, s_i^0)$, where $Q_i = \{[x]_i \mid x \in \Sigma^*\}$, $s_i^0 = ([\epsilon]_i, 0)$ and $\longrightarrow_{\mathcal{A}i} = \{([x]_i, a, [xa]_i) \mid x \in \Sigma^*\}$.
Finally, $\mathcal{A} = (A_1, A_2, \ldots, A_n, \longrightarrow_{\mathcal{A}}, F)$, where
$\longrightarrow_{\mathcal{A}} = \{(([x]_1, [x]_2, \ldots, [x]_n), a, ([xa]_1, [xa]_2, \ldots, [xa]_n)) \mid x \in \Sigma^*\}$ and
$F = \{([x]_1, [x]_2, \ldots, [x]_n) \mid x \in L\}$.

Example 6.2. Let $\widetilde{\Sigma} = \{\{a\},\{b\}\}$. In Fig. 14 we construct a DSA for the language $(ab)^*$. The i-equivalences are computed from the table.

x	$l(x)$	1-parity(x)	y	$l(y)$	2-parity(y)
ϵ	$l(\epsilon) = \begin{pmatrix} 0\,0 \\ 0\,0 \end{pmatrix}$	0	ϵ	$l(\epsilon) = \begin{pmatrix} 0\,0 \\ 0\,0 \end{pmatrix}$	0
a	$l(a) = \begin{pmatrix} 0\,1 \\ 0\,0 \end{pmatrix}$	1	ab	$l(ab) = \begin{pmatrix} 0\,1 \\ 2\,0 \end{pmatrix}$	1
aba	$l(aba) = \begin{pmatrix} 0\,0 \\ 2\,0 \end{pmatrix}$	0	$(ab)^2$	$l((ab)^2) = \begin{pmatrix} 0\,0 \\ 1\,0 \end{pmatrix}$	0
$(ab)^2a$	$l((ab)^2a) = \begin{pmatrix} 0\,2 \\ 1\,0 \end{pmatrix}$	1	$(ab)^3$	$l((ab)^3) = \begin{pmatrix} 0\,2 \\ 0\,0 \end{pmatrix}$	1
$(ab)^3a$	$l((ab)^3a) = \begin{pmatrix} 0\,1 \\ 0\,0 \end{pmatrix}$	0	$(ab)^4$	$l((ab)^4) = \begin{pmatrix} 0\,1 \\ 2\,0 \end{pmatrix}$	0
$(ab)^4a$	$l((ab)^4a) = \begin{pmatrix} 0\,0 \\ 2\,0 \end{pmatrix}$	1	$(ab)^5$	$l((ab)^5) = \begin{pmatrix} 0\,0 \\ 1\,0 \end{pmatrix}$	1
$(ab)^5a$	$l((ab)^5a) = \begin{pmatrix} 0\,2 \\ 1\,0 \end{pmatrix}$	0	$(ab)^6$	$l((ab)^6) = \begin{pmatrix} 0\,2 \\ 0\,0 \end{pmatrix}$	0
$(ab)^6a$	$l((ab)^6a) = \begin{pmatrix} 0\,1 \\ 0\,0 \end{pmatrix}$	1	$(ab)^7$	$l((ab)^7) = \begin{pmatrix} 0\,1 \\ 2\,0 \end{pmatrix}$	1

So, $Q_1 = \{[\epsilon]_1\} \cup \{[(ab)^m a]_1 \mid m \leq 5\}$, and $Q_2 = \{[\epsilon]_2\} \cup \{[(ab)^m]_2 \mid m \leq 6\}$.

$$
\begin{tikzcd}
([\epsilon]_1, [\epsilon]_2) \arrow[d, "a"] & \\
([a]_1, [a]_2) \arrow[d, "b"] & ([(ab)^6]_1, [(ab)^6]_2) \arrow[l, "a"'] \\
([ab]_1, [ab]_2) \arrow[d, "a"] & ([(ab)^5 a]_1, [(ab)^5 a]_2) \arrow[u, "b"] \\
([aba]_1, [aba]_2) \arrow[d, "b"] & ([(ab)^5]_1, [(ab)^5]_2) \arrow[u, "a"] \\
([(ab)^2]_1, [(ab)^2]_2) \arrow[d, "a"] & ([(ab)^4 a]_1, [(ab)^4 a]_2) \arrow[u, "b"] \\
([(ab)^2 a]_1, [(ab)^2 a]_2) \arrow[d, "b"] & ([(ab)^4]_1, [(ab)^4]_2) \arrow[u, "a"] \\
([(ab)^3]_1, [(ab)^3]_2) \arrow[r, "a"] & ([(ab)^3 a]_1, [(ab)^3 a]_2) \arrow[u, "b"]
\end{tikzcd}
$$

$$(ab)^m \downharpoonleft 1 = (ab)^{m-1} a$$
$$(ab)^m \downharpoonleft 2 = (ab)^m$$
$$(ab)^m a \downharpoonleft 1 = (ab)^m a$$
$$(ab)^m a \downharpoonleft 2 = (ab)^m$$

Fig. 14. A DSA for the language $(ab)^*$.

6.5 Properties of the constructed DSA $\mathcal{A}$

1. We observe that by construction, $\longrightarrow_{\mathcal{A}}$ is deterministic.
2. $\mathcal{A}$ is stutter-free.

Proof. Suppose $\overline{p}\xrightarrow{a}_{\mathcal{A}}\overline{q}$. We need to show that for all $i \in loc(a), \overline{p}[i] \neq \overline{q}[i]$.
By our construction, there is an $x \in \Sigma^*$ such that $\overline{p} = ([x]_1, [x]_2, \ldots, [x]_n)$ and $\overline{q} = ([xa]_1, [xa]_2, \ldots, [xa]_n)$. For all $i \in loc(a)$, i-parity$(x) \neq i$-parity(xa), hence $[x]_i \neq [xa]_i$. Therefore, $\overline{p}[i] \neq \overline{q}[i]$. □
3. $\mathcal{A}$ is four-alternation-free.

Proof. Let $r \in \Sigma^*$ be such that $r \Downarrow i = r_1 a$, $r_1 \Downarrow k = r_2 b$, $r_2 \Downarrow i = r_3 c$, $r_3 \Downarrow k = r_4 d$ and $r_4 \Downarrow i = r_5$. Further, let $a, c \in \Sigma_i \setminus \Sigma_k$ and $b, d \in \Sigma_k \setminus \Sigma_i$. We show that $(r_5)_{\mathcal{A}} \neq (r)_{\mathcal{A}}$ which shows $\mathcal{A}$ is four-alternation-free.
We show that $l(r)(i,k) \neq l(r_5)(i,k)$, which gives us the result. Let $l(r_5)(i,k) = c \ mod \ 3$. We know that for all $x \in \Sigma^*, i, j \in Loc$, $l(x)(i,j) = l(x \Downarrow i)(i,j)$. We repeatedly use this observation below.

$$
\begin{array}{lll}
l(r_4)(i,k) & = l(r_4 \Downarrow i)(i,k) & \\
 & = l(r_5)(i,k) & \text{assumption} \\
 & = c \ mod \ 3 & \text{say} \\
l(r_3)(k,i) & = l(r_3 \Downarrow k)(k,i) & \\
 & = l(r_4 d)(k,i) & \text{assumption} \\
 & = (l(r_4)(i,k)+1) \ mod \ 3 & \text{since } d \in \Sigma_k \setminus \Sigma_i \\
 & = (c+1) \ mod \ 3 & \\
l(r_2)(i,k) & = l(r_2 \Downarrow i)(i,k) & \\
 & = l(r_3 c)(i,k) & \text{assumption} \\
 & = (l(r_3)(k,i)+1) \ mod \ 3 & \text{since } c \in \Sigma_i \setminus \Sigma_k \\
 & = (c+2) \ mod \ 3 & \\
l(r_1)(k,i) & = l(r_1 \Downarrow k)(k,i) & \\
 & = l(r_2 b)(k,i) & \text{assumption} \\
 & = (l(r_2)(i,k)+1) \ mod \ 3 & \text{since } b \in \Sigma_k \setminus \Sigma_i \\
 & = c \ mod \ 3 & \\
l(r)(i,k) & = l(r \Downarrow i)(i,k) & \\
 & = l(r_1 a)(i,k) & \text{assumption} \\
 & = (l(r_1)(k,i)+1) \ mod \ 3 & \text{since } a \in \Sigma_i \setminus \Sigma_k \\
 & = (c+1) \ mod \ 3 &
\end{array}
$$

Hence, $l(r_5)(i,k) \neq l(r)(i,k)$, as required. ■

Lemma 6.2. $\forall x \in \Sigma^* : (([\epsilon]_1, 0), ([\epsilon]_2, 0), \ldots, ([\epsilon]_n, 0)) \overset{x}{\Rightarrow}_{\mathcal{A}} ([x]_1, [x]_2, \ldots, [x]_n)$.

Proof: By the construction, $\longrightarrow_{\mathcal{A}}$ is deterministic and by Corollary 6.22.1 $\longrightarrow_{\mathcal{A}}$ satisfies the *asynchrony* property. Hence we get the result by induction on $|x|$. ■

Proposition 6.23 $L(\mathcal{A}\ ) = L$.

Proof: ($\supseteq$): Let $x \in L$. Then, by the previous lemma

$$([\epsilon]_1, [\epsilon]_2, \ldots, [\epsilon]_n) \overset{x}{\Rightarrow}_{\mathcal{A}} ([x]_1, [x]_2, \ldots, [x]_n) \in F.$$

Hence $x \in L(\mathcal{A}\ )$.
($\subseteq$): Let $x \in L(\mathcal{A}\ )$. Then, $([x]_1, [x]_2, \ldots, [x]_n) \in F$. This implies

$$([x]_1, [x]_2, \ldots, [x]_n) = ([y]_1, [y]_2, \ldots, [y]_n)$$

for some $y \in L$.
By Corollary 6.22.2, $x \backslash L = y \backslash L$. So since $y \in L$, $x \in L$. ■
This proposition completes the proof of Theorem 6.14, which, alongwith Theorem 6.10, gives us Theorem 6.1.