



MODELS FOR CONCURRENCY:
LOCAL PRESENTATIONS FOR FINITE STATE
DISTRIBUTED SYSTEMS

THESIS

Submitted to the University of Madras
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

by
Swarup Kumar Mohalik

DEPARTMENT OF THEORETICAL COMPUTER SCIENCE
INSTITUTE OF MATHEMATICAL SCIENCES
CHENNAI

June 1998

॥ ॐ श्री सदगुरु माधवनाथाय नमः ॥

कां साडेपंधरया रजतवणी । तैशीं स्तुतीचीं बोलणीं ।
उगियांचि माथा ठेविजे चरणीं । हेंचि भले ॥

॥ I pay my obeisance to Sadguru Madhavanath ॥

Singing in His praise is like attempting to adorn gold with silver. (It is) better to silently surrender at His lotus-feet.

In Chapter IX of Bhavartha Deepika by Saint Dnaneshwar.

CERTIFICATE

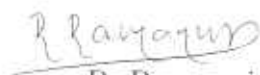
This is to certify that the Ph.D. thesis titled "*Models for Concurrency: Local Presentations for Finite State Distributed Systems*" submitted by Swarup Kumar Mohalik is a bonafide research work done under my supervision. The research work presented in this thesis has not formed the basis for the award to the candidate of any Degree, Diploma, Associateship, Fellowship or other similar titles. It is further certified that the thesis represents independent work by the candidate and collaboration was necessitated by the nature and scope of the problems dealt with.

THE INSTITUTE OF MATHEMATICAL SCIENCE

C.I.T. CAMPUS, MADRAS-600 113.

IMSC, CHENNAI

June 1998



R. Ramanujam

Thesis Supervisor

ACKNOWLEDGMENT

Time and again I faltered. And not only during the research programme. But every time Jam suggested ways out. I am grateful to him not only for his guidance and generous help during research but for his reassuring presence all the while.

I should not even try to put in word the love and patience of Manisha, my wife. Her co-operation and encouragement was crucial. I should not forget to mention my mother who left her cozy world in the village to help us take care of our son. Shreesh, our son, who caused most of the chaos, more than made up just by his loving gestures. My eldest brother, deserves special mention. At a point of time when I was at crossroads of my career he encouraged and supported me to continue in the higher studies.

Milind and Saritha inspired so much confidence I could turn to them for every little thing and they never disappointed. During course-work and discussions I learnt many things from Meena, Arvind, Venkatesh, Kamal and particularly from Thiagarajan and Madhavan. I will always remain grateful to all of them. I must also remember the generous help of the library and office staff, notably of Mr. Jayaraman, Venkatesan, Shankar and, in particular, Munnuswamy because of his ever-eagerness to help out.

I also owe it to my friends the many events in Matscience that made life joyful - Srimi's jokes, SVN's unbelievably innocent pranks, Jyotisman's short but sweet presence, introduction to western classical music by Subram, the ever-smiling Manas, long walks and popular lectures on Mathematics from Suri, a stint of Carnatic music with Raghavan alias Rags, Vinod's soccer lessons from Pushan and Sabu, Debanand and Suman who provided a home away from home, exercises in the gym with Mohan - and many many other happenings.

I must mention two others who I love and respect the most: Shreesh, my senior and friend who joined Ramakrishna mission after finishing his PhD programme, who set me an example of selflessness and my beloved Dadasaheb with whose blessings I believe I will be ever-peaceful.

Swarup Kumar Mohalik

Contents

1	Motivation	1
1.1	Models of concurrency	2
1.2	Process models and local presentation	3
1.3	Goal of the thesis	4
1.4	Contribution of the thesis	5
1.5	Scope	7
2	Preliminaries	9
2.1	Finite automata and regular languages	9
2.1.1	Transition systems and automata	10
2.1.2	Simulations and language acceptance	11
2.1.3	Regular expressions and Kleene's theorem	12
2.2	Languages for distributed systems	13
2.2.1	General notation	14
2.2.2	Shuffle languages	15
2.2.3	Synchronized shuffle languages	16
2.2.4	Regular consistent languages	19
2.2.5	Ochmanski's theorem	22
2.3	Automata for distributed systems	23
2.3.1	Product systems	24
2.3.2	Syntax for regular shuffle languages	27

2.3.3 . A Kleene theorem for regular shuffle languages	28
2.3.4 . Product systems with global final states	29
2.3.5 . Union closure of $\mathcal{L}(RSL_{\widetilde{\Sigma}})$	32
2.3.6 . Syntax for $\mathcal{L}(BRS L_{\widetilde{\Sigma}})$	33
2.4 . Other automata models for distributed systems	34
2.4.1 . Asynchronous transition systems	34
2.4.2 . Distributed transition systems	35
2.4.3 . Zielonka automata	35
2.4.4 . Local presentation for RCL's	37
2.5 . Infinite behaviour and ω -regular languages	38
2.5.1 . Finite state systems for infinite behaviour	38
2.5.2 . Syntax and McNaughton's theorem	40
2.5.3 . Simulations for ω -automata	40
2.5.4 . FSDS's and ω -languages	41
2.5.5 . Products and infinite behaviour	41
2.5.6 . Other classes of infinite behaviour	43
3 View-based presentation	45
3.1 . View-based systems	47
3.2 . Equivalence of $\mathcal{L}(VS)$ and RCL over $\widetilde{\Sigma}$	48
3.3 . Views	50
3.3.1 . Zielonka automaton and views	54
3.4 . View-based systems for regular consistent languages	56
3.5 . Discussion	59
4 Assumption and commitment in automata	60
4.1 . Perspective	60
4.1.1 . Parallel programs and compositional reasoning	60
4.1.2 . Local reasoning	61
4.1.3 . AC-framework and automata theory	63
4.2 . Local reasoning in finite-state process models	64
4.3 . Assumption-commitment on transitions	66
4.3.1 . A mutual exclusion example	66

6.3.1 • Languages	138
6.3.2 • <i>MonACS's</i> and <i>BFCRSL</i>	139
7 Conclusion	144
7.1 • Summary	144
7.2 • Automata theoretic issues	146
7.3 • Complexity	147
7.4 • Model-checking	147

List of Figures

2.1	A simple product system accepting $\{abc, bac\}$.	25
2.2	A slightly more complex product system accepting $(abc + bac)^*$.	26
2.3	A Zielonka automaton for $(ac + bc)^*$ over $\langle \Sigma_1 = \{a, c\}, \Sigma_2 = \{b, c\} \rangle$.	37
2.4	Product of ω -automata.	42
3.1	Interacting processes.	46
3.2	A view-based system for $([ab]c + [aabb]c)^*$.	49
3.3	Distributed computation on the string $abcdeab$.	51
3.4	Relating Zielonka automaton and i -views.	55
4.1	Two-processor mutual exclusion.	68
4.2	AT-system for the language $([ab]c + [aabb]c)^*$. $\Sigma_1 = \{a, c\}, \Sigma_2 = \{b, c\}$	71
4.3	The sender.	73
4.4	The receiver.	74
4.5	ACS for bit transmission	76
4.6	An infinite state protocol for STP.	78
4.7	Infinitely many assumptions and commitments for the infinite state protocol.	79
4.8	An incorrect folding of the infinite state protocol for STP.	79
4.9	Assumptions and commitments for a correct finite state protocol of Fig. 4.10.	80
4.10	A correct folding of the infinite state protocol for STP.	81
5.1	An example ACS accepting $(ab)^*$ over the alphabet $(\{a\}, \{b\})$.	85
5.2	Union and concatenation of ACS's.	88

5.3	$event(xa) \rightarrow j event(ya)$	94
5.4	When \mathcal{A} is four-alternation-free, such loops are not present in \mathcal{A} .	96
5.5	The case $u \Downarrow k \neq (u \Downarrow i) \Downarrow k$.	97
5.6	A DSA for the language $(ab)^*$.	111
5.7	Constructing M' from M .	131
6.1	MonACS for the language $L = [ab][cd]$.	140

1 Motivation

In day-to-day life, one comes across a number of services that are *distributed* in nature, for example, network applications like computer aided education, remote diagnoses, reservation systems, remote bank transactions, electronic mail and the World Wide Web, to name a few. One observes that such systems are essentially spatially distributed autonomous *processes* (or *agents* as we call them at times) with some kind of communication mechanism that enables them to transfer and/or share information among themselves. These are usually called *distributed systems*.

Distributed systems have become essential because of a variety of reasons. The basic cause, however, is the lack of resources, both informational and computational, in a single localized system. They provide faster service when the communication medium is fast (by having dedicated servers on the network) and they provide a high degree of reliability (by duplicating services). These systems also facilitate access to diverse kind of information available at different locations in LANs and WANs. Recently, with the advent of Java, applications that use such distributed information are becoming commonplace.

With some reflection, one notes that spatial distribution is only relative to the granularity at which one views a system. Therefore, while it is easy to think of networks of computers as natural examples of distribution, even a set of components on a chip can be seen as a distributed system where each component does its own computation and interacts with others over the the data and control lines on the chip. Such a distributed view of any

system is particularly interesting from the point of analysis of its behaviour. This is because it offers the hope of cutting down complexity of verification of large systems by studying properties of the component processes and then composing them. There have been several attempts from this point of view and there is some success for a limited classes of properties (notably, *safety* properties), though compositional verification in full generality still remains a holy grail for theoretical computer science.

Design and analysis of distributed systems has to address a range of problems that are absent in sequential systems. They arise mainly as a consequence of the need for synchronization among agents in the system. Two fundamental problems are those of *contention* and *cooperation* [LL] which lead to problems such as starvation and deadlock. One must be able to verify the design and implementation of distributed systems to avoid these and other problems.

1.1 Models of concurrency

In order to understand the nature of distributed computing, and to help the design and analysis of distributed systems, we need mathematical frameworks in which one can describe them and reason about their behaviour. In the theory of concurrency one studies models like Petri nets, event structures, trace languages and labeled transition systems or automata [BC, Dro, Muk, NRT, WN]. These models are designed so as to capture the semantics of distributed programs; reasoning about these programs is carried out in process algebras, dynamic and temporal logics, automata theory etc.

In many areas of theoretical computer science, notably model checking, one studies *finite state systems*, mainly to be able to get decision procedures that help in mechanizing verification. The assumption of "finite number of states" is actually not very restrictive for the following reason. Many interesting properties of distributed systems are *control* properties (like *deadlock*, *reachability*, *termination* etc). They depend upon *boundedly* many changes in the values of only a finite number of variables (*semaphores*, *number of messages sent*, *number of messages received* etc.) [MP]. Consequently, for the purpose of analysis, the

state space of the system can be partitioned into a finite number classes. In the light of this, it suffices to study models of concurrency in the context of finite state systems.

1.2 Process models and local presentation

Typically, the models mentioned above are at a *global* level, i.e., they describe system behaviour from the viewpoint of an external observer who can view the *entire* system. As opposed to this, we can also consider **process models** (as, for instance, usually studied in distributed algorithms) where no such global observer is assumed[LL].

In process models, a distributed system is assumed to consist of a finite number of *sequential processes*, which have some resources allocated to them. They proceed asynchronously and periodically exchange information among themselves. Process models are distinguished by the mode of communication employed among the agents, which may be message passing, synchronization on actions or information exchange through shared variables or any other protocol. The global behaviour of the over-all system depends upon (a) the behaviour of individual processes, (b) the *protocol* specifying the interaction between processes and (c) some global specifications imposed by the environment (like fairness constraints, termination at desired states) that the behaviour should satisfy.

Abstracting away from process models, we say that

a class of distributed systems is **locally presented** if it is modeled as a set of components, one for each process, and the global behaviour is completely defined by a **fixed** set of construction rules **universal** to that class.

In this thesis, we study local presentations of *finite state* distributed systems (FSDS), where each component is modelled as a finite state automaton(FA) over a fixed finite set of actions and global behaviour of the system is specified by a **product** of the component processes. By imposing different structures on the component automata and by different rules for product construction, we get different classes of local presentations.

An important reason to study local presentations is that the sequential components can serve as natural models for temporal logics based on local reasoning. Compositional model checking is one of the major goals of computer-aided verification and we believe that local presentations (particularly over infinite words) may help [AH,KV]. Consider a temporal logic with local assertions that may refer to other agents' local formulas and with global formulas that assert compatibility. Then the hope is that each formula can be associated with a component automaton and it may be possible to do model checking individually for each agent and globally for compatibility.

Moreover, the local presentations considered in the thesis (namely, Assumption-compatible systems of Chapter 5) seem to be closely related to the class of *knowledge-based programs* [FHMV]. It is hoped that these systems offer an automata-theoretic account of knowledge in distributed systems.

1.3 Goal of the thesis

We want to study locally presented FSDS's from the point of view of their language behaviour. This is well-motivated for the following reason. We know that FA's form a robust class of models for finite state sequential systems. Their language theory is well-studied [HU] and serves as a foundation for many areas of computer science and, in particular, for verification methodologies like model checking [VW] and bisimulation techniques [Hoa, Mil]. To achieve similar ends in the case of locally presented FSDS's, developing a language theory for them makes eminent sense. Our goal is to establish tight connections (essentially Kleene's theorem) between classes of FSDS's that are locally presented, the languages they accept and their syntactic presentation in a *top-level parallel* fashion which reflects the architecture of process models.

Notice that this is essentially a *distribution* problem: given a class of behaviours in terms of languages, find whether there is a class of locally presented FSDS's that characterize it. Without any constraint on the number of processes in the system and on the resource allocation, this problem has an easy solution [BP]. Things, however, become substantially

complicated when there is an *a priori*, *fixed* distribution of resources (or, the actions).

Even with this constraint, the problem of distribution of global behaviour has been solved in various ways. One could distribute the transitions by imposing some *independence* condition on them while keeping the states global. The independence relation models the spatial distribution of actions in the system. Thus one gets the *asynchronous transition systems* of [Bed]. One could also allow concurrent transitions on many actions thus allowing for true concurrent behaviour as opposed to interleaving behaviour, and this is the basic idea behind *distributed transition systems*[LPRT]. These models lack any explicit notion of sequential components and hence are not locally presented.

States of the system can also be distributed among the processes. Asynchronous automata[Zie] and asynchronous cellular automata[CMZ] model such distribution. But in these formalisms there is some global specification, in terms of *global transitions*, so that behaviour of the systems is not *derived* from that of its components. Hence we do not consider them to be locally presented.

Study of even simple systems shows us that in order to locally present complex behaviour, some amount of global information has to be encoded in the local states of the components. When the processes communicate and cooperate with each other, this global information is used to filter out undesired computations. Given a global behaviour, we study what type of global information can be distributed into the local states and what should be the product construction that achieves this global behaviour. To do this in a *uniform* way is the real challenge.

1.4 Contribution of the thesis

The salient points of the thesis are

1. introduction of a framework for local presentation of finite state distributed systems (FSDS),

2. showing that a class of systems, called Assumption Compatible Systems characterize *all* regular behaviour over the given set of actions that is distributed over the processes,
3. showing that by putting constraints on system structure, we get different classes of behaviour, and
4. showing that infinite behaviour can also be characterized by these systems with suitable acceptance conditions.

The framework is in the spirit of the *assumption - commitment* [FP, MC] or *rely - guarantee* [Jon] paradigm which was introduced to facilitate compositional reasoning. The main idea behind the paradigm is that each process makes assumptions about the behaviour of other processes and commits to fulfilling those made by other processes about its own behaviour. We can compose processes only when mutual assumptions of processes are met. One can then reason about the behaviour of each process separately (locally), assuming that others maintain relevant properties and reason globally about their compatibility. While a number of researchers seem to have studied this paradigm 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. We formulate our framework using finite state automata as components. Assumptions and commitments are made by having a special alphabet called the *commit alphabet*. The product is then constructed with compatibility of assumptions and commitments as a main criterion.

In a distributed system, since processes may execute asynchronously, distribution of a given behaviour becomes complicated. This is because one has to ensure that the global behaviour is exactly constructed from local behaviours. If every process could *know* the state of computation of every other process, this becomes easy but in a distributed system, any process's view about the system at any point of time is necessarily partial. Then it becomes unclear as how to preserve the global behaviour via distribution. In Chapter 3, we present **view-based systems** that are locally presented and capture the well-studied behaviour of *regular consistent languages (RCL)* accepted by asynchronous cellular automata. Our

focus here is on deriving some intuition about the nature of distribution. The assumption - commitment framework is introduced later in Chapter 4.

In Chapter 5 we propose a class of locally presented systems, in the assumption - commitment framework. We call these **Assumption-compatible systems**. We show that this class of systems captures the most general kind of behaviour that the finite state systems can exhibit, namely, *all* regular behaviours over the set of actions in the system. The flexibility of the framework is demonstrated in Chapter 6 by showing that with simple restrictions in the framework, we can capture various natural classes of language behaviour. We also show that by only changing the environmental specification (keeping the product construction unchanged), one can capture all *infinite* behaviours of finite state systems.

1.5 Scope

Our study is restricted to only top-level parallelism of static process models where the number of components is constant and each component can carry out only a fixed set of actions. Also, in our model, the only way in which processes can explicitly communicate is by synchronization (handshaking) and this is modeled by having actions common to components. Surely, one needs to relax these constraints and study the automata theory of other models, like message-passing systems. But even with this simple model, the problem of distribution seems well-motivated and remains non-trivial.

There are several other directions in which this work needs to be expanded. The algebraic theory of locally presented FSDS's awaits formulation. We have not addressed the issues of complexity and succinctness in the thesis, while for design of actual systems, study of these is essential. Logics that can take advantage of the proposed models for better decision procedures and model-checking need to be designed. In order to demonstrate the utility of the suggested framework, one must analyze and verify protocols in these models. Last but not the least, the relationship of these models with others (like knowledge based programming)[FHMV] is to be explored in depth. All these promise to lead the present work into a fruitful area of research in theoretical computer science.

Organization of the thesis

Chapter	Title	Contents
1	Motivation	General development, goal and contribution of the thesis.
2	Preliminaries	Notation, elementary notions in product systems, trace theory and infinite behaviours.
3	View-based systems	A first attempt at local presentation: systems characterizing regular trace languages.
4	Assumption-commitment paradigm	Description and illustration by detailed examples.
5	Assumption-compatible systems	The main framework is introduced and expressive power in terms of language acceptance is studied. A simple syntax for the systems is given. The theory is extended to regular infinite behaviours.
6	Assumption-compatible systems with restrictions	We show how some interesting subclasses of regular languages can be captured by putting restrictions on assumption-compatible systems.
7	Conclusion	Summary of results and directions for further research.

2 Preliminaries

The main purpose of this chapter is to study the concept of *products* of finite state automata. This notion is crucial for local presentations of finite state distributed systems (FSDS) because interaction between components of such systems is captured by the construction of the product. We describe the simplest product construction where synchronization is the only way to control behaviour. It turns out that products of locally presented FSDS's accept only a very weak class of languages. In the rest of the thesis, we will essentially be working with variants of this notion to increase the expressive power (in terms of language acceptance) of the models. Hence we fix some uniform notation and note several preliminary concepts and results about products in the chapter.¹

2.1 Finite automata and regular languages

Since we will be dealing with finite state systems throughout the thesis, and, in particular, about the class of languages they accept, we give some basic definitions and results in the language theory of finite state systems[HU]. The notation is nonstandard, but the reader is requested to bear with this in the interest of what comes in the sequel.

¹Some of the results in the chapter are classical, some known and some new but simple.

2.1.1 Transition systems and automata

Let Σ be a finite and nonempty alphabet. Σ^* is the set of all finite strings over Σ . We use a, b, c, \dots for the letters of Σ and x, y, z, \dots for strings. The empty string is denoted as ϵ . Concatenation of two strings x and y is denoted as either $x \cdot y$ or just as xy . Any subset $L \subseteq \Sigma^*$ is called a *language* over Σ .

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. We use letters p, q etc. for the states of a TS. When $(q, a, q') \in \longrightarrow$, we write it as $q \xrightarrow{a} q'$.

The one step transition on letters is extended to transitions on strings over Σ^* . We denote the extended transition relation as $\Longrightarrow : Q \times \Sigma^* \times Q$. It is defined inductively as follows: (1) $p \xrightarrow{\epsilon} p$, (2) $p \xrightarrow{xa} q$ iff there exists an $r \in Q$ such that $(p \xrightarrow{x} r \text{ and } r \xrightarrow{a} q)$. When $p \xrightarrow{x} q$, we say that there is a *run* of M from p to q on x .

We call a TS *deterministic* (DTS) if \longrightarrow is a function on $(Q \times \Sigma)$. This entails that whenever $p \xrightarrow{a} q$ and $p \xrightarrow{a} q'$ we have $q = q'$. Notice that if M is a DTS, the extended transition relation is a function. This implies that given any $p \in Q$ and $x \in \Sigma^*$, there is a unique q such that $p \xrightarrow{x} q$. (Existence of q is ensured by the totality assumption). In this case, we denote by $(x)_M$ the state q such that $q^0 \xrightarrow{x} q$.

The tuple $A = (M, F)$, where M is a TS and $F \subseteq Q$, is called a *finite state automaton* (FA). When $q_0 \xrightarrow{x} q_k$ and $q_k \in F$, we say that the string x is accepted by A . The set of all strings accepted by A (also called the language of A) is denoted as $L(A)$ or $L(M, F)$.

Observation 2.1 Let (M, F) be a finite state automaton and let $F = \{q_1^f, q_2^f, \dots, q_k^f\}$. Then,
$$L(M, F) = \bigcup_{i \in \{1, \dots, k\}} L(M, \{q_i^f\}).$$

Definition 2.2 Given Σ , the class of languages accepted by FA's over Σ is defined as:

$$\mathcal{L}(FA_\Sigma) \stackrel{\text{def}}{=} \{L \subseteq \Sigma^* \mid \text{there is an FA } (M, F) \text{ such that } L = L(M, F)\}.$$

We also call this class of languages **recognizable** and denote it as Rec_Σ .

An FA (M, F) is called *deterministic* (DFA) if M is a DTS. We denote the class of languages accepted by DFA over Σ as $\mathcal{L}(DFA_\Sigma)$. It is easy to show that non-determinism does not add to expressive power of FA's in terms of language acceptance, i.e., $\mathcal{L}(DFA_\Sigma) = \mathcal{L}(FA_\Sigma) = Rec_\Sigma$. DFA's have many nice properties. Easy complementation is one of them. This has been extremely useful in automata-theoretic techniques for model checking.

Observation 2.3 Let $(M = (Q, \longrightarrow, q^0), F)$ be an FA, Then its complement is $(M, Q - F)$. It is easy to show that $\Sigma^* - L(M, F) = L(M, Q - F)$.

2.1.2 Simulations and language acceptance

Let $M = (Q, \longrightarrow, q^0)$ and $M' = (Q', \longrightarrow', q^{0'})$ be two transition systems over Σ . We say that M' *simulates* M iff there exists a map $\Theta : Q' \rightarrow Q$ such that

1. $\Theta(q^{0'}) = q^0$,
2. for $p', q' \in Q'$, if $p' \xrightarrow{a'} q'$ then $\Theta(p') \xrightarrow{a} \Theta(q')$,
3. for $p, q \in Q$, if $p \xrightarrow{a} q$ and there exists $p' \in Q'$ such that $\Theta(p') = p$, then there exists a $q' \in Q'$ such that $\Theta(q') = q$ and $p' \xrightarrow{a'} q'$.

M' is essentially an unfolding of M .

Proposition 2.4 Let $M = (Q, \longrightarrow, q^0)$ and $M' = (Q', \longrightarrow', q^{0'})$ be two transition systems over Σ such that M' simulates M . Also, let $F' = \{p' \mid \Theta(p') \in F\}$. Then $L(M, F) = L(M', F')$.

Proof: We first show the left-to-right inclusion $L(M, F) \subseteq L(M', F')$. Consider the following claim.

Claim: Suppose that $p, q \in Q$ and there is a $p' \in Q'$ such that $\Theta(p') = p$. If $p \xrightarrow{x} q$ then there is a $q' \in Q'$ such that $\Theta(q') = q$ and $p' \xrightarrow{x'} q'$.

Assume the claim. Let $x \in L(M, F)$. Then, there is a state $q^f \in F$ such that $q^0 \xrightarrow{x} q^f$. Since $\Theta(q^{0'}) = q^0$ (from condition (1)), by the claim, there is a $q^{f'} \in Q'$ such that $\Theta(q^{f'}) = q^f$ and $q^{0'} \xrightarrow{x'} q^{f'}$. Hence $q^{f'} \in F'$ and $x \in L(M', F')$. Thus, $L(M, F) \subseteq L(M', F')$.

Proof of claim: The proof is by induction on length of x . The base case when $x = \epsilon$ is trivial. For the induction step, let $x = ya$ and let $p \xrightarrow{x} q$. We have to show that there is a $q' \in Q'$ such that $\Theta(q') = q$ and $p' \xrightarrow{x} q'$.

Since $p \xrightarrow{ya} q$, there is an $r \in Q$ such that $p \xrightarrow{y} r \xrightarrow{a} q$. By induction hypothesis, there is an $r' \in Q'$ such that $\Theta(r') = r$ and $p' \xrightarrow{y} r'$. Then, by condition (3) of simulation, there is a $q' \in Q'$ such that $\Theta(q') = q$ and $r' \xrightarrow{a} q'$. Hence $p' \xrightarrow{ya} q'$, or, $p' \xrightarrow{x} q'$. This proves the claim.

Now, we need to show the other inclusion $L(M, F) \supseteq L(M', F')$. A straightforward induction on $|x|$ using condition (2) shows that whenever $p' \xrightarrow{x} q'$ in M' , $\Theta(p') \xrightarrow{x} \Theta(q')$ in M .

Let $x \in L(M', F')$. Then, there is a state $q^{f'} \in F'$ such that $q^{0'} \xrightarrow{x} q^{f'}$. By the observation above, $\Theta(q^{0'}) \xrightarrow{x} \Theta(q^{f'})$. Since $\Theta(q^{0'}) = q^0$ and $\Theta(q^{f'}) \in F$ (by the definition of F'), $x \in L(M, F)$. Therefore, we have $L(M, F) \supseteq L(M', F')$ and the proposition is proved. ■

2.1.3 Regular expressions and Kleene's theorem

Let REG_{Σ} be the smallest set such that

1. $\emptyset \in REG_{\Sigma}$,
2. $\Sigma \subseteq REG_{\Sigma}$,
3. if $X, Y \in REG_{\Sigma}$, then $X + Y$ and XY are in REG_{Σ} ; and
4. if $X \in REG_{\Sigma}$, then X^* is in REG_{Σ} .

The elements of REG_{Σ} are called *regular expressions* over Σ . We assign to each expression a language over Σ^* . First we define the *concatenation* and $*$ operation on languages over Σ .

$$L_1 \cdot L_2 \stackrel{\text{def}}{=} \{xy \mid x \in L_1 \text{ and } y \in L_2\}.$$

$$L^0 = \{\epsilon\} \text{ and } L^{n+1} = L \cdot L^n. \text{ Set } L^* \stackrel{\text{def}}{=} \bigcup_{n \geq 0} L^n.$$

Semantics of regular expressions is given by a function $[] : REG_{\Sigma} \rightarrow 2^{\Sigma^*}$ defined as follows:

- $[\emptyset] = \emptyset$,
- $[a] = \{a\}$ when $a \in \Sigma$,
- $[X + Y] = [X] \cup [Y]$,
- $[XY] = [X] \cdot [Y]$,
- $[X^*] = [X]^*$.

Definition 2.5 A set $L \subseteq \Sigma^*$ is called **regular** if there exists a regular expression R in REG_{Σ} such that $L = [R]$. We denote the set of all regular languages from Σ^* as Reg_{Σ} .

Informally, a subset of Σ^* is regular if and only if it is obtained from the letters of Σ by a finite number of unions, concatenations and the star operation. Thus, we now have two classes of languages over Σ : the class of recognizable sets characterized by finite state automata over Σ and regular sets defined by regular expressions over Σ . The following well-known theorem of Kleene establishes the equivalence of the classes of recognizable and regular sets over Σ^* .

Theorem 2.6 (Kleene) $Rec_{\Sigma} = Reg_{\Sigma}$.

Kleene's theorem is significant because it allows one to express succinctly, through the regular expressions, the languages accepted by FA's. In any class of models based on automata, in order to have such succinct presentation of language behaviour, a syntactic characterization is helpful; in other words, one proves a Kleene theorem for such models.

2.2 Languages for distributed systems

Behaviour of finite state *sequential* systems is described by regular languages over some alphabet. What kind of languages describe behaviour of finite state *distributed* systems?

Naturally, these are regular since we are considering only finite state systems. But we are interested in knowing exactly what additional properties these languages have by virtue of being able to model distribution. For example, suppose there are two processes P_1 and P_2 in the system and they can carry out two actions a and b (respectively) independently. Then we can have both ab and ba as the behaviour since these actions are not causally related. Therefore any language this system accepts will be closed under such commutations of a and b . Similarly, since there may be different kinds of interaction in distributed systems, one can expect various subclasses of regular languages capturing behaviour of different kinds of distributed systems. In the following we present some well-known subclasses of languages for such systems. But first we fix some notation.

2.2.1 General notation

As discussed in Chapter 1, we want to model a static network of processes. Fix once and for all that there are “ n ” processes in the system. The finite set of *locations* is called $Loc = \{1, \dots, n\}, n > 0$.

We model the distribution of actions among the processes by a *distributed alphabet* $\tilde{\Sigma}$. It is a tuple $(\Sigma_1, \dots, \Sigma_n)$, 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 $\tilde{\Sigma}$, we often speak of the set $\Sigma \stackrel{\text{def}}{=} \Sigma_1 \cup \dots \cup \Sigma_n$ 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 agents participating in this action. Let $loc : \Sigma \rightarrow 2^{\{1, \dots, n\}}$ be defined by $loc(a) \stackrel{\text{def}}{=} \{i \mid a \in \Sigma_i\}$. So $loc(a)$ (called “locations of a ”) gives the set of agents that participate (or, synchronize) in the action a . By definition, for all $a \in \Sigma$, $loc(a) \neq \emptyset$.

We extend the notion of locations (or, participating processes) to strings over Σ^* . If $x \in \Sigma^*$, $alph(x) \stackrel{\text{def}}{=} \{a \in \Sigma \mid a \text{ occurs in } x\}$. Then, $loc(x) \stackrel{\text{def}}{=} \bigcup_{a \in alph(x)} loc(a)$.

From the definition of locations, we can derive an irreflexive and symmetric relation \mathcal{I} on Σ . We call this an *independence* relation. The independence relation is defined to be $\mathcal{I} \stackrel{\text{def}}{=} \{(a, b) \in \Sigma \times \Sigma \mid \text{loc}(a) \cap \text{loc}(b) = \emptyset\}$. Informally, $(a, b) \in \mathcal{I}$ iff the participants in the actions a and b are disjoint. Since processes are spatially distributed, it entails that the participants of b remain unaffected by the execution of a and vice versa. Hence in the system, independent actions may proceed asynchronously. This is a crucial property of the distributed systems we consider. We call this the *asynchrony* property of the systems. Extending the same idea to strings over Σ^* , we say two strings x and y are independent, i.e., $(x, y) \in \mathcal{I}$ iff $\text{loc}(x) \cap \text{loc}(y) = \emptyset$. Two independent strings can be thought of as independent execution sequences of two disjoint sets of processes.

Lastly, we say that the distributed alphabet $\bar{\Sigma}$ is *non-trivial* iff the independence relation on Σ is non-empty. For example, the distributed alphabet $(\Sigma_1 = \{a, b\}, \Sigma_2 = \{b, c\})$ is non-trivial because $(a, c) \in \mathcal{I}$. On the other hand, the distributed alphabet $\bar{\Sigma} = \langle \Sigma_1 = \{a, b\}, \Sigma_2 = \{b, c\}, \Sigma_3 = \{c, a\} \rangle$ is trivial because $\text{loc}(a) \cap \text{loc}(b) = \{1\} \neq \emptyset$ and so on for every pair of letters and hence \mathcal{I} is empty.

2.2.2 Shuffle languages

The simplest kind of interaction in the distributed system is disjoint parallelism, when there is *no* interaction among the components. In that case, global behaviour is just interleaving of local actions. This is the idea behind **Shuffle languages**.

Given two finite non-empty alphabets Σ_1 and Σ_2 , let $x \in \Sigma_1^*$ and $y \in \Sigma_2^*$. Define, $x \hat{\parallel} y$ (shuffle of x and y) as

$$x \hat{\parallel} y \stackrel{\text{def}}{=} \{x_1 y_1 \cdots x_k y_k \mid \text{for all } i, x_i \in \Sigma_1^*, y_i \in \Sigma_2^* \text{ and } x = x_1 x_2 \cdots x_k, y = y_1 y_2 \cdots y_k\}$$

Shuffle of x and y gives all possible interleavings of the two strings. For example, when $\Sigma_1 = \{a, b\}$ and $\Sigma_2 = \{c, d\}$, $ab \hat{\parallel} cd = \{abcd, acbd, cabd, acdb, cadb, cdab\}$. One extends this operation to languages over Σ .

Let $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$,

$$L_1 \hat{\parallel} L_2 \stackrel{\text{def}}{=} \{z \in \Sigma^* \mid \exists x \in L_1, y \in L_2 : z \in x \hat{\parallel} y\}$$

Observation 2.7 $\hat{\parallel}$ is associative and commutative. Hence, given languages $L_i \subseteq \Sigma_i^*$, $i = 1 \dots n$, their shuffle can be written as $L_1 \hat{\parallel} \dots \hat{\parallel} L_n$.

The following proposition verifies that shuffle of regular languages is also regular.

Proposition 2.8 Suppose $L_1 \subseteq \text{Reg}_{\Sigma_1}$ and $L_2 \subseteq \text{Reg}_{\Sigma_2}$. Then $L_1 \parallel L_2 \in \text{Reg}_{\Sigma_1 \cup \Sigma_2}$.

Proof: From the assumption, there are finite automata (M_1, F_1) and (M_2, F_2) such that for $i \in \{1, 2\}$, $L_i = L(M_i, F_i)$. Let $M_i = (Q_i, \rightarrow_i, q_i^0)$. Construct an FA $(M = (Q, \rightarrow, q^0), F)$ over $\Sigma_1 \cup \Sigma_2$ where

1. $Q = Q_1 \times Q_2$,
2. $q^0 = (q_1^0, q_2^0)$,
3. $F = F_1 \times F_2$, and
4. $\rightarrow \subseteq (Q \times (\Sigma_1 \cup \Sigma_2) \times Q)$ defined as:

$$(p_1, q_1) \xrightarrow{a} (p_2, q_2) \text{ iff } \begin{cases} \text{either } (p_1 \xrightarrow{a}_1 p_2 \text{ and } q_1 = q_2), \text{ or} \\ (p_1 = p_2 \text{ and } q_1 \xrightarrow{a}_2 q_2). \end{cases}$$

It is easy to see that M accepts the language $L_1 \hat{\parallel} L_2$. This proves the proposition. \blacksquare

Corollary 2.9 Let $\bar{\Sigma} = (\Sigma_1, \dots, \Sigma_n)$ and suppose, for all $i \in \text{Loc}$, $L_i \in \text{Reg}_{\Sigma_i}$. Then $L_1 \hat{\parallel} \dots \hat{\parallel} L_n \in \text{Reg}_{\Sigma}$.

2.2.3 Synchronized shuffle languages

We now consider a somewhat more interesting subclass of languages for distributed systems where the only way of communication is via synchronization on common actions. Note that given two alphabets Σ_1 and Σ_2 , the intersection $\Sigma_1 \cap \Sigma_2$ may not be empty. In this case the common actions act as synchronization points.

For example, let $\Sigma_1 = \{a, c\}$ and $\Sigma_2 = \{b, c\}$. Then, synchronization shuffle (or just sync-shuffle) of $c \in \Sigma_1^*$ and $c \in \Sigma_2^*$ is the string c and not cc as would be generated

by shuffle. Similarly, sync-shuffle of ac and cb should be $\{acb\}$ whereas shuffle of ac and cb would generate the set $\{accb, cacb, cbac, acbc, cabc\}$.

We give the formal definition below. For this it is useful to define a *component projection* map $\lceil: (\Sigma^* \times Loc) \rightarrow \Sigma^*$. Given a string over Σ^* , it finds the maximal subsequence over any local alphabet. It is defined inductively as follows.

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

One can extend the component projection to sets of strings $L \subseteq \Sigma^*$ as:

$$L \lceil i \stackrel{\text{def}}{=} \{x \lceil i \mid x \in L\}.$$

Let $x \in \Sigma_1^*$ and $y \in \Sigma_2^*$. Then,

$$x \hat{\parallel}_S y \stackrel{\text{def}}{=} \{z \in (\Sigma_1 \cup \Sigma_2)^* \mid z \lceil 1 = x \text{ and } z \lceil 2 = y\}.$$

Thus, when $\Sigma_1 = \{a, c\}$ and $\Sigma_2 = \{b, c\}$, $ac \hat{\parallel}_S b = \emptyset$. This is because for any z to be in the sync-shuffle, $z \lceil 1$ must be ac . Hence, c must occur in z and therefore, c must also occur in $z \lceil 2$. But by definition of sync-shuffle, $z \lceil 2$ must only be b , which is a contradiction. Hence, the sync-shuffle is empty.

We now extend sync-shuffle to languages over the distributed alphabet $\tilde{\Sigma}$. Let $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$,

$$L_1 \hat{\parallel}_S L_2 \stackrel{\text{def}}{=} \{z \in \Sigma^* \mid \exists x \in L_1, y \in L_2 : z \in x \hat{\parallel}_S y\}$$

Observation 2.10 In general, when a distributed alphabet has n components, one can extend $\hat{\parallel}_S$ to an n -ary operator. Let $x_i \subseteq \Sigma_i^*, i = 1 \dots n$. Then,

$$x_1 \hat{\parallel}_S \dots \hat{\parallel}_S L_n \stackrel{\text{def}}{=} \{x \in \Sigma^* \mid \text{for all } i \in Loc, x \lceil i = x_i\}.$$

Extending to languages $L_i \subseteq \Sigma_i^*, i = 1 \dots n$, n -way sync-shuffle can be defined as:

$$L_1 \hat{\parallel}_S \dots \hat{\parallel}_S L_n \stackrel{\text{def}}{=} \{x \in \Sigma^* \mid \text{for all } i \in Loc, x \lceil i \in L_i\}.$$

Proposition 2.11 Suppose $L_i \subseteq \text{Reg}_{\Sigma_i}, i = 1 \dots n$. Then $L_1 \hat{\parallel}_S \dots \hat{\parallel}_S L_n \in \text{Reg}_{\Sigma}$.

The proof of this proposition is similar to that of in Proposition 2.8. The only change is in the definition of transition relation where we now ensure that all the participating agents make local transitions when they synchronize. When $n = 2$, for example, we can define $\longrightarrow \subseteq (Q \times (\Sigma_1 \cup \Sigma_2) \times Q)$ as: $(p_1, q_1) \xrightarrow{a} (p_2, q_2)$ iff either

1. $p_1 \xrightarrow{a}_1 p_2$ and $q_1 = q_2$, when $a \in \Sigma_1 \setminus \Sigma_2$, or
2. $p_1 = p_2$ and $(q_1 \xrightarrow{a}_2 q_2)$, when $a \in \Sigma_2 \setminus \Sigma_1$, or
3. $p_1 \xrightarrow{a}_1 p_2$ and $(q_1 \xrightarrow{a}_2 q_2)$, when $a \in \Sigma_1 \cap \Sigma_2$.

Definition 2.12 A language $L \subseteq \Sigma^*$ is called a regular shuffle language (*RSL*) if for all $i \in \text{Loc}$, there are regular languages $L_i \subseteq \Sigma_i^*$ such that $L = L_1 \hat{\parallel}_S \dots \hat{\parallel}_S L_n$. Given a distributed alphabet $\tilde{\Sigma}$, the class of regular shuffle languages over $\tilde{\Sigma}$ is denote as $\mathcal{L}(\text{RSL}_{\tilde{\Sigma}})$.

From now on we consider only synchronized shuffle of languages. Hence, we drop the subscript "S" and use $\hat{\parallel}$ for sync-shuffle.

By the above proposition we know that the synchronized shuffle of regular languages is regular. Is the converse true? That is, is every regular language a regular shuffle language? Formally, given a distributed alphabet $\tilde{\Sigma}$ and a regular language L over Σ^* , do there exist $L_i \subseteq \Sigma_i^*, i \in \text{Loc}$, such that $L = L_1 \hat{\parallel} \dots \hat{\parallel} L_n$? Note that constructing a distributed alphabet $\tilde{\Sigma}$ and an *RSL* over $\tilde{\Sigma}$ for a given regular language is easy. The problem is non-trivial only with $\tilde{\Sigma}$ fixed or given.

If the distributed alphabet is such that $\Sigma_1 = \Sigma_2 = \dots = \Sigma_n$, then letting $L_i = L$ for all $i \in \text{Loc}$, we get an easy answer to the question. For general distributed alphabets, however, it is not the case that every language can be expressed as a sync-shuffle. In fact, we can show that for non-trivial distributed alphabets $\mathcal{L}(\text{RSL}_{\tilde{\Sigma}})$ is a strict subclass of Reg_{Σ} . How do we do this?

We know that the class of regular languages over any alphabet is closed under boolean operations and concatenation. So, if $\mathcal{L}(\text{RSL}_{\tilde{\Sigma}})$ was equal to Reg_{Σ} , then the former

should also be closed under these operations. But as we show in the following, this is not true for all distributed alphabets because of a simple characterization of sync-shuffle.

Proposition 2.13 *Given a distributed alphabet $\tilde{\Sigma}$, let $L = L_1 \hat{\parallel} \dots \hat{\parallel} L_n$ where for all $i \in \text{Loc}$, $L_i \subseteq \Sigma_i^*$. Then, $L = (L[1] \hat{\parallel} \dots \hat{\parallel} L[n])$.*

Proof: Since, $x \in L$ implies $x[i] \in L[i]$, by definition of $\hat{\parallel}$, $L \subseteq (L[1] \hat{\parallel} \dots \hat{\parallel} L[n])$.

On the other hand, if $x \in (L[1] \hat{\parallel} \dots \hat{\parallel} L[n])$, for all i , $x[i] \in L[i]$, that is, there is a $u_i \in L$ such that $x[i] = u_i[i]$. But then $u_i[i] \in L_i$. Hence, $x[i] \in L_i$. This implies $x \in L_1 \hat{\parallel} L_2 \hat{\parallel} \dots \hat{\parallel} L_n = L$ and therefore $(L[1] \hat{\parallel} \dots \hat{\parallel} L[n]) \subseteq L$. ■

Corollary 2.14 *Let $\tilde{\Sigma}$ be a non-trivial distributed alphabet. Then $\mathcal{L}(RSL_{\tilde{\Sigma}})$ is not closed under union, complementation or concatenation.*

Proof: By assumption, there are at least two letters a and b in Σ and $(a, b) \in \mathcal{I}$. To avoid much notation, assume that there are only two processes and that $\tilde{\Sigma} = (\{a\}, \{b\})$ satisfying the assumption.

Take $L_1 = \{a\}$ and $L_2 = \{b\}$. Both $L_1, L_2 \in \mathcal{L}(RSL_{\tilde{\Sigma}})$ since $L_1 = \{a\} \hat{\parallel} \{\epsilon\}$ and $L_2 = \{\epsilon\} \hat{\parallel} \{b\}$.

Let $L_{\cap} = \Sigma^* \setminus L_1$ so that $\{a\} \not\subseteq L_{\cap}$. Now, $\epsilon, ab \in L_{\cap}$. So, $a \in L_{\cap}[1]$ and $\epsilon \in L_{\cap}[2]$. This implies $\{a\} \subseteq (L_{\cap}[1] \hat{\parallel} (L_{\cap}[2])$. So, $L_{\cap} \neq (L_{\cap}[1] \hat{\parallel} (L_{\cap}[2])$ and hence $L_{\cap} \notin \mathcal{L}(RSL_{\tilde{\Sigma}})$.

Let $L_{\cup} = L_1 \cup L_2 = \{a, b\}$. Then, $(L_{\cup}[1] \hat{\parallel} (L_{\cup}[2]) = \{\epsilon, a, b, ab, ba\} \neq L_{\cup}$. Hence, $L_{\cup} \notin \mathcal{L}(RSL_{\tilde{\Sigma}})$.

Let $L_{\cdot} = L_1 L_2 = \{ab\}$. Then, $(L_{\cdot}[1] \hat{\parallel} (L_{\cdot}[2]) = \{ab, ba\} \neq L_{\cdot}$. Hence, $L_{\cdot} \notin \mathcal{L}(RSL_{\tilde{\Sigma}})$ and thus the corollary is proved. ■

2.2.4 Regular consistent languages

In distributed systems, independent actions can happen in any order in the absence of a global synchronization scheme (like a common clock). This kind of behaviour is modeled as languages that are closed with respect to commutation of consecutive independent actions.

The resulting languages are called *Regular consistent languages* (RCL) or *Regular trace languages* (RTL) of [Maz]. We discuss this class of languages, and later the automata models for them (Zielonka automata) in some detail, because Zielonka automata are quite close to local presentations.

Definition 2.15 Define the relation \sim on Σ^* as: for all $x, y \in \Sigma^*$, $x \sim y \stackrel{\text{def}}{=} \text{for all } i \in \text{Loc}, x[i] = y[i]$. It is easy to see that \sim is an equivalence. The equivalence classes of \sim are called *traces*.

In trace theory, it is customary to present \sim in an alternative form. Recall that given $\tilde{\Sigma}$, the independence relation $\mathcal{I} \subseteq \Sigma \times \Sigma$ is defined as $\mathcal{I} = \{(a, b) \mid \text{loc}(a) \cap \text{loc}(b) = \emptyset\}$. We define $\mathcal{D} \stackrel{\text{def}}{=} \Sigma \setminus \mathcal{I}$. Then the graph G where elements from Σ form the vertices and \mathcal{D} is the edge relation is called the *dependence graph* of (Σ, \mathcal{D}) .

Definition 2.16 Two words x and y are *1-trace equivalent*, $x \sim_t y$, if there are words $u, v \in \Sigma^*$ and $(a, b) \in \mathcal{I}$ such that $x = uabv$ and $y = ubav$. The trace equivalence, \sim_t , is the reflexive transitive closure of \sim_t .

The definitions of \sim and \sim_t can be shown to be equivalent [Maz].

Proposition 2.17 For all $x, y \in \Sigma^*$, $x \sim_t y$ iff $x \sim y$.

Proof: (\Rightarrow) : It suffices to show that if $x \sim_t y$ then $x \sim y$. Suppose $x \sim_t y$. Then there exists $u, v \in \Sigma^*$ and $(a, b) \in \mathcal{I}$ such that $x = uabv$ and $y = ubav$. We need to show that for all $i \in \text{Loc}$, $x[i] = y[i]$. Note that since $\text{loc}(a) \cap \text{loc}(b) = \emptyset$,

$$ab[i] = ba[i] = \begin{cases} a & \text{when } i \in \text{loc}(a), \\ b & \text{when } i \in \text{loc}(b). \end{cases}$$

1. for all $i \notin \text{loc}(a) \cup \text{loc}(b)$, $x[i] = (u[i])(v[i]) = y[i]$,
2. for all $i \in \text{loc}(a)$, $x[i] = (u[i])(ab[i])(v[i]) = (u[i])a(v[i]) = y[i]$, and
3. for all $j \in \text{loc}(b)$, $x[j] = (u[j])(b[i])(v[i]) = (u[j])b(v[j]) = y[j]$.

Hence, for all $i \in Loc$, $x[i] = y[i]$, which implies $x \sim y$.

(\Leftarrow) : The proof is by induction on length of x . Let $x = \epsilon$. Since $y[i] = \epsilon$ for all $i \in Loc$, $y = \epsilon$, and hence $x \sim_t y$.

Let $x = ua$. Then, by assumption, for all $i \in loc(a)$, $x[i] = (ua)[i] = y[i]$. Since $(ua)[i] = (u[i]a)$, $y = vaw$ for some $v, w \in \Sigma^*$ such that for all $i \in loc(a)$, $w[i] = \epsilon$. This implies that $loc(a) \cap loc(w) = \emptyset$. Hence, $y \sim_t vwa$. By the first half of the proof, for all $i \in Loc$, $y[i] = (vwa)[i]$.

By assumption, for all $i \in Loc$, $x[i] = y[i]$. Hence, we have for all $i \in Loc$, $x[i] = (ua)[i] = (vwa)[i]$. This gives us for all $i \in Loc$, $u[i] = vw[i]$. By induction hypothesis, $u \sim_t vw$. Hence $ua \sim_t vwa$ or $x \sim_t y$. ■

In the light of the above, we will use the symbol \sim to mean \sim_t as well. From the definition of \sim , we know that if $x \sim y$ then $alph(x) = alph(y)$. Hence for a trace t , $alph(t) \stackrel{\text{def}}{=} \{alph(x) \mid t = [x]_\sim\}$. Then, similar to strings, one can define $loc(t) \stackrel{\text{def}}{=} loc(x)$ where $t = [x]_\sim$. We say two traces are independent iff their locations are disjoint i.e., $(t_1, t_2) \in \mathcal{I}$ iff $loc(t_1) \cap loc(t_2) = \emptyset$.

$M(\Sigma, \mathcal{I}) = \Sigma^* / \sim$ is called the trace monoid. Let $\phi : \Sigma^* \rightarrow M(\Sigma, \mathcal{I})$ be a morphism such that $\phi(x) = [x]_\sim$. The syntactic congruence \sim_T on $M(\Sigma, \mathcal{I})$ is defined by

$$\forall r, t \in M(\Sigma, \mathcal{I}), t \sim_T r \text{ iff } \forall t_1, t_2 \in M(\Sigma, \mathcal{I}), t_1 t t_2 \in T \Leftrightarrow t_1 r t_2 \in T.$$

Definition 2.18 A trace language $T \subseteq M(\Sigma, \mathcal{I})$ is regular iff the syntactic congruence \sim_T is of finite index. Equivalently, T is regular iff $\phi^{-1}T$ is a regular subset of Σ^* .

One can read the definition of regular trace languages as those regular languages that are also closed under the equivalence relation \sim . In the thesis we make little use of the algebraic theory of languages and hence will confine ourselves to this view of regular trace languages.

Note that the closure of a regular language under \sim need not be regular. For example, let $\Sigma = \{a, b\}$ and $L = (ab)^*$. If $\mathcal{I} = \{(a, b), (b, a)\}$, then $\phi^{-1}L$ is the language containing strings with equal number of a 's and b 's, which is not regular any more.

Definition 2.19 A language $L \subseteq \Sigma^*$ is said to be a regular consistent language iff $L \in \text{Reg}_{\tilde{\Sigma}}$ and is closed under \sim .

Let $\mathcal{L}(\text{RCL}_{\tilde{\Sigma}})$ denote the class of all regular consistent languages over $\tilde{\Sigma}$. When a distributed alphabet is fixed, we refer to this class as just “RCL”.

2.2.5 Ochmanski’s theorem

Ochmanski’s theorem characterizes regular trace languages in terms of trace regular expressions. For the semantics of the trace regular expressions, one needs a notion of *connected* traces.

Definition 2.20 (Connected traces) A trace $t \in M$ is called *connected* if the dependence graph of t is connected or, equivalently, if the letters of $\text{alph}(t)$ induce a connected subgraph in the graph of the dependence alphabet (Σ, \mathcal{D}) , that is, $(\text{alph}(t), \mathcal{D}[\text{alph}(t)])$ is connected.

Every trace $t \in M$ can be partitioned into connected components $t = t_1 \oplus \dots \oplus t_n$ where $t_i \subseteq t$ are non-empty connected subtraces and $(t_i, t_j) \in I$ for $1 \leq i, j \leq n, i \neq j$. Given a trace language $L \subseteq M$ define the language of its connected components by

$$\text{Con}(L) = \{t' \in M \mid t' \text{ is a connected component of some } t \in L\}$$

The regular expressions given by Ochmanski are:

$$\text{TrReg}_{\tilde{\Sigma}} ::= t \in M \mid r_1 + r_2 \mid r_1 \cdot r_2 \mid r^\dagger$$

Semantics of these expressions is similar to that for regular expressions except for iteration where it is defined as $[r^\dagger] = (\text{Con}([r]))^*$.

Theorem 2.21 (Ochmanski) Let $L \subseteq M$ be a trace language. Then L is recognizable iff there is a regular expression $r \in \text{TrReg}_{\tilde{\Sigma}}$ such that $L = [r]$.

2.3 Automata for distributed systems

A finite state sequential system can be modeled as an FA. The natural idea to model finite state distributed systems is by a collection of FA's, one for each process, over some local alphabets. Since we are interested in process models that are top-level parallel, each component process is taken as a sequential system.

Definition 2.22 A distributed system over $\tilde{\Sigma}$ is a tuple $\tilde{M} = (M_1, \dots, M_n)$, where for all $i \in \text{Loc}$, $M_i = (Q_i, \rightarrow_i, q_i^0)$ is a TS over Σ_i .

Fix a distributed system $\tilde{M} = (M_1, \dots, M_n)$ over $\tilde{\Sigma}$. Global transitions of \tilde{M} is given by a *product transition system* (product TS).

Definition 2.23 Let $Q \stackrel{\text{def}}{=} Q_1 \times \dots \times Q_n$. A TS $\hat{M} = (\hat{Q}, \rightarrow, (q_1^0, \dots, q_n^0))$, where $\hat{Q} \subseteq Q$, $\rightarrow \subseteq (\hat{Q} \times \Sigma \times \hat{Q})$ and $(q_1^0, \dots, q_n^0) \in \hat{Q}$, is called a **product TS** of \tilde{M} on Σ if it satisfies the **asynchrony condition**:

$$(p_1, \dots, p_n) \xrightarrow{a} (q_1, \dots, q_n) \text{ iff}$$

1. $\forall i \in \text{loc}(a), p_i \xrightarrow{a} q_i$, and
2. $\forall j \notin \text{loc}(a), p_j = q_j$.

\hat{M} is a **complete product TS** if $\hat{Q} = Q$.

We use \bar{p}, \bar{q} etc. to denote elements of Q . For any global state $\bar{s} = (p_1, p_2, \dots, p_n)$, $\bar{s}[i]$ denotes the i -th component p_i . From the definition of a product TS of a distributed system, we make the following observations.

Observation 2.24 1. Suppose \hat{M} is a complete product TS of \tilde{M} and $\text{loc}(a) \cap \text{loc}(b) = \emptyset$. Then, for all \bar{p} and \bar{q} in \hat{Q} ,

$$(\bar{p} \xrightarrow{ab} \bar{q} \text{ implies } \bar{p} \xrightarrow{ba} \bar{q}).$$

2. Suppose that all the M_i 's are deterministic. Then, (1) the product \hat{M} is also deterministic, and (2) for all $x, y \in \Sigma^*$, if $y[i] = \epsilon$, then $(x)_{\hat{M}}[i] = (xy)_{\hat{M}}[i]$.

Global behaviour of a system is given by the language accepted by the *product automaton* of \tilde{M} .

Definition 2.25 The *product automaton* of \tilde{M} is a pair (\widehat{M}, F) where \widehat{M} is a product TS of \tilde{M} and $F \subseteq \hat{Q}$. The language accepted by \tilde{M} is $L(\tilde{M}) \stackrel{\text{def}}{=} L(\widehat{M}, F)$.

From Observation 2.24(1) above, it is clear that languages accepted by systems of complete products are consistent.

In the following two sections, we exemplify the notion of product construction by the simplest of rules. The consequent classes of distributed automata which we call *product systems* and *extended product systems* characterize classes of languages we have seen before.

2.3.1 Product systems

Define a class of distributed systems in the following manner.

Definition 2.26 A *Product System* over $\tilde{\Sigma}$ is a tuple

$$\tilde{M} = (M_1, \dots, M_n, \langle F_1, \dots, F_n \rangle), \text{ where}$$

1. (M_1, \dots, M_n) is a distributed system over $\tilde{\Sigma}$,
2. for all $i \in \text{Loc}$, (M_i, F_i) is an FA over Σ_i , and
3. the product automaton of \tilde{M} is (\widehat{M}, F) where \widehat{M} is the complete product TS of \tilde{M} and $F = \prod_{i=1}^n F_i$.

Let $PS_{\tilde{\Sigma}}$ stand for the class of product systems over $\tilde{\Sigma}$. The class of languages accepted by $PS_{\tilde{\Sigma}}$ is denoted as $\mathcal{L}(PS_{\tilde{\Sigma}})$. Formally, $\mathcal{L}(PS_{\tilde{\Sigma}}) = \{L \subseteq \Sigma^* \mid \text{there is a PS } \tilde{M} \text{ over } \tilde{\Sigma} \text{ such that } L = L(\tilde{M})\}$. Later in the section we will show that this is exactly the class of regular shuffle languages over $\tilde{\Sigma}$ (namely, $\mathcal{L}(RSL_{\tilde{\Sigma}})$).

Example Let $\tilde{\Sigma} = \{\{a, c\}, \{b, c\}\}$. Fig. 2.1 illustrates a very simple $PS \tilde{M} = ((M_1, F_1), (M_2, F_2))$. Here $F_1 = \{p_3\}$ and $F_2 = \{q_3\}$. $\hat{Q} = Q_1 \times Q_2$. The language accepted by \tilde{M} is

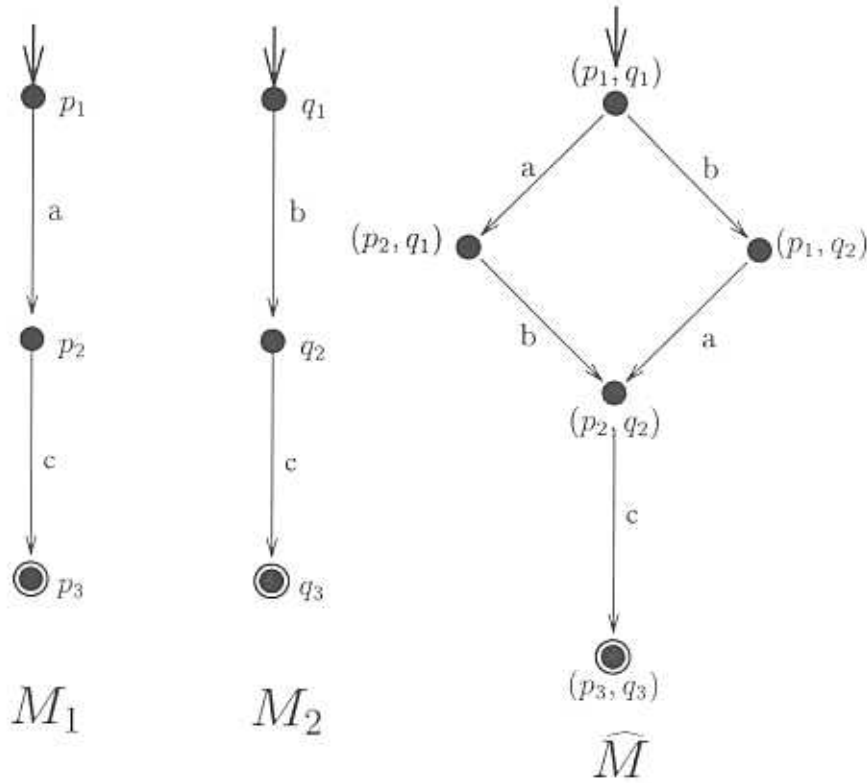


Figure 2.1: A simple product system accepting $\{abc, bac\}$.

$L(\tilde{M}) = \{abc, bac\}$. Note that in the complete product TS, the states (p_3, q_2) and (p_2, q_3) become unreachable from the initial state.

In 2.2 we give another example PS over $\tilde{\Sigma}$ and its product. Note that all the global states are reachable here.

Proposition 2.27 *Let $\tilde{M} = (M_1, \dots, M_n, < F_1, \dots, F_n >)$ be a product system. Then*

$$L(\tilde{M}) = L(M_1, F_1) \parallel \dots \parallel L(M_n, F_n).$$

Proof: (\subseteq) Let $x \in L(\tilde{M})$. Note that $L(\tilde{M}) = L(\tilde{M}, \Pi_{i=1}^n F_i)$ where \tilde{M} is the product of \tilde{M} on $\Pi_{i=1}^n Q_i$. Hence, there is an accepting run $(q_1^0, \dots, q_n^0) \xrightarrow{x} (q_1^f, \dots, q_n^f)$ where for all $i \in Loc, q_i^f \in F_i$.

Claim: For all $x \in \Sigma^*, (p_1, \dots, p_n) \xrightarrow{x} (q_1, \dots, q_n)$ implies (for all $i \in Loc, p_i \xrightarrow{x|_i} q_i$.)

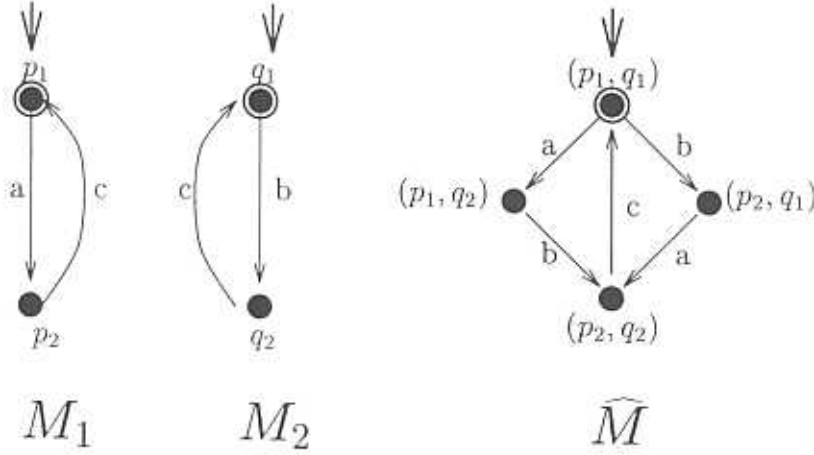


Figure 2.2: A slightly more complex product system accepting $(abc + bac)^*$.

Assume the claim. Then, for all $i \in Loc$, $q_i^0 \xrightarrow{x[i]} q_i^f$ which means for all $i \in Loc$, $x[i] \in L(M_i, F_i)$. Hence $x \in L(M_1, F_1) \parallel \dots \parallel L(M_n, F_n)$.

Proof of claim: We prove the claim by induction on the length of x . When $x = \epsilon$, $p_i = q_i$ for all i . Hence the claim holds trivially. For the induction step, let $x = ya$. If $(p_1, \dots, p_n) \xrightarrow{ya} (q_1, \dots, q_n)$ then $(p_1, \dots, p_n) \xrightarrow{y} (r_1, \dots, r_n) \xrightarrow{a} (q_1, \dots, q_n)$ for some state (r_1, \dots, r_n) in \widehat{M} . Because of the asynchrony property, $\forall i \in loc(a), r_i \xrightarrow{a} q_i$, and $\forall i \notin loc(a), r_i = q_i$. By induction hypothesis, for all i , $p_i \xrightarrow{y[i]} r_i$.

Let $i \in loc(a)$. We have $p_i \xrightarrow{y[i]} r_i \xrightarrow{a} q_i$. This implies $p_i \xrightarrow{x[i]} q_i$.

Now let $i \notin loc(a)$. Then $x[i] = y[i]$. Hence, we get $p_i \xrightarrow{x[i]} q_i$. ■

(\supseteq): Let $x \in L(M_1, F_1) \parallel \dots \parallel L(M_n, F_n)$. Then for all $i \in Loc$, $x[i] \in L(M_i, F_i)$, which implies for all $i \in Loc$, $q_i^0 \xrightarrow{x[i]} q_i^f$ for some $q_i^f \in F_i$.

Claim: For all $x \in \Sigma^*$, (for all $i \in Loc$, $p_i \xrightarrow{x[i]} q_i$) implies $(p_1, \dots, p_n) \xrightarrow{x} (q_1, \dots, q_n)$.

Assume the claim. Then we have $(q_1^0, \dots, q_n^0) \xrightarrow{x} (q_1^f, \dots, q_n^f)$. Since (q_1^f, \dots, q_n^f) is in $\Pi_{i=1}^n F_i$ and $L(\widehat{M}) = L(\widehat{M}, \Pi_{i=1}^n F_i)$, we have $x \in L(\widehat{M})$.

Proof of claim: We prove the claim by induction on length of x again. When $x = \epsilon$, $x[i] = \epsilon$, $i \in Loc$. Hence, for all i , $p_i = q_i$. Then the claim follows trivially. For the induction hypothesis, let $x = ya$. By the assumption, for all $i \in loc(a)$, $p_i \xrightarrow{ya[i]} q_i$.

Let $i \in \text{loc}(a)$. Then, $p_i \xrightarrow{(y \upharpoonright_i) a} q_i$. Hence, there is an $r_i \in Q_i$ such that $p_i \xrightarrow{y \upharpoonright_i} r_i \xrightarrow{a} q_i$.

Let $i \notin \text{loc}(a)$. We have, $p_i \xrightarrow{y \upharpoonright_i} r_i = q_i$.

Thus for all $i \in \text{Loc}$, $p_i \xrightarrow{y \upharpoonright_i} r_i$. Therefore, by induction hypothesis, we have

$$(p_1, \dots, p_n) \xrightarrow{y} (r_1, \dots, r_n).$$

Also by asynchrony property,

$$(r_1, \dots, r_n) \xrightarrow{a} (q_1, \dots, q_n).$$

Hence, finally, $(p_1, \dots, p_n) \xrightarrow{\tau} (q_1, \dots, q_n)$. ■

2.3.2 Syntax for regular shuffle languages

We define a class of languages over Σ through the following syntax. This is presented in a top-level parallel fashion so that it reflects the structure of distributed systems. We build the syntax in two layers: first for 'local' expressions and then for parallel composition.

Let $\tilde{\Sigma}$ be a distributed alphabet. We define $SREG_{\tilde{\Sigma}}$ (*shuffle regular expressions over $\tilde{\Sigma}$*) as follows.

$$SREG_i ::= \emptyset \mid a \in \Sigma_i \mid p + q \mid p; q \mid p^*, \text{ where } p, q \in SREG_i$$

$$SREG_{\tilde{\Sigma}} ::= r_1 \parallel \dots \parallel r_n, \text{ where } r_i \in SREG_i$$

Notice that at the local level, we have regular expressions over the local alphabets. Hence, semantics for elements in REG_i is the usual map $\llbracket \cdot \rrbracket_i : SREG_i \rightarrow 2^{\Sigma_i}$. For the elements of $SREG_{\tilde{\Sigma}}$, we give the semantics via synchronized shuffle.

$$[\emptyset] = \emptyset,$$

$$[a]_i = \{a\},$$

$$[p + q]_i = [p]_i \cup [q]_i,$$

$$[p; q]_i = [p]_i \cdot [q]_i,$$

$$[p^*]_i = ([p]_i)^*, \text{ and finally}$$

$$[r_1 \parallel \cdots \parallel r_n] = [r_1]_1 \hat{\parallel} \cdots \hat{\parallel} [r_n]_n.$$

The class of regular languages generated by the $SREG_{\tilde{\Sigma}}$ expressions is denoted as $\mathcal{L}(SREG_{\tilde{\Sigma}})$. Formally, $\mathcal{L}(SREG_{\tilde{\Sigma}}) = \{L \subseteq \Sigma^* \mid \text{there is an } R \in SREG_{\tilde{\Sigma}} \text{ such that } L = [R]\}$.

2.3.3 A Kleene theorem for regular shuffle languages

We have three classes of languages: $\mathcal{L}(RSL_{\tilde{\Sigma}})$ defined using the $\hat{\parallel}$ operators, $\mathcal{L}(SREG_{\tilde{\Sigma}})$ defined syntactically and $\mathcal{L}(PS_{\tilde{\Sigma}})$ defined via product systems. We show below the equality of these three classes thus proving a Kleene theorem for regular shuffle languages over $\tilde{\Sigma}$.

Theorem 2.28 $\mathcal{L}(RSL_{\tilde{\Sigma}}) = \mathcal{L}(PS_{\tilde{\Sigma}}) = \mathcal{L}(SREG_{\tilde{\Sigma}})$.

Proof: ($\mathcal{L}(RSL_{\tilde{\Sigma}}) \subseteq \mathcal{L}(PS_{\tilde{\Sigma}})$): Since $L \in \mathcal{L}(RSL_{\tilde{\Sigma}})$, there exist regular languages $L_i \subseteq \Sigma_i^*$, for all $i \in Loc$ such that $L = L_1 \hat{\parallel} \cdots \hat{\parallel} L_n$. Since L_i 's are regular, there exist FSA (M_i, F_i) such that $L_i = L(M_i, F_i)$. Take the product system $\tilde{M} = (M_1, \dots, M_n, < F_1, \dots, F_n >)$. By Proposition 2.27, $L(\tilde{M}) = L(M_1, F_1) \hat{\parallel} \cdots \hat{\parallel} L(M_n, F_n)$. Hence $L(\tilde{M}) = L_1 \hat{\parallel} \cdots \hat{\parallel} L_n = L$, which shows $L \in \mathcal{L}(PS_{\tilde{\Sigma}})$.

($\mathcal{L}(PS_{\tilde{\Sigma}}) \subseteq \mathcal{L}(SREG_{\tilde{\Sigma}})$): Let $L \in \mathcal{L}(PS_{\tilde{\Sigma}})$. Then there is a product system $\tilde{M} = ((M_1, F_1), \dots, (M_n, F_n))$ over $\tilde{\Sigma}$ such that $L = L(\tilde{M})$. By Proposition 2.27,

$$L = L(M_1, F_1) \hat{\parallel} \cdots \hat{\parallel} L(M_n, F_n).$$

Since each $L(M_i, F_i)$ is recognizable over Σ_i , we have a regular expression r_i over Σ_i such that $[r_i]_i = L(M_i, F_i)$. Note that $r_i \in SREG_i, i \in Loc$. Take $r = r_1 \hat{\parallel} \cdots \hat{\parallel} r_n$.

We know $r \in SREG_{\tilde{\Sigma}}$. Also, $[r] = [r_1]_1 \hat{\parallel} \cdots \hat{\parallel} [r_n]_n = L(M_1, F_1) \hat{\parallel} \cdots \hat{\parallel} L(M_n, F_n) = L$. Hence $L \in \mathcal{L}(SREG_{\tilde{\Sigma}})$.

$(\mathcal{L}(SREG_{\tilde{\Sigma}}) \subseteq \mathcal{L}(RSL_{\tilde{\Sigma}}))$: Follows from definition of synchronized shuffle. ■

Observation 2.29 Consider a subclass of product systems over a given $\tilde{\Sigma}$ where the component FSA's are deterministic. Call this class $DPS_{\tilde{\Sigma}}$. Then $\mathcal{L}(DPS_{\tilde{\Sigma}}) = \mathcal{L}(PS_{\tilde{\Sigma}})$.

It is obvious that $\mathcal{L}(DPS_{\tilde{\Sigma}}) \subseteq \mathcal{L}(PS_{\tilde{\Sigma}})$. Interestingly, the converse also holds. One way to prove this is by using the Kleene theorem above.

By 2.24(1), we know that $\mathcal{L}(PS_{\tilde{\Sigma}}) \subseteq \mathcal{L}(RSL_{\tilde{\Sigma}})$. When $L \in \mathcal{L}(RSL_{\tilde{\Sigma}})$ there exist regular languages $L_i \subseteq \Sigma_i^*$, for all $i \in Loc$ such that $L = L_1 \hat{\parallel} \cdots \hat{\parallel} L_n$. Now, instead of taking arbitrary FSA's for L_i 's take DFA's (M_i, F_i) that accept L_i . Then, the system comprising these components is a deterministic product system. By Proposition 2.27, this system accepts L . Therefore, $\mathcal{L}(RSL_{\tilde{\Sigma}}) \subseteq \mathcal{L}(DPS_{\tilde{\Sigma}})$. This gives us the inclusion $\mathcal{L}(PS_{\tilde{\Sigma}}) \subseteq \mathcal{L}(DPS_{\tilde{\Sigma}})$.

Thus, deterministic product systems are sufficient to capture the class of regular shuffle languages.

2.3.4 Product systems with global final states

In the light of Corollary 2.14, we know that the class of product systems capture $\mathcal{L}(RSL_{\tilde{\Sigma}})$, a small subclass of Reg_{Σ} . Hence, it motivates us to look at classes of systems that accept richer classes of languages. A natural place to start the exploration is to take boolean closure of $\mathcal{L}(RSL_{\tilde{\Sigma}})$. It is shown in [Zie] (we illustrate this in the next section) that this is also a strict subset of Reg_{Σ} . Nevertheless, we present the class of distributed systems that characterize this class. Our purpose is to illustrate how structure conditions and rule of product construction actually affects the behaviour of distributed systems.

Boolean closure of regular shuffle languages

Definition 2.30 Fix $\tilde{\Sigma}$. Let $\mathcal{L}(BRS\tilde{L}_{\tilde{\Sigma}})$ be the least set that contains $\mathcal{L}(R\tilde{S}\tilde{L}_{\tilde{\Sigma}})$ and is closed under all the boolean operations.

The following class of distributed systems is intended to characterize $\mathcal{L}(BRS\tilde{L}_{\tilde{\Sigma}})$.

Definition 2.31 An **Extended Product of System (EPS)** over $\tilde{\Sigma}$ is a tuple

$\tilde{M} = (M_1, \dots, M_n, F)$, where

1. (M_1, \dots, M_n) is a distributed system over $\tilde{\Sigma}$,
2. $F \subseteq \prod_{i=1}^n Q_i$, and
3. the product automaton is (\hat{M}, F) , where \hat{M} is the complete product TS of \tilde{M} .

Let $EPS_{\tilde{\Sigma}}$ stand for the class of extended product systems over $\tilde{\Sigma}$. The class of languages accepted by $EPS_{\tilde{\Sigma}}$ is denoted as $\mathcal{L}(EPS_{\tilde{\Sigma}})$. Formally,

$$\mathcal{L}(EPS_{\tilde{\Sigma}}) = \{L \subseteq \Sigma^* \mid \text{there is a EPS } \tilde{M} \text{ over } \tilde{\Sigma} \text{ such that } L = L(\tilde{M})\}.$$

The following proposition is an immediate consequence of the definition.

Proposition 2.32 For any distributed alphabet $\tilde{\Sigma}$, $\mathcal{L}(PS_{\tilde{\Sigma}}) \subseteq \mathcal{L}(EPS_{\tilde{\Sigma}})$.

Let $DEPS_{\tilde{\Sigma}}$ be the subclass of EPS 's where the component processes are DFA's. As in the case of product systems, we show that the following proposition holds.

Proposition 2.33 $\mathcal{L}(BRS\tilde{L}_{\tilde{\Sigma}}) = \mathcal{L}(EPS_{\tilde{\Sigma}}) = \mathcal{L}(DEPS_{\tilde{\Sigma}})$.

Proof: Obviously, $\mathcal{L}(DEPS_{\tilde{\Sigma}}) \subseteq \mathcal{L}(EPS_{\tilde{\Sigma}})$. It suffices to prove the following claims.

Claim: [1] $\mathcal{L}(EPS_{\tilde{\Sigma}}) \subseteq \mathcal{L}(BRS\tilde{L}_{\tilde{\Sigma}})$.

Claim: [2] $\mathcal{L}(BRS\tilde{L}_{\tilde{\Sigma}}) \subseteq \mathcal{L}(DEPS_{\tilde{\Sigma}})$.

Proof of claim:[1] Let $L \in \mathcal{L}(EPS_{\tilde{\Sigma}})$. Then there is an EPS $\tilde{M} = (M_1, \dots, M_n, F)$ such that $L = L(\tilde{M}) = L(\hat{M}, F)$, where (\hat{M}, F) is the product of \tilde{M} . Since the product is an FA,

$$L(\hat{M}, F) = \bigcup_{\bar{q}\bar{f} \in F} L(\hat{M}, \bar{q}\bar{f}).$$

Now, since $\bar{q}_f \in \Pi_{i=1}^n Q_i$, it is of the form (q_f^1, \dots, q_f^n) . Consider the product system $\tilde{M}' = (M_1, \dots, M_n, < \{q_f^1\}, \dots, \{q_f^n\} >)$. Then the product automaton of \tilde{M}' is also (\hat{M}, \bar{q}_f) . Hence, $L(\hat{M}, \bar{q}_f) \in \mathcal{L}(PS_{\hat{\Sigma}}) = \mathcal{L}(RSL_{\hat{\Sigma}})$. Therefore, by definition, $L = L(\hat{M}, F) \in \mathcal{L}(BRS L_{\hat{\Sigma}})$.

Note that L is, in fact, a union of languages from $\mathcal{L}(RSL_{\hat{\Sigma}})$. ■

Proof of claim:[2] (By induction on the structure of languages in $\mathcal{L}(BRS L_{\hat{\Sigma}})$).

We need to show that if a given language $L \in \mathcal{L}(RSL_{\hat{\Sigma}})$, then $L \in \mathcal{L}(DEPS_{\hat{\Sigma}})$ and that $\mathcal{L}(DEPS_{\hat{\Sigma}})$ is closed under boolean operations.

Suppose $L \in \mathcal{L}(RSL_{\hat{\Sigma}})$. Then, by Observation 2.29 we have a deterministic product system $\tilde{M} = (M_1, \dots, M_n, < F_1, \dots, F_n >)$ accepting L . We construct the deterministic EPS

$$\tilde{M}' = (M_1, \dots, M_n, \Pi_{i=1}^n F_i).$$

Since the products of both \tilde{M} and \tilde{M}' are same, $L \in \mathcal{L}(DEPS_{\hat{\Sigma}})$.

Suppose $L \in \mathcal{L}(DEPS_{\hat{\Sigma}})$. Then there is a deterministic EPS $\tilde{M} = (M_1, \dots, M_n, F)$ accepting L . Construct $\tilde{M}' = (M_1, \dots, M_n, \Pi_{i=1}^n Q_i - F)$. Then, \tilde{M}' is a DEPS accepting the complement of L .

Suppose $L_1, L_2 \in \mathcal{L}(DEPS_{\hat{\Sigma}})$. Then for $i \in \{1, 2\}$, there are deterministic EPS's $\tilde{M}_i = (M_1^i, \dots, M_n^i, F^i)$, accepting L_1 and L_2 respectively. We use the following construction to get a deterministic EPS accepting $L_1 \cup L_2$.

Given two FSA's $A = (S, \rightarrow_1, s^0, S^f)$ and $B = (T, \rightarrow_2, t^0, T^f)$, if we want to construct an FSA that accepts $L(A) \cup L(B)$, then we can take the exclusive union of the states and transitions and identify the initial states. But this gives us a nondeterministic FA. In order to construct a deterministic FA accepting the union language, define the FA $A \otimes B$ as follows. $A \otimes B = (S \times T \times \{0, 1, 2\} \cup \{DEAD\}, \rightarrow, (s^0, t^0, 0), F)$ where $DEAD$ is a special state, $F = \{(p, q, 0) \mid p \in S^f \text{ and } q \in T^f\} \cup \{(p, q, 1) \mid p \in S^f\} \cup \{(p, q, 2) \mid q \in T^f\}$ and $(p_1, p_2, i) \xrightarrow{a} (q_1, q_2, j)$ iff either

1. $p_1 \xrightarrow{a} q_1, p_2 \xrightarrow{a} q_2$ and $c = d = 0$, or
2. $p_1 \xrightarrow{a} q_1, p_2 = q_2$ and $i \in \{0, 1\}, j = 1$, or

3. $p_1 = q_1, p_2 \xrightarrow{a} {}_2q_2$ and $i \in \{0, 2\}, j = 2$.

For all other states (p_1, p_2, i) and letters $a \in \Sigma$, set $(p_1, p_2, i) \xrightarrow{a} DEAD$. The essential idea is to take transitions on common actions as far as possible (thus avoiding non-determinism) and then taking the individual transitions. One verifies that \rightarrow is a deterministic relation and $L(A \otimes B) = L(A) \cup L(B)$.

Construct an EPS $\tilde{M} = (M_1, \dots, M_n, F)$ with $M_i = M_i^1 \otimes M_i^2$, for all i . Each i -local state is a triple (p, q, c) where p is a state of M_i^1 , q a state of M_i^2 and $c \in \{0, 1, 2\}$. Hence a global state is of the form $((p_1, q_1, c_1), \dots, (p_n, q_n, c_n))$. The set of final states is given as $F = \{((p_1, q_1, c_1), \dots, (p_n, q_n, c_n)) \mid (p_1, \dots, p_n) \in F_1 \text{ and for all } j: c_j \in \{0, 1\}, \text{ or } (q_1, \dots, q_n) \in F_2 \text{ and for all } j: c_j \in \{0, 2\}\}$.

Intuitively, we keep the final states of the systems from getting mixed up.

Then \tilde{M} is a deterministic EPS since each M_i is deterministic. One also verifies that $L(\tilde{M}) = L(\tilde{M}_1) \cup L(\tilde{M}_2) = L_1 \cup L_2$. Hence, $L_1 \cup L_2 \in \mathcal{L}(DEPS_{\tilde{\Sigma}})$. ■

Corollary 2.34 $\mathcal{L}(DEPS_{\tilde{\Sigma}})$ is closed under all boolean operations. Hence, so is $\mathcal{L}(EPS_{\tilde{\Sigma}})$.

2.3.5 Union closure of $\mathcal{L}(RSL_{\tilde{\Sigma}})$

Consider the least class of languages that contains $\mathcal{L}(RSL_{\tilde{\Sigma}})$ and is closed under only union. Call the class $\mathcal{L}(URSL_{\tilde{\Sigma}})$. It is obvious that $\mathcal{L}(URSL_{\tilde{\Sigma}}) \subseteq \mathcal{L}(BRSL_{\tilde{\Sigma}})$. The proof of Claim (1) of Proposition 2.33 shows that $\mathcal{L}(EPS_{\tilde{\Sigma}}) \subseteq \mathcal{L}(URSL_{\tilde{\Sigma}})$, hence the inclusion $\mathcal{L}(BRSL_{\tilde{\Sigma}}) \subseteq \mathcal{L}(URSL_{\tilde{\Sigma}})$ also holds and we get the following result.

Theorem 2.35 $\mathcal{L}(URSL_{\tilde{\Sigma}}) = \mathcal{L}(BRSL_{\tilde{\Sigma}})$.

This theorem implies that the language $([ab]c + [aabb]c)^*$ over $\tilde{\Sigma} = (\{a, c\}, \{b, c\})$ is not in $\mathcal{L}(BRSL_{\tilde{\Sigma}})$. The reason is as follows. By the previous theorem, if the given language $L = ([ab]c + [aabb]c)^*$ was in $\mathcal{L}(BRSL_{\tilde{\Sigma}})$ it could be written as a finite union of regular shuffle languages L_1, \dots, L_k .

Note that two different strings x and y of L having same number of c 's can not be in a single L_i , because, in this case, x (resp. y) can be written as $x = uc[ab]cv$ ($y = uc[aabb]cv'$).

where the strings differ for the first time after a common prefix uc . Then since L_i is a regular shuffle language, $x[1 \parallel_S y[2 = [uc][aab]cv''$, where $v'' = v[1 \parallel_S v'[2$, will also be in L_i and hence in L , a contradiction.

But then L does contain large strings where number of c 's is more than k , hence there are more than k strings with same number of c 's. By pigeon-hole principle, some L_i then contains two different strings with same number of c 's and hence this leads to contradiction as observed. Therefore, $([ab]c + [aabb]c)^*$ is not in $\mathcal{L}(BRSL_{\tilde{\Sigma}})$.

2.3.6 Syntax for $\mathcal{L}(BRSL_{\tilde{\Sigma}})$

The syntax proposed for $\mathcal{L}(BRSL_{\tilde{\Sigma}})$ is same as shuffle regular expressions ($SREG$) in the local level. The only change is in the global level where we allow union (+) over global expressions (note that in light of Theorem 2.35, complementation is not necessary). The expressions are called $BSREG_{\tilde{\Sigma}}$ (boolean shuffle regular expressions over $\tilde{\Sigma}$).

$$BSREG_i ::= \emptyset \mid a \in \Sigma_i \mid p + q \mid p; q \mid p^*, \text{ where } p, q \in BSREG_i.$$

$$\begin{aligned} BSREG_{\tilde{\Sigma}} ::= & r_1 \parallel \dots \parallel r_n, \quad r_i \in BSREG_i, i = 1, \dots, n \\ & \mid R_1 + R_2, \quad R_1, R_2 \in BSREG_{\tilde{\Sigma}}. \end{aligned}$$

The semantics is as usual, with $[R_1 + R_2] = [R_1] \cup [R_2]$.

The class of regular languages generated by the $BSREG_{\tilde{\Sigma}}$ expressions is denoted as $\mathcal{L}(BSREG_{\tilde{\Sigma}})$. Formally,

$$\mathcal{L}(BSREG_{\tilde{\Sigma}}) = \{L \subseteq \Sigma^* \mid \text{there is an } R \in BSREG_{\tilde{\Sigma}} \text{ such that } L = [R]\}.$$

Observe that using Proposition 2.33 and Theorem 2.35, one can prove a Kleene theorem for the class of boolean closure of regular shuffle languages ($\mathcal{L}(BRSL_{\tilde{\Sigma}})$).

Theorem 2.36 $\mathcal{L}(BRSL_{\tilde{\Sigma}}) = \mathcal{L}(EPS_{\tilde{\Sigma}}) = \mathcal{L}(BSREG_{\tilde{\Sigma}})$.

2.4 Other automata models for distributed systems

There are several other models based on finite state automata that model finite state distributed systems. The essence of every such model is that some global information (e.g. global states or global transitions) is presented in a distributed manner. But as opposed to the product systems we saw in the previous section, these are not local presentations because either they operate with global states or with global transitions and are not composed from the local ones by some uniform construction.

2.4.1 Asynchronous transition systems

These are transition systems with an independence relation on transitions [Bed]. The idea is that this models the spatial distribution of actions in the system and in a global behaviour these independent transitions can occur in any order. We present a variant of these systems where independence of transitions is determined by their labels.

Formally, an asynchronous transition system (ATS) on a finite alphabet Σ is a tuple $M = (Q, \rightarrow, q^0, I, F)$ where (Q, \rightarrow, q^0) is a transition system on Σ , F is a set of final states and $I \subseteq \Sigma^2$ is an irreflexive, symmetric relation. The transitions are required to satisfy what are called *diamond* conditions.

Let $p, q, r \in Q$ and $a, b \in \Sigma$ such that $(a, b) \in I$. Then,

1. if $p \xrightarrow{a} q$ and $p \xrightarrow{b} r$ then there is a unique $s \in Q$ such that $q \xrightarrow{b} s$ and $r \xrightarrow{a} s$.
2. if $p \xrightarrow{a} q$ and $q \xrightarrow{b} s$ then there is a unique $r \in Q$ such that $p \xrightarrow{b} r$ and $r \xrightarrow{a} s$.

ATS's accept the class of regular consistent languages over Σ with I as the independence relation. But they operate on global states, and there is no notion of components at all. It is not at all clear how to distribute the states and transitions of arbitrary ATS's. Distribution of ATS's satisfying special properties have been studied in [WN, Muk]. But, as defined, ATS's are clearly not local presentations.

2.4.2 Distributed transition systems

These are transition systems where the transitions are labelled by subsets of actions [LPRT]. The idea is that the actions in this subset are executed concurrently.

A distributed transition system (DTS) is a tuple $M = (Q, \longrightarrow, q^0, F)$ where, the transition relation $\longrightarrow \subseteq Q \times 2^\Sigma \times Q$ satisfies:

1. $p \xrightarrow{\emptyset} q$ iff $p = q$,
2. if $p \xrightarrow{E} q$ then $\exists h : 2^E \rightarrow Q$ such that
 - (a) $h(\emptyset) = p$,
 - (b) $h(E) = q$, and
 - (c) $\forall E_1 \subseteq E_2 \subseteq E, h(E_1) \xrightarrow{E_2 - E_1} h(E_2)$.

The definition says that every concurrent step can be “broken up” into sub-steps in every possible way. For instance, doing a, b and c can be first done with a and b concurrently and then c or in the order b, c, a or ... Note that in multiprocessor implementation of distributed systems where we have n processes and k processors with $2 \leq k \leq n$, such situations are important and interesting, and distinct from the mere statement that concurrent steps may be executed in any order.

DTS's implicitly have a notion of sequential components in them, and the idea of *regions* has been used in [NRT] to extract such components. But such studies are carried out category theoretically and whether one can decompose finite state DTS's into communicating FSA's preserving language behaviour remains an open question. However, from the definition it is clear that DTS's do not have explicit local presentation.

2.4.3 Zielonka automata

A Zielonka automaton [Zie] on $\tilde{\Sigma}$ with n processes is a tuple $\mathcal{A} = (A_1, \dots, A_n, \Delta, F)$, where for every $i \in Loc$, $A_i = (Q_i, \Sigma_i, \Delta_i, s_i^0)$ is the i -th automaton. Let $\hat{Q} = \prod_{i \in Loc} Q_i$ and let Q_a denote $\prod_{i \in loc(a)} Q_i$. $F \subseteq \hat{Q}$ is the set of final states.

The transition relation is $\Delta = \{\delta_a \mid a \in \Sigma\}$, where $\delta_a \subseteq (Q_a \times Q_a)$. For each δ_a , $(q_{i_1}, \dots, q_{i_k}) \in \delta_a(p_{i_1}, \dots, p_{i_k})$, with $\{i_1, \dots, i_k\} = \text{loc}(a)$ implies

$$\text{for all } i_j \in \text{loc}(a), (p_{i_j}, a, q_{i_j}) \in \Delta_{i_j}.$$

\mathcal{A} is *deterministic* if $\forall a \in \Sigma$, δ_a is a function on Q_a .

The behaviour of \mathcal{A} is given via the associated global automaton \hat{A} which is defined as: $\hat{A} = (Q, \Rightarrow_{\mathcal{A}}, (s_1^0, \dots, s_n^0), F)$, where the transition relation $\Rightarrow_{\mathcal{A}}$ is defined as:

$$(p_1, p_2, \dots, p_n) \xRightarrow{\mathcal{A}} (q_1, q_2, \dots, q_n) \text{ iff } (q_{i_1}, \dots, q_{i_k}) \in \delta_a(p_{i_1}, \dots, p_{i_k}),$$

where $\{i_1, \dots, i_k\} = \text{loc}(a)$, and $p_j = q_j$, for all $j \notin \text{loc}(a)$.

This transition among the global states is extended to words over Σ^* in the natural way. An immediate corollary of the transition on global states is the following.

Proposition 2.37 *If $(a, b) \in \mathcal{I}$ then for all $s, s' \in Q$, $s \xRightarrow{ab}_{\mathcal{A}} s'$ iff $s \xRightarrow{ba}_{\mathcal{A}} s'$. Consequently $L(\mathcal{A})$ is closed under \sim .*

The language accepted by \mathcal{A} is defined as: $L(\mathcal{A}) = \{x \in \Sigma^* \mid \exists s \in F. s^0 \xRightarrow{x}_{\mathcal{A}} s\}$. Then from the above proposition, we know that $L(\mathcal{A})$ is consistent. Further, since the global automaton \hat{A} is an FSA, $L(\mathcal{A}) \in RCL_{\tilde{\Sigma}}$. Zielonka's theorem states that the converse is also true i.e., for every $L \in RCL_{\tilde{\Sigma}}$ there is a Zielonka automaton \mathcal{A} such that $L = L(\mathcal{A})$. Thus the class of Zielonka automata over $\tilde{\Sigma}$ characterize $RCL_{\tilde{\Sigma}}$.

Theorem 2.38 (Zielonka)

1. For every Zielonka automaton \mathcal{A} , $L(\mathcal{A}) \in RCL_{\tilde{\Sigma}}$.
2. For every regular consistent language $L \in RCL_{\tilde{\Sigma}}$ there exists a deterministic Zielonka automaton \mathcal{A} such that $L = L(\mathcal{A})$.

In Zielonka automata, global states are distributed and the transitions are also distributed to some extent in the sense that independent transitions take place asynchronously. But yet global transitions are not *derived* from the local transitions since the transitions depend upon the states of the participating agents. In a sense, there is a choice of global transitions and this choice is not given by any universal rule on the class of Zielonka automata;

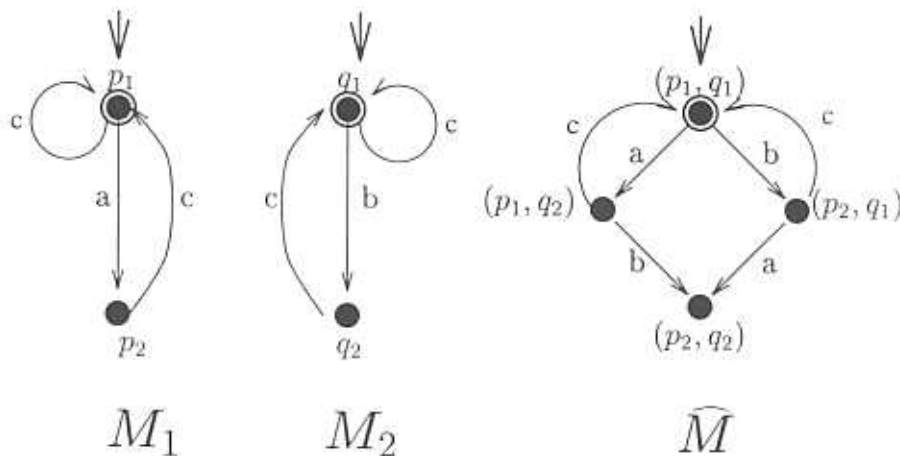


Figure 2.3: A Zielonka automaton for $(ac + bc)^*$ over $\langle \Sigma_1 = \{a, c\}, \Sigma_2 = \{b, c\} \rangle$.

hence they are not locally presented. For example, even if $(p_1, a, p_2) \in \Delta_1$ $(q_1, a, q_2) \in \Delta_2$ it may happen that $(q_1, q_2) \notin \delta_a(p_1, p_2)$. In Fig. 2.3, we illustrate this through a Zielonka automaton accepting the language $(ac + bc)^*$ over the alphabet $\Sigma_1 = \{a, c\}, \Sigma_2 = \{b, c\}$.

Example See Fig. 2.3. The action c is enabled at all the global states. But $\delta_c = \{((p_2, q_1), (p_1, q_1)), ((p_1, q_2), (p_1, q_1))\}$. Note, how *choosing* the right global transitions, it is easy to avoid spurious strings like c, abc etc.

2.4.4 Local presentation for RCL's

Asynchronous automata exactly characterize RCL[Zie]. Ochmanski's theorem[Och] provides a Kleene characterization for this class of languages. But as we saw neither are these automata locally presented nor is the distributed nature of the system reflected in the syntax given for RCL's by Ochmanski's theorem.

We saw that the product systems in this chapter provided local presentation for some simple class of languages. We want to ask whether such presentation is possible for RCL's as well. In order to do so, we need to encode some global information in the local states and give a rule for product construction that filters out the extra transitions which would result in a simple product. One such formulation is presented in Chapter 3. Though this is quite different from the main framework we offer in a later chapter, this is a first

attempt at distribution of global behaviour and follows from easy intuition.

2.5 Infinite behaviour and ω -regular languages

Operating systems, control systems, communication protocols, hardware systems (e.g. processors) are examples of systems that are *reactive* [Pnu] in nature. Their purpose is not just a transformation of values but to maintain an on-going interaction with their environment. Study of the infinite, non-terminating computation of these systems is a central issue in the theory of automata and languages [Tho]. These infinite computations are modelled by ω -strings over a give finite alphabet Σ . Sets of such ω -strings are called ω -languages over Σ .

2.5.1 Finite state systems for infinite behaviour

In the case of finite state systems, infinite behaviours exhibit certain regularity. As in the case of finite behaviours, one can express these behaviour using automata where, expectedly, the nature of acceptance is very different because now the automata have to be over infinite words.

Let Σ be a finite nonempty alphabet and let Σ^ω denote the set of all infinite strings over Σ . Given a TS $M = (Q, \longrightarrow, q^0)$ on Σ , a *run* ρ on an infinite string $x = a_1 a_2 \dots$ starting at $p_0 \in Q$ is a sequence of transitions of the automaton : $p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \dots$. We denote by $\rho(i)$ the i 'th state p_i on the run ρ . Let $Inf(\rho)$ denote the set of states that occur infinitely often in ρ , that is,

$$Inf(\rho) = \{q \mid \exists^\infty i, i \geq 0 \text{ such that } \rho(i) = q\}.$$

A run starting at the initial state q^0 is called a *run of M on x* .

Automata over infinite words are essentially TS's over Σ with acceptance conditions for infinite runs. We call them collectively as ω -automata. They are called *deterministic* or *non-deterministic* depending upon the nature of the transition relation of the underlying TS.

ω -automata were first studied by Büchi. Similar to the way finite automata accept strings which lead to a designated final state, these automata, which are again finite state machines, accept infinite strings which cause one or more of several designated states to be visited infinitely often.

Büchi automaton A Büchi automaton over Σ is a pair (M, \mathcal{B}) where $M = (Q, \rightarrow, q^0)$ is a TS over Σ and $\mathcal{B} \subseteq Q$ is called a *Büchi condition*. A run ρ of M on an infinite string x is said to be accepting if $\text{Inf}(\rho) \cap \mathcal{B} \neq \emptyset$. In this case x is accepted by (M, \mathcal{B}) . We define the ω -language accepted by (M, \mathcal{B}) as $L(M, \mathcal{B}) = \{x \in \Sigma^\omega \mid x \text{ is accepted by } (M, \mathcal{B})\}$.

Muller automaton A Muller automaton over Σ is a pair (M, \mathcal{T}) , where $M = (Q, \rightarrow, q^0)$ is a TS over Σ and $\mathcal{T} \subseteq 2^Q$ is called a *Muller condition* or a *Muller table*. A run ρ of M on an infinite string x is said to be accepting if there exists an $F \in \mathcal{T}$ such that $\text{Inf}(\rho) = F$.

It turns out that *deterministic* Muller automata suffice (but that non-deterministic Büchi automata are *required*) to accept the class of ω -regular languages [Tho].

At times, it is convenient to have Büchi automata with not one but a set of acceptance condition as described below.

Büchi automaton with multiple conditions A Büchi automaton with multiple conditions over Σ is a pair $(M, \{\mathcal{B}_1, \dots, \mathcal{B}_k\})$ where $M = (Q, \rightarrow, q^0)$ is a TS over Σ and $\mathcal{B}_j \subseteq Q$. A run ρ of M on an infinite string x is said to be accepting if $\text{Inf}(\rho) \cap \mathcal{B}_j \neq \emptyset$, for every $j \in \{1, \dots, k\}$.

The class of Büchi automata with multiple conditions are no more powerful than that with single condition in terms of languages acceptance. This can be shown by transforming the former to one having a single acceptance condition, by maintaining a counter (alongwith the states) that cycles through each condition.

2.5.2 Syntax and McNaughton's theorem

Similar to regular languages over finite length strings, one can define a syntax for ω -regular languages over infinite strings.

$$\begin{aligned} \omega\text{Reg}_\Sigma &::= R \cdot S^\omega, \quad R, S \in \text{Reg}_\Sigma \\ &\mid T_1 + T_2, \quad T_i \in \omega\text{Reg}_\Sigma \end{aligned}$$

The semantics of these elements are defined naturally. Similar to Kleene's theorem in the finite case, McNaughton's theorem [Tho] establishes the correspondence between the automata and syntax for ω -regular languages.

Theorem 2.39 *$\mathcal{L}(\omega\text{Reg}_\Sigma)$ is exactly characterized by non-deterministic Büchi automata and deterministic Muller automata over Σ .*

2.5.3 Simulations for ω -automata

Given two Büchi automata (M, \mathcal{B}) and (M', \mathcal{B}') , suppose M' simulates M . In the proof of Proposition 2.4, we showed that every (finite)path of M is simulated by a path of M' . The same result can be proved for infinite paths as well. Then, by suitable assignment of acceptance conditions, M' can exactly simulate the behaviour of M .

Proposition 2.40 *Let (M, \mathcal{B}) and (M', \mathcal{B}') are two Büchi automata such that M' simulates M via Θ and $\mathcal{B}' = \{p' \in Q' \mid \Theta(p') \in \mathcal{B}\}$. Then $L(M, \mathcal{B}) = L(M', \mathcal{B}')$.*

Similar result also holds for simulation between Muller automata.

Proposition 2.41 *Let (M, \mathcal{T}) and (M', \mathcal{T}') are two Muller automata such that M' simulates M via Θ and for every $F \in \mathcal{T}$, there is $F' \in \mathcal{T}'$ such that $F' = \{p' \in Q' \mid \Theta(p') \in F\}$. Then $L(M, \mathcal{T}) = L(M', \mathcal{T}')$.*

2.5.4 FSDS's and ω -languages

When we want to capture infinite behaviour of finite state distributed systems, a slight complication is introduced because of the distributed nature of the alphabet, since an infinite behaviour x may actually be constituting finite behaviour of some agents. For example, let $\Sigma_1 = \{a\}, \Sigma_2 = \{b\}$ and let $x = ab^\omega$. Then, even though x is an infinite behaviour of the system, the behaviour of agent 1 is just a , which is finite.

It is possible to accept such behaviour by having acceptance conditions for both finite and infinite behaviour. But the treatment then becomes messy without adding to intuition. Hence, in the following and in the rest of the thesis, *by infinite behaviour, we consider only those in which all the agents also exhibit infinite behaviour.*

Formally, given a distributed alphabet $\tilde{\Sigma}$,

Σ^ω contains only those strings x such that for all $i \in Loc$, $x[i]$ is infinite.

2.5.5 Products and infinite behaviour

Mirroring the questions we asked in case of FSDS's, we want to know what kind of infinite behaviour is exhibited by products of ω -automata. Essentially, we want to know whether the results that were true for finite behaviour extend to infinite behaviour or not.

We recall the definition of shuffle for finite strings. Let Σ_1 and Σ_2 be two alphabets and let $x_i \in \Sigma_i^*$. Then,

$$x_1 \parallel x_2 = \{x \mid x[i] = x_i, i = 1, 2\},$$

We extend this definition naturally to infinite strings.

As a natural extension of the notion of product systems in the finite case, one can define a collection of Büchi automata over a given $\tilde{\Sigma}$. To simplify notation, assume $Loc = \{1, 2\}$. Let (M, \mathcal{F}) and (N, \mathcal{G}) be two Büchi automata over Σ_1 and Σ_2 respectively.

As in the finite case, suppose the product is defined as (\hat{O}, \mathcal{H}) where \hat{O} is the complete product TS and $\mathcal{H} = (\mathcal{F} \times \mathcal{G})$. Then, we expect the following result, namely, $L(\hat{O}, \mathcal{H}) = L(M, \mathcal{F}) \parallel L(N, \mathcal{G})$. Unfortunately this is not the case.

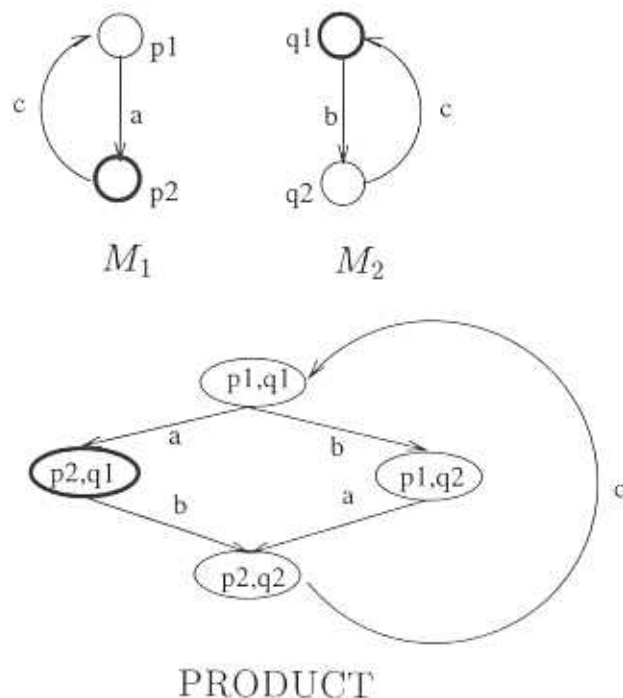


Figure 2.4: Product of ω -automata.

The first inclusion $L(\hat{O}, \mathcal{H}) \subseteq L(M, \mathcal{F}) \parallel L(N, \mathcal{G})$ is easy to prove. But we note immediately that the converse is not true. The following example illustrates this fact.

Example See Fig. 2.4. The local Büchi conditions are $\mathcal{F} = \{p_2\}$ and $\mathcal{G} = \{q_1\}$ respectively. Hence the Büchi condition for the product is $\mathcal{H} = \{(p_2, q_1)\}$.

Let $x = (bac)^\omega$. Clearly, $x \in L(M, \mathcal{F}) \parallel L(N, \mathcal{G})$. But $\text{Inf}(\rho) = \{(p_1, q_1), (p_1, q_2), (p_2, q_2)\}$. Hence, $\text{Inf}(\rho) \cap \mathcal{H} = \emptyset$, which implies that $x \notin L(\hat{O}, \mathcal{H})$.

In the previous example, the problem seems to be in taking the product of the individual Büchi conditions, because the states from individual Büchi conditions (p_2 and q_1 resp.), though they occur independently infinitely many times on the path for x , they do not occur simultaneously to give a state from the global Büchi condition. With this intuition, we take a different acceptance condition for the product. This is as follows:

Fix $\tilde{M} = ((M_1, \mathcal{B}_1) \cdots (M_n, \mathcal{B}_n))$, where (M_i, \mathcal{B}_i) are Büchi automata over Σ_i . Let $\hat{M} = (\hat{Q}, \rightarrow, \hat{q}^0)$ be the complete product TS of \tilde{M} .

Definition 2.42 Let $x \in \Sigma^\omega$. Given an infinite run ρ on x in \widehat{M} on $x \in \Sigma^\omega$, the set of global states occurring infinitely often in ρ is defined as

$$Inf(\rho) \stackrel{\text{def}}{=} \{\bar{s} \in \widehat{Q} \mid \exists^\infty j, \rho(j) = s\}.$$

The infinitely occurring i -local states in ρ is defined as:

$$Inf_i(\rho) \stackrel{\text{def}}{=} \{q \in Q_i \mid \exists^\infty j, \rho(j) = s \text{ and } s[i] = q\}.$$

Definition 2.43 Given $\widehat{M} = ((M_1, B_1) \cdots, (M_n, B_n))$, and the complete TS \widehat{M} as above. \widehat{M} is called a ω Product System with local Büchi condition when the product automaton is given as $(\widehat{M}, < B_1, \dots, B_n >)$ and the acceptance condition is given as follows.

A string $x \in \Sigma^\omega$ is accepted if there is a run ρ on x in \widehat{M} such that for all $i \in Loc, Inf_i(\rho) \cap B_i \neq \emptyset$.

One sees that if we take (M, \mathcal{F}) and (N, \mathcal{G}) as components of a ω Product System with local Büchi condition the problematic case turns out to be true. Thus the shuffle of ω -regular languages does have a local presentation as in the finite case.

Proof of $L(\widehat{O}, < \mathcal{F}, \mathcal{G} >) \supseteq L(M, \mathcal{F}) \parallel L(M, \mathcal{G})$:

Let $x \in L(M, \mathcal{F}) \parallel L(M, \mathcal{G})$. Then, $x[1] \in L(M, \mathcal{F})$. Thus there is an infinite run ρ_1 on $x[1]$ in M such that $Inf(\rho_1) \cap \mathcal{F} \neq \emptyset$. Similarly, we get an infinite ρ_2 on $x[2]$ in N such that $Inf(\rho_2) \cap \mathcal{G} \neq \emptyset$.

It is now easy to construct a run ρ in the product \widehat{O} for x inductively such that for $i = 1, 2, \rho[i] = \rho_i$, in which case, $Inf_i(\rho) = Inf(\rho_i)$.

Hence, $Inf_1(\rho) \cap \mathcal{F} \neq \emptyset$ and also $Inf_2(\rho) \cap \mathcal{G} \neq \emptyset$. Thus x is accepted by the product $(\widehat{O}, < \mathcal{F}, \mathcal{G} >)$. ■

2.5.6 Other classes of infinite behaviour

Similar to the classes of boolean closure of regular shuffle languages and the regular consistent languages, we could consider their counterparts over infinite strings. But our aim was not to

explore these languages in the present chapter except to introduce the concept of product of infinite behaviour. We defer the analysis of these till Chapter 5 where we show that the local presentation described there for finite behaviour smoothly extends to infinite behaviour, thus giving us product(-like) systems for the mentioned behaviour.

3 View-based presentation

In this chapter we explore a way of distribution of global behaviour that gives us a local presentation for the class of regular consistent languages. The central idea comes from [Ram1]. In the process models under our study, where a fixed finite number of *processes* proceed asynchronously and periodically exchange information between each other, the view of the system's global state available to any process at any instant of time is necessarily partial. Communication between processes is principally a mechanism for sharing system views and updating views based on information received.

For instance, consider a system of 4 processes P_1, P_2, P_3, P_4 which manipulate local variables x, y, z, w respectively (See Fig. 3.1). Suppose that the only local computations are increment (by 1), which are shown as horizontal rectangles. Suppose also that when the processes exchange information (denoted in the figure by horizontal lines with blobs denoting participating agents in the exchange), they know about the values of the variables of the participating agents. Lastly, suppose that initial value of all variables are zero and all the processes know this. Denote by $K(P_i)$ the values of variables of which P_i has knowledge. So, initially, $K(P_i) = \langle 0, 0, 0, 0 \rangle$ for all processes.

After a period of local computations, P_1 and P_2 interact through the action a ; simultaneously and independently, P_3 and P_4 interact via b . At this stage, actual values of variables are $\langle x, y, z, w \rangle = \langle 1, 0, 1, 0 \rangle$. But since P_1 has no way of knowing the value of z , $K(P_1) = \langle 1, 0, 0, 0 \rangle$; because of the action a , $K(P_2)$ also is $\langle 1, 0, 0, 0 \rangle$. Similarly,

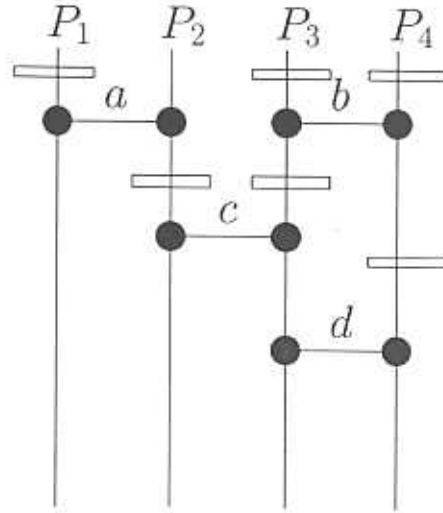


Figure 3.1: Interacting processes.

$K(P_3) = \langle 0, 0, 1, 1 \rangle$ and $K(P_4) = \langle 0, 0, 1, 1 \rangle$. So, P_2 has more recent information than P_3 about the value of x , and P_3 has better information about w than P_2 .

Now suppose, P_2 and P_3 increment their local variables and then exchange information via c . Since P_1 and P_2 do not participate in this action, their knowledge about the values remain same. But, now, $K(P_2) = \langle 1, 1, 2, 1 \rangle = K(P_3)$. Note that even if P_2 never directly interacted with P_4 , it gets the information about w indirectly through P_3 and hence can update its information about w . This sort of view updating is common in distributed protocols.

In [Ram1], distributed transition systems for such models were studied. These systems were locally presented in our sense and reflected the ideas discussed above. Informally, these systems are structured as follows:

- Local states of agents in the system describe their **views** of the system as a whole.
- Synchronization is the exchange of information whereby the synchronizing agents update their views so that they have **identical** views immediately after synchronization.

It was shown in [Ram1] that these transitions systems exactly described (in a categorical sense) a class of labelled event structures called *synchronization structures*. Since

these event structures arise naturally from traces (*a la* Mazurkiewicz) one would like to conjecture that similar finite state transition systems with some acceptance condition will accept regular trace languages (or the RCL's of Chapter 2). Then this would give us a presentation of trace behaviour in terms of products of automata. In this chapter, we show that this is indeed the case.

3.1 View-based systems

Let M_1, \dots, M_n be the component FSA's of a system \tilde{M} over the distributed alphabet $\tilde{\Sigma}$ where $M_i = (Q_i, \delta_i, q_i^0)$. We observe that in the case of product systems of Chapter 2, we have implicitly assumed that whenever $i \neq j, Q_i \cap Q_j \neq \emptyset$. Consider a situation when this is not the case, i.e., the local processes **share** some states. Below, we find some use for these **shared states**. We will insist that whenever transition synchronizes some automata, in the resulting global state, the local states of these automata must be identical.

Definition 3.1 A global transition $(p_1, \dots, p_n) \xrightarrow{a} (q_1, \dots, q_n)$ is said to be a perfect exchange if $\forall i, j \in \text{loc}(a) : q_i = q_j$.

Definition 3.2 A **View-based System (VS)** over $\tilde{\Sigma}$ is a tuple $\tilde{M} = (M_1, \dots, M_n, F)$, where the product automaton is $\hat{M} = (\hat{M}, F)$, such that $\hat{M} = (\hat{Q}, \longrightarrow, (q_1^0, \dots, q_n^0))$ is the complete product TS of \tilde{M} and each transition in \longrightarrow is a perfect exchange.

Remark: Note that the global transitions now satisfy the asynchrony condition and are perfect exchanges as well. Thus,

$$(p_1, \dots, p_n) \xrightarrow{a} (q_1, \dots, q_n) \text{ iff}$$

1. $\forall i \in \text{loc}(a) : p_i \xrightarrow{a} q_i$,
2. $\forall j \notin \text{loc}(a) : p_j = q_j$, and
3. $\forall i, j \in \text{loc}(a) : q_i = q_j$.

As usual, we extend the one step transition function \Rightarrow to words over Σ^* . Then the product accepts a string $x \in \Sigma^*$ if there is a state \bar{q} in F such that $(q_1^0, \dots, q_n^0) \xRightarrow{x} \bar{q}$ and the language accepted by \tilde{M} is given as $L(\tilde{M}) = \{x \in \Sigma^* \mid x \text{ is accepted by } \tilde{M}\}$.

The class of languages over $\tilde{\Sigma}$ accepted by view-based systems is denoted as $\mathcal{L}(VS)$. Formally, $\mathcal{L}(VS) = \{L \subseteq \Sigma^* \mid \text{there is a VS } \tilde{M} \text{ over } \tilde{\Sigma} \text{ such that } \tilde{M} \text{ accepts } L\}$.

Example In Figure 2, we have an example of a system over the distributed alphabet $(\{a, c\}, \{b, c\})$, whose product with final states $\{(s, s), (t, t)\}$ accepts the language $([ab]c + [aabb]c)^*$.

We refrain from drawing the full product which is quite messy. But it is quite easy to follow that each string in the language either leads to the global state (s, s) or (t, t) . On the other hand, strings like $aabc$ that are not in the language “get stuck” and do not lead to any final state. This “preventive” mechanism crucially depends upon *perfect exchange* in transitions on c .

3.2 Equivalence of $\mathcal{L}(VS)$ and RCL over $\tilde{\Sigma}$

In the rest of this chapter we show that view-based systems characterize regular consistent languages. Formally, we prove the following theorem.

Theorem 3.3 $\mathcal{L}(VS)_{\tilde{\Sigma}} = RCL_{\tilde{\Sigma}}$.

The left to right inclusion is easily seen from the following proposition.

Proposition 3.4 Let \mathcal{I} be the independence relation associated with $\tilde{\Sigma}$. If $(a, b) \in \mathcal{I}$ then for all $s, s' \in Q$, $s \xRightarrow{ab} s'$ iff $s \xRightarrow{ba} s'$. Hence, for all $x, y \in \Sigma^*$, if $x \sim y$, $x \in L(\tilde{M})$ iff $y \in L(\tilde{M})$.

In order to prove the second part, we refer to Zielonka’s theorem, by which there exists a deterministic Zielonka automaton \mathcal{A} which accepts a regular consistent language L . Then we distribute the transitions of \mathcal{A} to build a view-based system \tilde{M} and show that \tilde{M} accepts L too [MR1]. The following section is a preparation for such distribution.

3.3 View

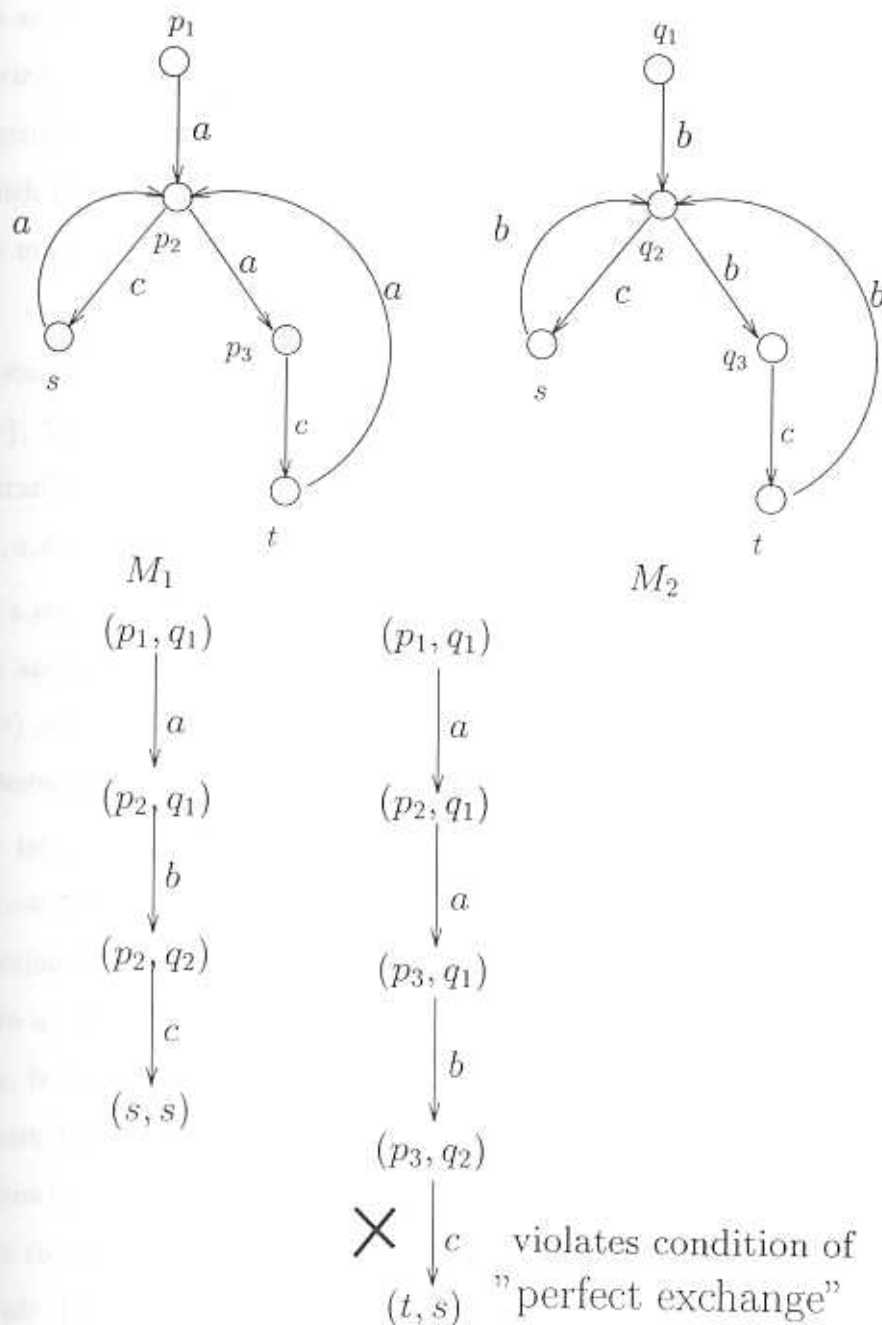


Figure 3.2: A view-based system for $([ab]c + [aabb]c)^*$.

3.3 Views

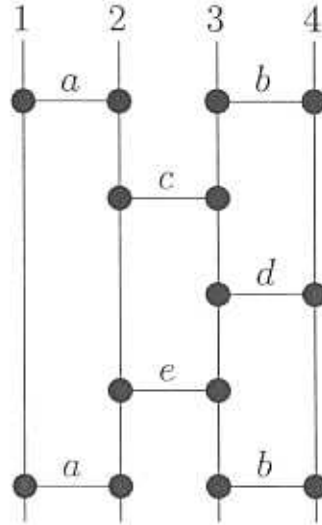
As we noted in the introduction, associated with any computation on a string $x \in \Sigma^*$ of actions, each agent (or a group of agents) has only a partial view. One natural way to define this partial view for an agent is to take the effect of only (and all) those actions such that either the agent participates in it or there is a causal chain from the action to some other action in which the agent participates. This allows for the possibility that computation by other agents may go on independently, of which the concerned agent does not have any information.

For example, take a distributed alphabet $\tilde{\Sigma} = (\Sigma_1, \Sigma_2, \Sigma_3, \Sigma_4)$, where $\Sigma_1 = \{a\}$, $\Sigma_2 = \{a, c, e\}$, $\Sigma_3 = \{b, c, d, e\}$ and $\Sigma_4 = \{b, d\}$. A subsequence of a string is formed by picking arbitrarily letters in the same order as in the given string. Hence, the subsequences of abc are $\{\epsilon, a, b, c, ab, bc, ac, abc\}$.

Call a string causally connected if consecutive actions in the string have a common participating agent. For example, $cdea$ is causally connected, since $loc(c) \cap loc(d) = \{3\}$, $loc(d) \cap loc(e) = \{3\}$, $loc(e) \cap loc(a) = \{2\}$. Similarly, ac , acb and acd are all causally connected whereas ab and ad are not causally-connected.

Now let $x = abcdeab$. The computation associated with x is shown in Fig. 3.3. Then the 1-view (the view of agent 1) is $abcdea$. Note that even if 1 does not participate in the second action b , there is a causally connected subsequence $bcdea$ at the end of which 1 participates in a . Hence this occurrence of b should be considered in the 1-view. The last b in x , however, is not in 1-view because neither 1 participates in it nor is there any causally connected chain to some 1-action (actions from Σ_1). When we consider group views, we consider actions in which at least one in the group participates and the actions with causally related chains to any action of the former kind. Then, in the example, we have $\{1, 3\}$ -view of x is $abcdeab$. Reasoning along this line, we give some i -views and group-views for the string $abcdeab$ in the following table.

Figure 3.3: Distributed computation on the string $abcdeab$.



$abcdeab$

<i>i</i> -views		group-views	
1-view	$abcdea$	$\{1,2\}$ -view	$abcdea$
2-view	$abcdea$	$\{2,3\}$ -view	$abcdeab$
3-view	$abcdeb$	$\{3,4\}$ -view	$abcdeb$
4-view	$abcdeb$	$\{1,2,3,4\}$ -view	$abcdeab$

When we construct the view-based system for a given regular consistent language L , the local states are constructed from the views associated with strings. The existence of A accepting L essentially means that these views can be captured by a finite number of states. The exact correspondence between the view-based system and the language is then established by the connection between views and states of the Zielonka automaton. In order to get this connection, we formalize our intuition about views. Since the group view is more general than the individual view (which is a group with a single individual), we define group views in follows.

Given two strings x and y of Σ^* , x is called a *subsequence* of y , denoted $x \ll y$, if $x = a_1 \dots a_n$ and there exist strings $x_0, \dots, x_n \in \Sigma^*$ such that $y = x_0 a_1 x_1 a_2 x_2 \dots a_n x_n$. Note that \ll is a partial order on Σ^* .

Let $\alpha \subseteq Loc$ and $u = a_1 \dots a_n$. For any j , $1 \leq j \leq n$, a_j is said to have an α -chain in u if there is a sequence $j = i_0 < i_1 < i_2 < \dots < i_m \leq n$ such that

- $loc(a_{i_m}) \cap \alpha \neq \emptyset$, and
- $\forall k, 0 \leq k < m, loc(a_{i_k}) \cap loc(a_{i_{k+1}}) \neq \emptyset$.

In this case, $a_j a_{i_1} a_{i_2} \dots a_{i_m}$ is called an α -chain of u .

Informally, an α -chain of u is a causally connected subsequence of u ending in an α -action (an action with a participant from the set of agents α).

In the previous example, if $u = cad$, then a does not have a $\{4\}$ -chain in u since $4 \notin loc(a)$ and $loc(a) \cap loc(d) = \emptyset$. On the other hand, b has a $\{4\}$ -chain cd since $loc(c) \cap loc(d) = \{3\}$ and $4 \in loc(d)$.

For $\alpha \subseteq Loc$ and $u = a_1 \dots a_n \in \Sigma^*$, u is said to be α -connected iff $\forall j \in \{1, \dots, n\}$, a_j has an α -chain in u . For the example distributed alphabet, cad is not 4-connected (a does not have any 4-chain) while $caed$ is 4-connected because now a has the 4-chain aed .

Let $C_\alpha(x)$ denote the α -connected subsequences of x . Essentially, $C_\alpha(x)$ contains all the causally connected chains that are candidates for the α -view of x . We observe that if $x \ll y$ then $C_\alpha(x) \subseteq C_\alpha(y)$. Also if $\alpha \subseteq \beta$, then $C_\alpha(x) \subseteq C_\beta(x)$.

The following definition is given in anticipation of the immediately following proposition, which asserts that the α -connected subsequences of x have a maximum.

Definition 3.5 Let $x \in \Sigma^*$, $\alpha \subseteq Loc$. The α -view of x , denoted $x \downarrow \alpha$ is defined to be the \ll -maximum α -connected subsequence of x .

(In the following we use the notation $x \downarrow i$ to denote $x \downarrow \{i\}$.)

Proposition 3.6 For all $\alpha \subseteq Loc$ and for all $x \in \Sigma^*$, $C_\alpha(x)$ has a \ll -maximum.

Proof: (By induction on the length of x). For $x = \epsilon$, $C_\alpha(x) = \{\epsilon\}$, for all $\alpha \subseteq Loc$. Then the maximum element is ϵ .

To prove the induction step, assume that for all strings x of length k , for all $\alpha \subseteq Loc$, $C_\alpha(x)$ has a \ll -maximum for all α . Now, let $y = xa$ and consider any $\alpha \subseteq Loc$. By induction hypothesis, there is a \ll -maximum z in $C_\alpha(x)$. We want to show that $C_\alpha(xa)$ also has a \ll -maximum.

We consider the following two cases.

1. **Case $loc(a) \cap \alpha = \emptyset$.** For this case, we show that $C_\alpha(x) = C_\alpha(xa)$. Then, z is the \ll -maximum in $C_\alpha(xa)$.

By the observation above, if $u \ll v$, then $C_\alpha(u) \subseteq C_\alpha(v)$. Hence, $C_\alpha(x) \subseteq C_\alpha(xa)$.

On the other hand, if $u \in C_\alpha(xa)$ then $u \ll x$, because any string va with $v \ll x$ is not α -connected under this case. Hence we also have $C_\alpha(xa) \subseteq C_\alpha(x)$.

2. **Case $loc(a) \cap \alpha \neq \emptyset$.** Let w stand for the \ll -maximum subsequence in $C_{\alpha \cup loc(a)}(x)$. Note that this exists by induction hypothesis. By definition, w is $(\alpha \cup loc(a))$ -connected.

Claim: Let $loc(a) \cap \alpha \neq \emptyset$. Then va is α -connected iff v is $(\alpha \cup loc(a))$ -connected.

Assume the claim. Then wa is α -connected. In order to show that wa is the \ll -maximum subsequence in $C_\alpha(xa)$, let $u \in C_\alpha(xa)$. Hence $u \ll xa$ which means either $u \ll x$ or $(u = va \text{ and } v \ll x)$. We need to show that in either case, $u \ll wa$.

Case 1: $u \ll x$. Since u is α -connected and $loc(a) \cap \alpha \neq \emptyset$, u is $\alpha \cup loc(a)$ -connected. Since w is the \ll -maximum in $C_{\alpha \cup loc(a)}(x)$, $u \ll w$ and therefore $u \ll wa$.

Case 2: $u = va$ and $v \ll x$. Since $u = va$ is α -connected, by the claim, v is $\alpha \cup loc(a)$ -connected. Also, since w is the \ll -maximum in $C_{\alpha \cup loc(a)}(x)$, $v \ll w$, and therefore, $u = va \ll wa$. Thus we have shown that wa is the \ll -maximum subsequence in $C_\alpha(xa)$ and the proposition is proved, pending the proof of claim.

Proof of claim: Let $v = a_1 \dots a_k$.

(\Rightarrow). Since va is α -connected, for every a_j , $1 \leq j \leq k$, a_j has an α -chain in va . We have to show that each a_j has $\alpha \cup loc(a)$ -chain in v .

Fix j . Let the α -chain for a_j be $w = a_j a_{j_1} \dots a_{j_m}$. If $j_m < k$, then $w \ll v$ and $loc(a_{j_m}) \cap \alpha \neq \emptyset$, hence $loc(a_{j_m}) \cap (\alpha \cup loc(a)) \neq \emptyset$.

If $j_m = k$, then $a_j a_{j_1} \dots a_{j_{m-1}} \ll v$ and $loc(a_{j_{m-1}}) \cap loc(a) \neq \emptyset$. Then $loc(a_{j_{m-1}}) \cap (\alpha \cup loc(a)) \neq \emptyset$.

In either case a_j has a $\alpha \cup loc(a)$ -chain in v . Since j was arbitrary, therefore, v is $\alpha \cup loc(a)$ -connected.

(\Leftarrow) Given that v is $\alpha \cup loc(a)$ -connected, we have to show that every a_j has an α -chain in va .

For all a_j , $1 \leq j \leq k$, a_j has an $\alpha \cup loc(a)$ -chain, say w , in v . Then, from definition of α -chain, w is either an α -chain of a_j in v or w is a $loc(a)$ -chain of a_j in v .

In the first case, w is also an α -chain of a_j in va . In the second case, since $loc(a) \cap \alpha \neq \emptyset$, wa is an α -chain of a_j in va . Hence va is α -connected, proving the claim and the proposition. \blacksquare

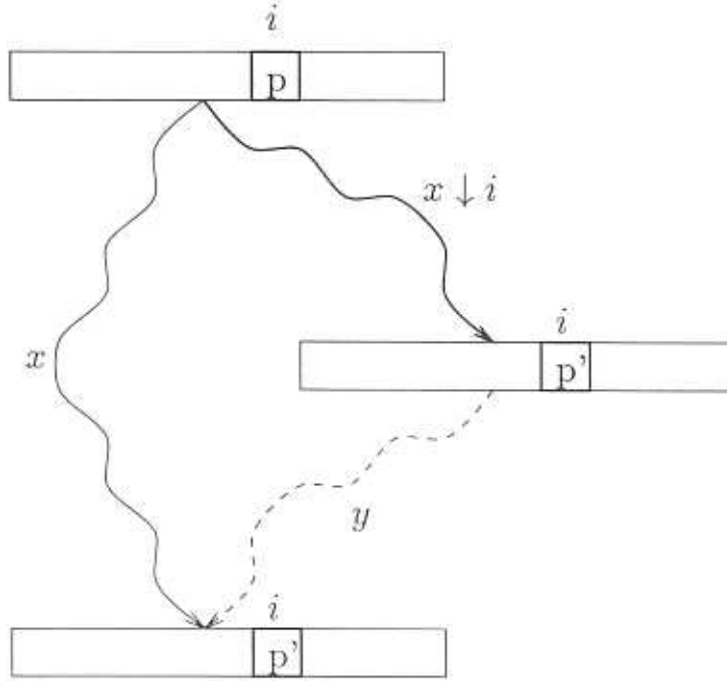
In the proof of above proposition, we observe two crucial properties of views which we use repeatedly later in the chapter:

1. When $loc(a) \cap \alpha = \emptyset$, the \ll -maximum α -connected subsequence of x is same as that of xa . In other words, $x \downarrow \alpha = xa \downarrow \alpha$. In particular, for $i \notin loc(a)$, $x \downarrow i = xa \downarrow i$.
2. When $loc(a) \cap \alpha \neq \emptyset$, if z is the \ll -maximum α -connected subsequence of x then za is the \ll -maximum α -connected subsequence of xa . In other words, $xa \downarrow \alpha = (x \downarrow (\alpha \cup loc(a))) \cdot a$. In particular, for $i \in loc(a)$, $xa \downarrow i = (x \downarrow loc(a)) \cdot a$.

3.3.1 Zielonka automaton and views

The connection between a given deterministic Zielonka automaton \mathcal{A} and i -views of a string is based on a simple idea. We know that if $x \sim y$ then $(x)_{\mathcal{A}} = (y)_{\mathcal{A}}$ i.e., both the strings lead to the same state in \mathcal{A} . We also observe that by commutation of independent actions in the string, we can factor any string x into an \sim -equivalent zy such that z is the i -view of x and i does not take part in y . Then it is the case that the i -state of a global state in \mathcal{A} reached via x depends only on the i -view of x (namely, $x \downarrow i$). For a pictorial idea, see Fig. 3.4. In the following, we make this idea precise.

Figure 3.4: Relating Zielonka automaton and i -views.



Proposition 3.7 For all $x \in \Sigma^*$, $i \in Loc$, $(x)_{\mathcal{A}}[i] = (x \downarrow i)_{\mathcal{A}}[i]$.

Proof: The following claim pins down the factoring of x into i -view and i -independent part.

Claim: For all $x \in \Sigma^*$, for all $\alpha \subseteq Loc$, there is a $y \in \Sigma^*$ such that $x \sim (x \downarrow \alpha)y$ and $y[\alpha] = \epsilon$.

Assume the claim and fix an $i \in Loc$. Then there is an $y \in \Sigma^*$ such that $x \sim (x \downarrow i)y$ and $y[i] = \epsilon$. Hence, $(x)_{\mathcal{A}} = ((x \downarrow i)y)_{\mathcal{A}}$, from which we get:

$$(x)_{\mathcal{A}}[i] = ((x \downarrow i)y)_{\mathcal{A}}[i] = (x \downarrow i)_{\mathcal{A}}[i].$$

Proof: (of claim) Fix $\alpha \subseteq Loc$. Let $x = x_0 a_1 x_1 a_2 x_2 \dots a_n x_n$ such that $x \downarrow \alpha = a_1 a_2 \dots a_n$, $x_j \in \Sigma^*$, $a_j \in \Sigma$ for all j , $0 \leq j \leq n$. Take $y = x_0 x_1 \dots x_n$.

Notice that for any $\alpha \subseteq Loc$, $x[\alpha]$ is α -connected. Hence, $x[\alpha] \ll x \downarrow \alpha$. This immediately shows, from the construction of y , that $y[\alpha] = \epsilon$.

Now we show that for all i , $0 \leq i < n$, $\text{loc}(x_i) \cap \text{loc}(a_{i+1} \dots a_n) = \emptyset$. Suppose not. Then there is an i , and b in x_i such that $\text{loc}(b) \cap \text{loc}(a_j) \neq \emptyset$ for some j , $i+1 \leq j \leq n$. But a_j has an α -chain, call it σ , in x . Hence, $b \cdot \sigma$ is an α -chain of b in x . This implies b is α -connected in x and hence $b = a_k$ for some k . But by the construction of y , again, this is a contradiction. Thus, essentially we get that for all i, j such that $0 \leq i < n, i < j \leq n, x_i \not\mathcal{I} a_{i+1}$.

By using this fact and by definition of \sim , we know that

$$\begin{aligned}
x &= x_0 a_1 x_1 a_2 x_2 \dots a_n x_n \\
&\sim a_1 x_0 x_1 a_2 x_2 \dots a_n x_n && \text{commuting } x_0 \text{ and } a_1 \\
&\sim a_1 a_2 x_0 x_1 x_2 \dots a_n x_n && \text{commuting } x_0 x_1 \text{ and } a_2 \\
&\sim \vdots \\
&\sim a_1 a_2 \dots a_n x_0 x_1 \dots x_n && \text{commuting } x_0 \dots x_{n-1} \text{ and } a_n \\
&= (x \downarrow \alpha) y.
\end{aligned}$$

This proves the claim and the proposition. ■

3.4 View-based systems for regular consistent languages

We have seen that the languages accepted by view-based systems are indeed regular and consistent. Here we prove the converse, i.e., $RCL_{\tilde{\Sigma}} \subseteq \mathcal{L}(VS)$. Since the class of deterministic Zielonka automata over $\tilde{\Sigma}$ characterize $RCL_{\tilde{\Sigma}}$, it suffices to prove the following theorem.

Theorem 3.8 *Let \mathcal{A} be a deterministic Zielonka automaton. Then there exists a view-based system \tilde{M} such that $L(\mathcal{A}) = L(\tilde{M})$.*

Proof: For all $x \in \Sigma^*$, $a \in \Sigma$, define the event associated with x as the last transition on the path for x in \mathcal{A} .

$$\text{event}(x) = \begin{cases} ((\epsilon)_{\mathcal{A}}, \epsilon, (\epsilon)_{\mathcal{A}}) & \text{if } x = \epsilon, \\ ((y)_{\mathcal{A}}, a, (x)_{\mathcal{A}}) & \text{if } x = ya. \end{cases}$$

(Note that $(x)_{\mathcal{A}}$ stands for the global state $\delta(s^0, x)$ in the deterministic Zielonka automaton \mathcal{A} .) We define by $\gamma(x) \stackrel{\text{def}}{=} (\text{event}(x \downarrow 1), \dots, \text{event}(x \downarrow n))$.

Now we construct the VS $\widetilde{M} = (M_1, \dots, M_n, F)$ where, for all $i \in Loc$, $M_i = (Q_i, \rightarrow_i, q_i^0)$ such that

- $Q_i = \{\gamma(x \downarrow i) \mid x \in \Sigma^*\}$.
- $q_i^0 = \gamma(\epsilon)$.
- $p \xrightarrow{a}_i q$ iff there is some $u \in \Sigma^*$ such that $p = \gamma(u \downarrow i)$ and $q = \gamma(ua \downarrow i)$, and
- $F \stackrel{\text{def}}{=} \{(\gamma(x \downarrow 1), \dots, \gamma(x \downarrow n)) \mid x \in L(\mathcal{A})\}$.

We show that $L(\mathcal{A}) = L(\widetilde{M})$.

(\subseteq ;) We first observe that for every $x \in \Sigma^*$, there exists a path in \widetilde{M} such that $(q_1^0, \dots, q_n^0) \xrightarrow{x} (\gamma(x \downarrow 1), \dots, \gamma(x \downarrow n))$ and then the inclusion follows at once. This can be easily proved by induction on $|x|$. The crucial point to observe is that \widetilde{M} never gets “stuck” on x if \mathcal{A} doesn't.

When $x = \epsilon$, $(\gamma(x \downarrow 1), \dots, \gamma(x \downarrow n)) = (\gamma(\epsilon), \dots, \gamma(\epsilon)) = (q_1^0, \dots, q_n^0)$. Hence the base case is trivial. For the induction step, it suffices to show that $(\gamma(x \downarrow 1), \dots, \gamma(x \downarrow n)) \xrightarrow{a} (\gamma(xa \downarrow 1), \dots, \gamma(xa \downarrow n))$ is in the product. We check that the conditions of asynchrony and perfect exchange holds for this transition implying thereby the transition is in the product.

1. For all $i \notin loc(a)$, $xa \downarrow i = x \downarrow i$, hence $\gamma(xa \downarrow i) = \gamma(x \downarrow i)$,
2. For all $i \in loc(a)$, $\gamma(x \downarrow i) \xrightarrow{a}_i \gamma(xa \downarrow i)$ by definition of local transitions, and
3. Lastly, for all $i \in loc(a)$, $\gamma(xa \downarrow i) = event(xa \downarrow i) = event(x \downarrow loc(a) \cdot a)$. Hence, for all $i, j \in loc(a)$, $\gamma(xa \downarrow i) = \gamma(xa \downarrow j)$.

(\supseteq ;) To prove this inclusion we first make the following claim.

Claim: Let $(q_1^0, \dots, q_n^0) \xrightarrow{x} (\gamma(y_1 \downarrow 1), \dots, \gamma(y_n \downarrow n))$. Then for every $i \in Loc$, $(x)_{\mathcal{A}}[i] = (y_i)_{\mathcal{A}}[i]$.

Assume the claim. Let $x \in L(\widetilde{M})$. Then there is a path $(q_1^0, \dots, q_n^0) \xrightarrow{x} \bar{q}$ in the product \widetilde{M} where $\bar{q} = (\gamma(z \downarrow 1), \dots, \gamma(z \downarrow n))$, for some $z \in L(\mathcal{A})$. By the claim,

$\forall i \in Loc, (x)_{\mathcal{A}}[i] = (z)_{\mathcal{A}}[i]$ meaning thereby $(x)_{\mathcal{A}} = (z)_{\mathcal{A}}$. Since $z \in L(\mathcal{A})$, $x \in L(\mathcal{A})$ and we get the required inclusion.

Proof of claim: The proof is by induction on the length of x . The base case is trivial. For the induction step, assume the hypothesis for all $x \in \Sigma^*$ such that $|x| = k$. Let $y = xa$ and let the path in \widehat{M} for $y = xa$ be

$$(q_1^0, \dots, q_n^0) \xrightarrow{x} (\gamma(y_1 \downarrow 1), \dots, \gamma(y_n \downarrow n)) \xrightarrow{a} (\gamma(z_1 \downarrow 1), \dots, \gamma(z_n \downarrow n)).$$

By induction hypothesis,

$$\text{for all } i \in Loc, (x)_{\mathcal{A}}[i] = (y_i)_{\mathcal{A}}[i]. \quad (1)$$

In the induction step, We want to show that

$$\text{for all } i \in Loc, (xa)_{\mathcal{A}}[i] = (z_i)_{\mathcal{A}}[i]. \quad (2)$$

Since in the following, we use Proposition 3.7 (relating views and states of Zielonka automaton) many times, we recall it for ready reference.

Proposition 3.7: For all $x \in \Sigma^*$, $i \in Loc$, $(x)_{\mathcal{A}}[i] = (x \downarrow i)_{\mathcal{A}}[i]$.

Fix an $i \in Loc$. If $i \notin loc(a)$, $\gamma(y_i \downarrow i) = \gamma(z_i \downarrow i)$. By definition of γ and *event*, we get $(y_i \downarrow i)_{\mathcal{A}} = (z_i \downarrow i)_{\mathcal{A}}$. So, in particular for i , $(y_i \downarrow i)_{\mathcal{A}}[i] = (z_i \downarrow i)_{\mathcal{A}}[i]$. Now, using Proposition 3.7, we get

$$(y_i)_{\mathcal{A}}[i] = (z_i)_{\mathcal{A}}[i]. \quad (3)$$

By the induction hypothesis (1), then, we get $(x)_{\mathcal{A}}[i] = (z_i)_{\mathcal{A}}[i]$. But $(x)_{\mathcal{A}}[i] = (xa)_{\mathcal{A}}[i]$, since $i \notin loc(a)$. So finally, we get $(xa)_{\mathcal{A}}[i] = (z_i)_{\mathcal{A}}[i]$ and we are done.

For $i \in loc(a)$, $\gamma(y_i \downarrow i) \xrightarrow{a} \gamma(z_i \downarrow i)$. So by the construction of \rightarrow_i , for all $i \in loc(a)$, $\exists u_i \in \Sigma^*$ such that

1. $\gamma(y_i \downarrow i) = \gamma(u_i \downarrow i)$, and
2. $\gamma(u_i a \downarrow i) = \gamma(z_i \downarrow i)$.

Now, from the first conjunct, using Proposition 3.7, we get for all $i \in \text{loc}(a)$, $(y_i)_{\mathcal{A}} [i] = (u_i)_{\mathcal{A}} [i]$. Combining with induction hypothesis (1), we have

$$\text{for all } i \in \text{loc}(a), (x)_{\mathcal{A}} [i] = (u_i)_{\mathcal{A}} [i]. \quad (4)$$

Similarly, from the second conjunct, we get

$$\text{for all } i \in \text{loc}(a), (z_i)_{\mathcal{A}} [i] = (u_i a)_{\mathcal{A}} [i]. \quad (5)$$

Since the transition is a perfect exchange, for all $j, k \in \text{loc}(a)$, $\gamma(z_j \downarrow j) = \gamma(z_k \downarrow k)$, hence, for all $j, k \in \text{loc}(a)$, $\gamma(u_j a) = \gamma(u_k a)$. From the definition of γ and *event*, for all $j, k \in \text{loc}(a)$, $(u_j \downarrow \text{loc}(a))_{\mathcal{A}} = (u_k \downarrow \text{loc}(a))_{\mathcal{A}}$, and hence, in particular for k ,

$$\text{for all } j, k \in \text{loc}(a), (u_j)_{\mathcal{A}} [k] = (u_k)_{\mathcal{A}} [k]. \quad (6)$$

Now, by equations 4 and 6 above, for all $k, i \in \text{loc}(a)$, $(x)_{\mathcal{A}} [k] = (u_k)_{\mathcal{A}} [k] = (u_i)_{\mathcal{A}} [k]$. Then, by the property of Zielonka automaton, for all $k \in \text{loc}(a)$, $(xa)_{\mathcal{A}} [k] = (u_i a)_{\mathcal{A}} [k]$. In particular, when $k = i$, $(xa)_{\mathcal{A}} [i] = (u_i a)_{\mathcal{A}} [i]$. Using equation 5 then we get $(xa)_{\mathcal{A}} [i] = (z_i)_{\mathcal{A}} [i]$. This proves the claim and the theorem. ■

3.5 Discussion

We have shown how the distribution of regular consistent behaviours can be done in a simple way in view-based systems. But since we have required that the local states capture partial views of the global computation, these systems are essentially semantic in nature. Because of this, it is difficult to code up programming intuition in these models and that is reflected in the difficulty in getting a corresponding nice syntax which could give us a Kleene's theorem. This is one of the motivations for a syntactic approach which we discuss in the next chapter.

4 ■ Assumption and commitment in automata

4.1 Perspective

Compositionality is a desired criterion for verification methodologies, particularly for development and analysis of large systems. The idea is to decompose a system into smaller subsystems and 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.

4.1.1 Parallel programs and compositional reasoning

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 $P_1 \parallel \dots \parallel P_n$ 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 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 it 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].

4.1.2 Local reasoning

In this thesis, 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. One must note that this is also a notion of compositionality in the broader sense of the term. After all, one has to compose subsystems and local reasoning should lead to properties of global (compatible) behaviour.

Note 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.

4.1.3 AC-framework and automata theory

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.

4.2 Local reasoning in finite-state process models

Our scope of study is process models of finite state distributed systems. When one wants to do compositional reasoning for these systems in an AC-framework, one specifies a process P as $\langle A \rangle P \langle C \rangle$ where A is the assumption on the environment and C is the commitment of process P . For example, in [MC], every process is specified by a triple $r|h|s$ where r and s are predicates over communication histories and h is the process. This triple is interpreted as follows: s is true initially in h and if r holds at all times prior to a communication then s holds at all times prior to and following that communication. If we represent the systems as transition systems, the assumption-commitment requirement asserts a system invariant: let $p \xrightarrow{a} q$ be any transition, then if A holds at p then C holds at q .

On the other hand, 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. We discuss this in slightly more detail now. In the next chapter, we will formalize the intuition and the necessary concepts.

In order to express the assumptions and commitments of 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 $\mathcal{C} = ((\mathcal{C}_1, \preceq_1), \dots, (\mathcal{C}_n, \preceq_n))$. 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.

In the rest of the chapter, we discuss how one can incorporate assumption and commitment in automata and model interesting distributed protocols.

4.3 Assumption-commitment on transitions

A theoretically simpler 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 .)

4.3.1 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.

The commitment alphabet is $\mathcal{C} = \langle C_1 = (\{p1, np1\}, \preceq_1), C_2 = (\{p2, np2\}, \preceq_2) \rangle$. 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. 4.1.

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 4.1 shows the two processes and also the product showing the global behaviour. See that the assumptions at the local transition $W_1 \xrightarrow{c} C_1$ are (p_1, np_2) and those at $W_2 \xrightarrow{c} C_2$ 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 $(W_1, W_2) \xRightarrow{c} (C_1, C_2)$ is not possible; so that at no point both the processes can be in the critical section, thus satisfying the safety requirement.

4.3.2 An example class of systems

As suggested above, alongwith a distributed alphabet $\tilde{\Sigma}$, we have an alphabet which we call the **commitment alphabet**. This is a tuple $\mathcal{C} = ((C_1, \preceq_1), \dots, (C_n, \preceq_n))$ where

- for all $i \in Loc$, C_i is a non-empty set, and
- for all $i \neq j$, $C_i \cap C_j = \{\perp\}$.

The element \perp is the null assumption (or commitment). We call $\mathcal{C} = C_1 \cup \dots \cup C_n$ the *commit set*. For $a \in \Sigma$, we use the notation \mathcal{C}_a to denote the set $\bigcup_{i \in loc(a)} C_i$.

We want to annotate each transition on a with assumptions about participating agents. Each such annotation gives an element of C_i for each agent i participating in a . Hence, for each $a \in \Sigma$, the annotations are from the set of functions

$$\Phi_a \stackrel{\text{def}}{=} \{\phi : loc(a) \mapsto \mathcal{C}_a \mid \forall i \in loc(a), \phi(i) \in C_i\}.$$

We now extend the distributed alphabet so that we have actions of the form $\langle a, \phi \rangle$ where $\phi \in \Phi_a$. When $\phi(i) = \perp$ for $i \in loc(a)$, we treat this as a “don’t care” condition.

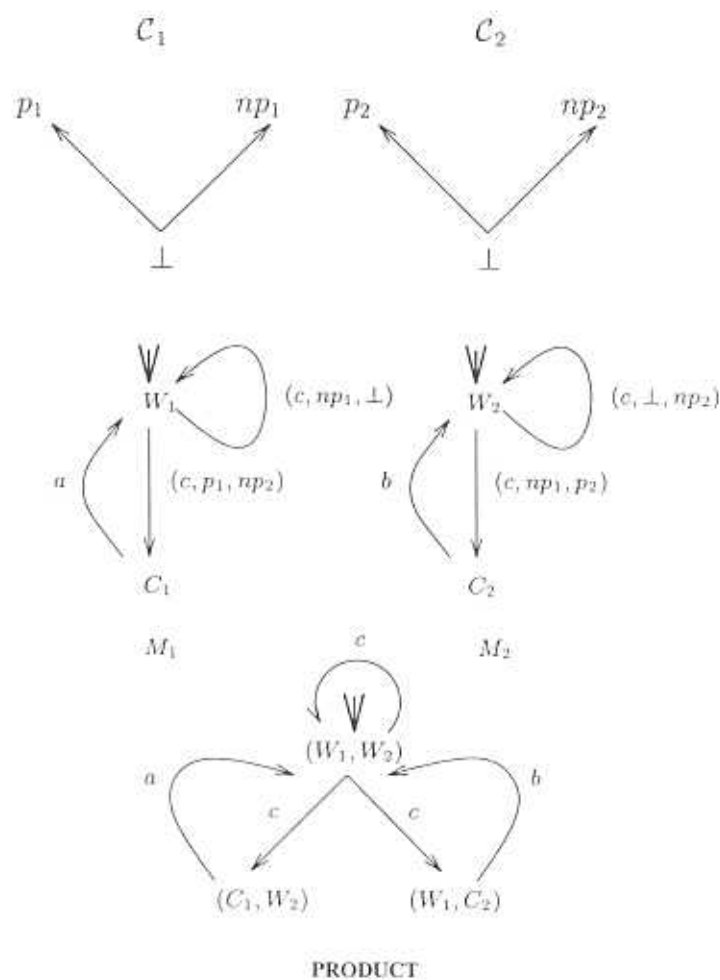


Figure 4.1: Two-processor mutual exclusion.

Since we consider only finite state systems, we use only finitely many annotations for local transitions. This implies that it is sufficient to have finite commitment alphabets \mathcal{C}_i .

Definition 4.1 *Given a distributed alphabet, $\tilde{\Sigma} = (\Sigma_1, \dots, \Sigma_n)$ and a commit alphabet $\tilde{\mathcal{C}}$, we define extended alphabets as follows:*

$$\begin{aligned}\Sigma_i^c &\stackrel{\text{def}}{=} \{ \langle a, \phi \rangle \mid a \in \Sigma_i \text{ and } \phi \in \Phi_a \}. \\ \tilde{\Sigma}^c &\stackrel{\text{def}}{=} \langle \Sigma_1^c, \dots, \Sigma_n^c \rangle, \quad \Sigma^c \stackrel{\text{def}}{=} \bigcup_{i \in \text{Loc}} \Sigma_i^c.\end{aligned}$$

We now define *AT-automata*, the class of distributed systems which make assumptions and commitments on synchronized transitions. We first define individual automata and then systems of such automata.

Definition 4.2 *Consider the distributed alphabet $\tilde{\Sigma} = (\Sigma_1, \dots, \Sigma_n)$, the commit alphabet $\tilde{\mathcal{C}}$ and the associated extended alphabet $\tilde{\Sigma}^c$ over the set of locations Loc . Let $i \in \{1, 2, \dots, n\}$.*

1. An **AT-automaton** over Σ_i^c is a tuple (M_i, f_i) where
 - $M_i = (Q_i, \longrightarrow_i, q_i^0)$ is a TS, and
 - $f_i : \longrightarrow_i \mapsto \Phi$ is such that for any transition $\tau = p \xrightarrow{a} q$, $f_i(\tau) \in \Phi_a$.
2. A **System of AT-automata (ATS)** over the extended alphabet $\tilde{\Sigma}^c$ is a tuple

$$\tilde{M} = \langle M_1, \dots, M_n, \langle f_1, \dots, f_n \rangle, F \rangle,$$

where each (M_i, f_i) is an AT-automaton over Σ_i^c , and $F \subseteq (Q_1 \times \dots \times Q_n)$.

The global behaviour of \tilde{M} is given below as that of the product automaton \hat{M} associated with the system. Note that the system is then a finite state machine over Σ , thus hiding away assumptions and commitments as internal details. This fits with the intuition that the behaviour of a distributed system is globally specified on Σ , and the machines in the system are programmed to achieve this, using internal coordination mechanisms like synchronization and commitments among themselves. In the case of ATS's this is achieved by putting an extra compatibility condition on the transition of the product TS.

Definition 4.3 A global transition $(p_1, p_2, \dots, p_n) \xrightarrow{a} (q_1, q_2, \dots, q_n)$ is said to be **compatible** if

1. for all $j \in \text{loc}(a)$, there exists $\tau_j = p_j \xrightarrow{a} q_j$ such that $f_j(\tau_j) = \phi_j \in \Phi_a$, and
2. for all $i, k \in \text{loc}(a)$, $\phi_i(k) \preceq_k \phi_k(k)$.

Definition 4.4 Given a system of AT-automata over $\widetilde{\Sigma}^c$

$$\widetilde{M} = (M_1, M_2, \dots, M_n, \langle f_1, \dots, f_n \rangle, F)$$

the product automaton associated with the system is given by (\widehat{M}, F) , where

- $\widehat{M} = (\widehat{Q}, \longrightarrow, (q_1^0, \dots, q_n^0))$ is the complete product TS of \widetilde{M} , and
- each transition in \longrightarrow is compatible.

The class of languages over Σ accepted by systems of AT-automata is denoted as $\mathcal{L}(ATS)_{\widetilde{\Sigma}}$. Formally,

$$\mathcal{L}(ATS)_{\widetilde{\Sigma}} = \{L \subseteq \Sigma^* \mid \exists \widetilde{C} \text{ and an AT system } \widetilde{M} \text{ over } \widetilde{\Sigma}^c \text{ such that } L = L(\widehat{M}, F)\}.$$

The class of languages accepted by such systems have been shown to be the same as regular consistent languages in [MR2]. Thus these systems have the same expressive power as view-based systems of Chapter 3. However, we omit the proof here as this will follow from a later result in Chapter 6.

Theorem 4.5 $\mathcal{L}(ATS)_{\widetilde{\Sigma}} = \mathcal{L}(RCL_{\widetilde{\Sigma}})$.

We now present a simple example of these automata. Consider the familiar language $L = ([ab]c + [aabb]c)^*$ over the distributed alphabet $\widetilde{\Sigma} = \Sigma_1 = \{a, c\}, \Sigma_2 = \{b, c\}$. In Chapter 2, we noted that this can not be captured by product systems as they would accept strings like $aabc$ which are not in L . In Chapter 3, we gave a view-based system that accepts L . Here we give an AT-system that accepts this language.

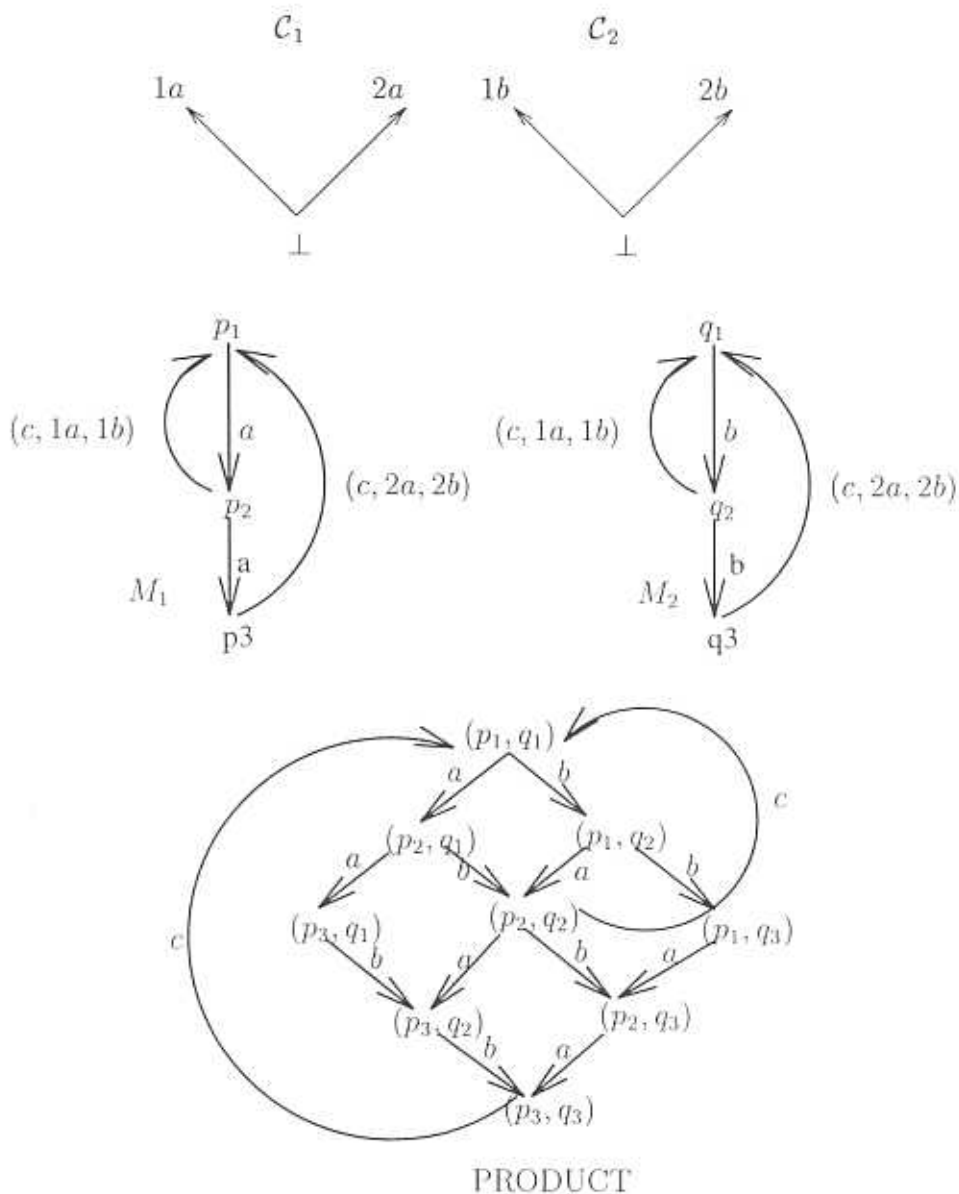


Figure 4.2: AT-system for the language $([ab]c + [aabb]c)^*$. $\Sigma_1 = \{a, c\}$, $\Sigma_2 = \{b, c\}$

Refer to Fig. 4.2. Here the commit alphabet is $\mathcal{C} = \langle \mathcal{C}_1 = (\{1a, 2a\}, \preceq_1), \mathcal{C}_2 = (\{1b, 2b\}, \preceq_2) \rangle$; the orders $\preceq_i, i = 1, 2$ are as shown in the figure. $1a$ records that one a has been seen and $2a$ records that two a 's have been seen. Now, in P_1 , transition $\tau = p_2 \xrightarrow{c} p_1$ with $f_1(\tau) = (1a, 1b)$ says that the synchronization transition will be compatible only when a c -transition in P_2 commits to having recorded $1b$. Such reasoning gives us that only $(p_2, q_2) \xrightarrow{c} (p_1, q_1)$ and $(p_3, q_3) \xrightarrow{c} (p_1, q_1)$ are compatible global transitions. Consider now a string $aabc$. For this to be accepted by the system, the transition $(p_3, q_2) \xrightarrow{c} (p_1, q_1)$ should be in the product. But $f_1(p_3 \xrightarrow{c} p_1) = (2a, 2b)$ and $f_2(q_2 \xrightarrow{c} q_1) = (1a, 1b)$ and $2b \not\preceq_2 1b$ and $1a \not\preceq_1 2a$. Hence the transition is not compatible. Thus all the "bad" strings are filtered out by the compatibility condition in the product and we get the desired behaviour.

4.4 Assumption-commitment on states

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).

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

4.4.1 Bit transmission problem

In Section 4.1.2, we introduced the bit transmission system where 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.

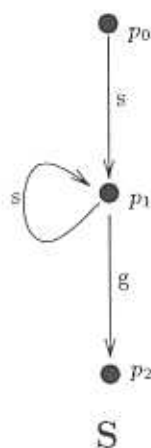


Figure 4.3: The sender.

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. 4.3). The alphabet Σ_1 of sender S is $\{s, g\}$, where

- s : S sends the bit to R , and
- 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

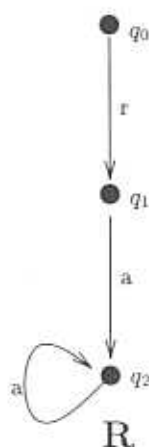


Figure 4.4: The receiver.

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. 4.4) 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 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).

Commitment alphabet

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 are 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 $\{bit_recd, ack_recd\}$. 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.)

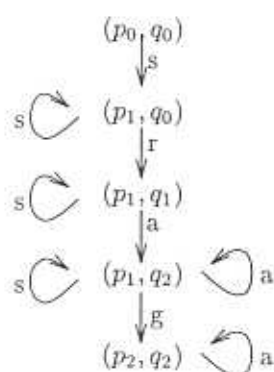
		Commitment	Assumption
(S)	p_0	$\neg bit_sent$ and $\neg ack_recd$ by S	$\neg bit_recd$ by R
	p_1	bit_sent and $\neg ack_recd$ by S	R may be in any state.
	p_2	bit_sent and ack_recd by S	bit_recd and ack_sent by R

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. 4.4.1 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 p_0 is $\mu = \neg bit_sent \wedge \neg ack_recd$ but $\mu \not\preceq \lambda$. Similar compatibility considerations rule out global states (p_2, q_0) , (p_2, q_1) and (p_0, q_2) .



Product

Figure 4.5: ACS for bit transmission

Then, we see from the product automaton that with the final state $F = (p_2, q_2)$, the behaviour of the ACS is $s^+ r s^+ a (s + a)^* g a^*$, 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.

4.4.2 Sequence transmission problem

In this section we 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. 4.6 with the assumption and commitments from Fig. 4.7. (Note that there are now infinite number of elements of the commitment alphabet; in the table a state p_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.

At a first attempt, if we banish the labels altogether, we get a protocol as in Fig. 4.8. But 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

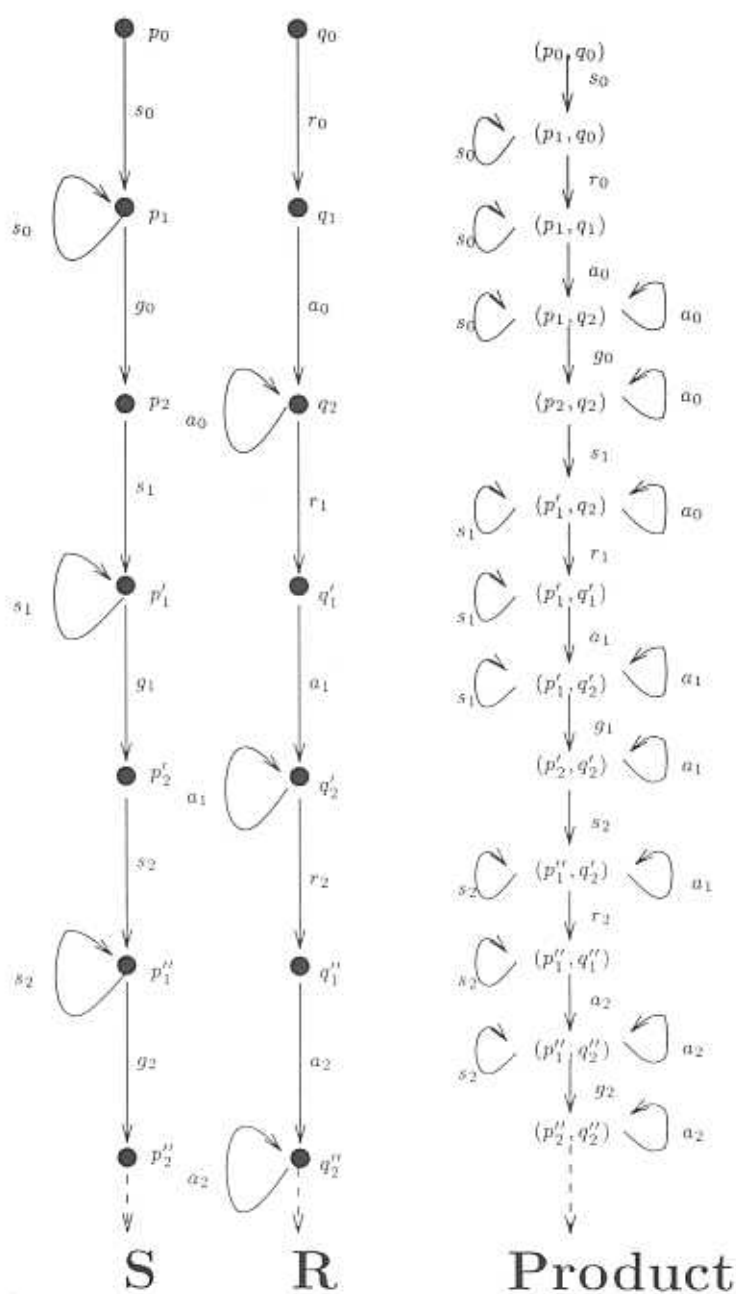


Figure 4.6: An infinite state protocol for STP.

	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

Figure 4.7: Infinitely many assumptions and commitments for the infinite state protocol.

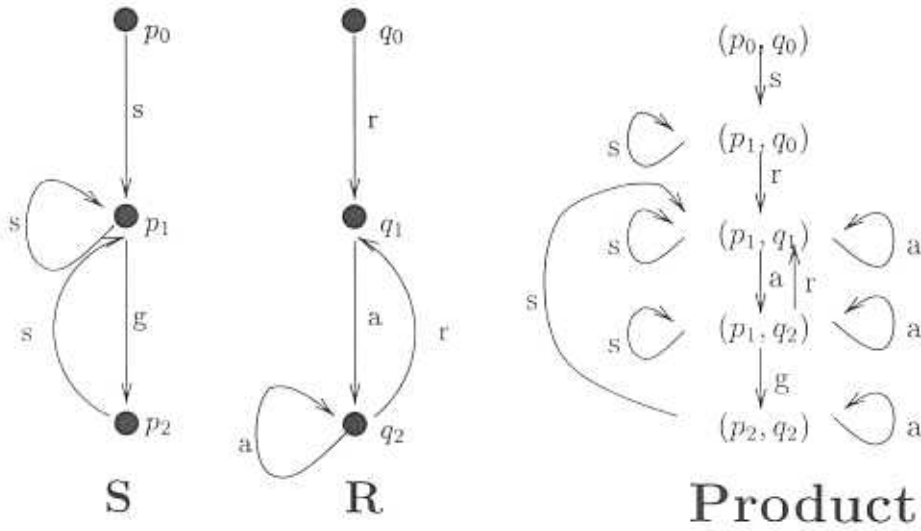


Figure 4.8: An incorrect folding of the infinite state protocol for STP.

of the i -th message. Hence it then starts sending the $(i+1)$ -th message. Thus 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 the assumption and commitments do not reflect this.

In order to remedy this deficiency, we have two-bits 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. 4.10 with the assumptions and commitments as in the table

	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	bit_0_recd and ack_0_sent by R
	$\neg bit_1_sent$ and $\neg ack_1_recd$ by S	$\neg bit_1_recd$ 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	bit_1_recd and ack_1_sent by R
	$\neg bit_0_sent$ and $\neg ack_0_recd$ by S	$\neg bit_0_recd$ 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	bit_0_sent by S
	$\neg bit_1_recd$ and $\neg ack_1_sent$ 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	bit_1_sent by S
	$\neg bit_0_recd$ and $\neg ack_0_sent$ by S	$\neg bit_0_recd$ by R

Figure 4.9: Assumptions and commitments for a correct finite state protocol of Fig. 4.10.

in Fig. 4.9 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. In fact, the crucial point to note in this illustration is that the discovery of the deficiency and the subsequent remedy can be done entirely locally for each of the sender and receiver without reference to global states. This only reiterates our stand on ease of component design in a distributed systems using the assumption-commitment paradigm.

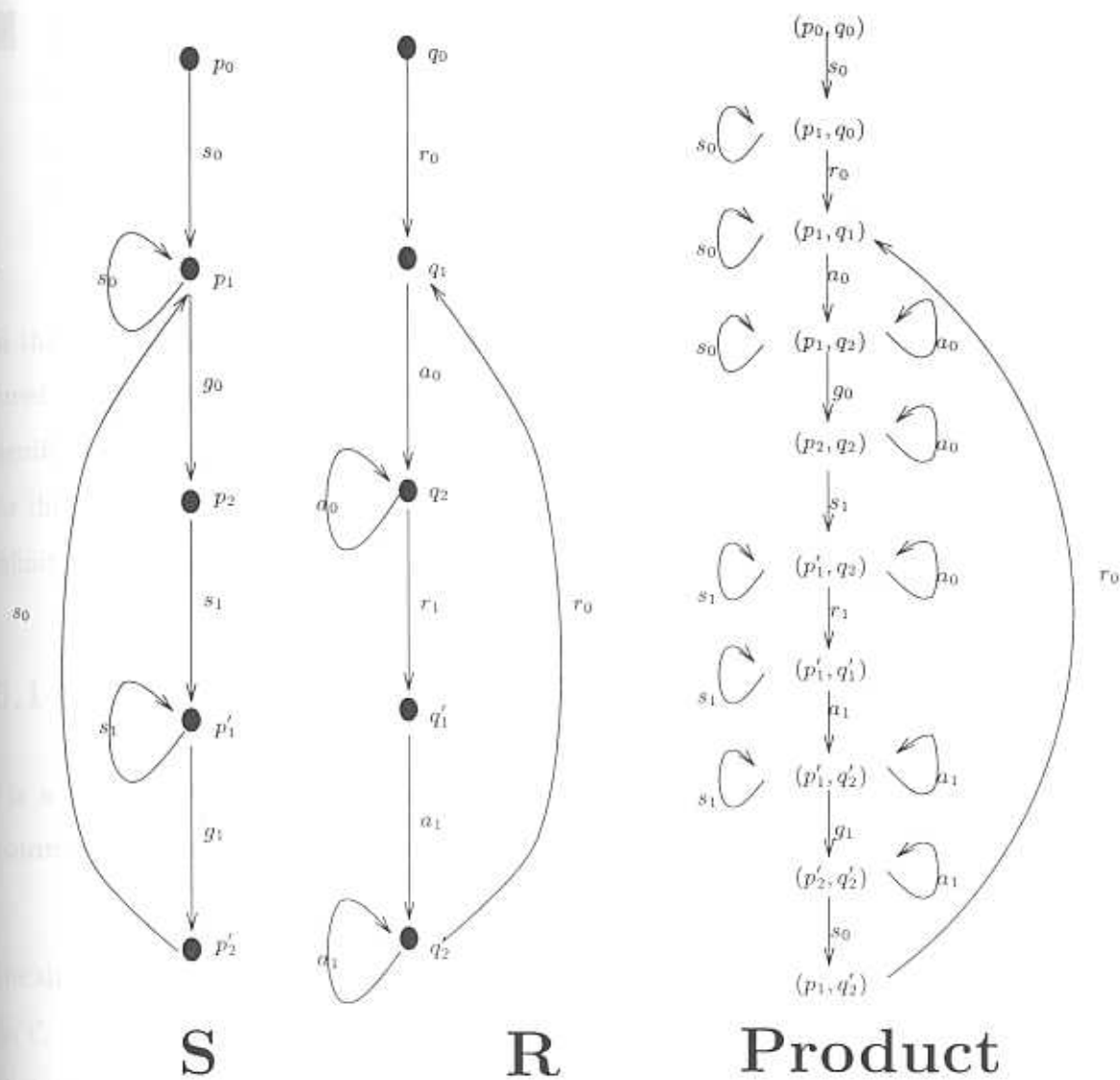


Figure 4.10: A correct folding of the infinite state protocol for STP.

5 ■ Assumption-compatible systems

In this chapter we show that all regular behaviours on a distributed alphabet can be captured by Assumption-Compatible systems when we consider their finite behaviour. More significantly, we give a syntax that reflects top-level parallelism and prove a Kleene theorem for this syntax. We also see that these results smoothly generalize when we consider regular infinite behaviour.

5.1 Definition

Fix a distributed alphabet $\bar{\Sigma} = (\Sigma_1, \dots, \Sigma_n)$. As suggested in Section 4.3, we also have a **commitment alphabet**

$$\bar{\mathcal{C}} = ((\mathcal{C}_1, \preceq_1), \dots, (\mathcal{C}_n, \preceq_n)).$$

Recall that 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 an ordering on \mathcal{C}_i . The element \perp is the null assumption (or commitment). We call $\mathcal{C} = \mathcal{C}_1 \cup \dots \cup \mathcal{C}_n$ the *commit set*. As we observed before, since we work with finite state systems, it suffices to consider finite \mathcal{C}_i .

The difference is that instead of assumption and commitment on transitions, we now have assumption and commitments annotating local states. These annotations come from the set

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

Note also that the orders \preceq_i are binary relations with no special properties, as per the definition. Natural assumptions and commitments may require that these have more structure. For example, if the commitment alphabet is constructed from boolean formulae (as was done in the Chapter 4) 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 drawing. But such structure on the orders are not mandatory by definition.

Definition 5.1 Let $i \in \{1, 2, \dots, n\}$. An **AC transition system**(AC-TS) over $(\Sigma_i, \tilde{\mathcal{C}})$ is a tuple (M_i, f_i) where $M_i = (Q_i, \longrightarrow_i, q_i^0)$ is a TS over Σ_i and $f_i : Q_i \rightarrow \Phi$ 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 5.2 An **Assumption-compatible system**(ACS) over $(\tilde{\Sigma}, \tilde{\mathcal{C}})$ is given by a tuple

$$\tilde{M} = (M_1, M_2, \dots, M_n, \langle f_1, f_2, \dots, f_n \rangle, F),$$

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

Global behaviour of \tilde{M} is given below as that of the product automaton \hat{M} associated with the system. Unlike in the earlier distributed systems, the product TS is not over the whole global state space but over global states where assumptions and commitments of local states are mutually compatible.

Definition 5.3 Given an ACS $\tilde{M} = (M_1, M_2, \dots, M_n, \langle f_1, f_2, \dots, f_n \rangle, F)$. A global state $(p_1, p_2, \dots, p_n) \in Q$ is called **compatible** iff

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

Definition 5.4 The product automaton of \widetilde{M} is defined to be (\widehat{M}, F) where

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

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

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

Note that $\mathcal{L}(ACS_{\widetilde{\Sigma}})$ need not be closed under \sim . As an example, we describe an ACS in Fig. 5.1 that accepts the language $(ab)^*$, where $\widetilde{\Sigma} = (\{a\}, \{b\})$.

Example Let $\widetilde{C} = (\mathcal{C}_1, \mathcal{C}_2)$ where the commit alphabet \mathcal{C}_i are as shown in Fig. 5.1. 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 $f_1(p_1)(2) = \nu_6 \not\preceq_2 \nu_2 = f_2(q_2)(2)$.

5.2 Closure properties of ACS's

In the following section, we prove that ACS's over $\widetilde{\Sigma}$ characterize all regular languages. Hence, from the point of view of language acceptance, ACS's are closed under complementation, union and concatenation. Still, it is instructive to do the direct constructions on ACS's for these operations. We illustrate this in the following.

For simplicity of notation, we restrict ourselves to the case when $Loc = \{1, 2\}$. Fix a distributed alphabet $\widetilde{\Sigma}$ and two commit alphabets \mathcal{C} and \mathcal{D} .

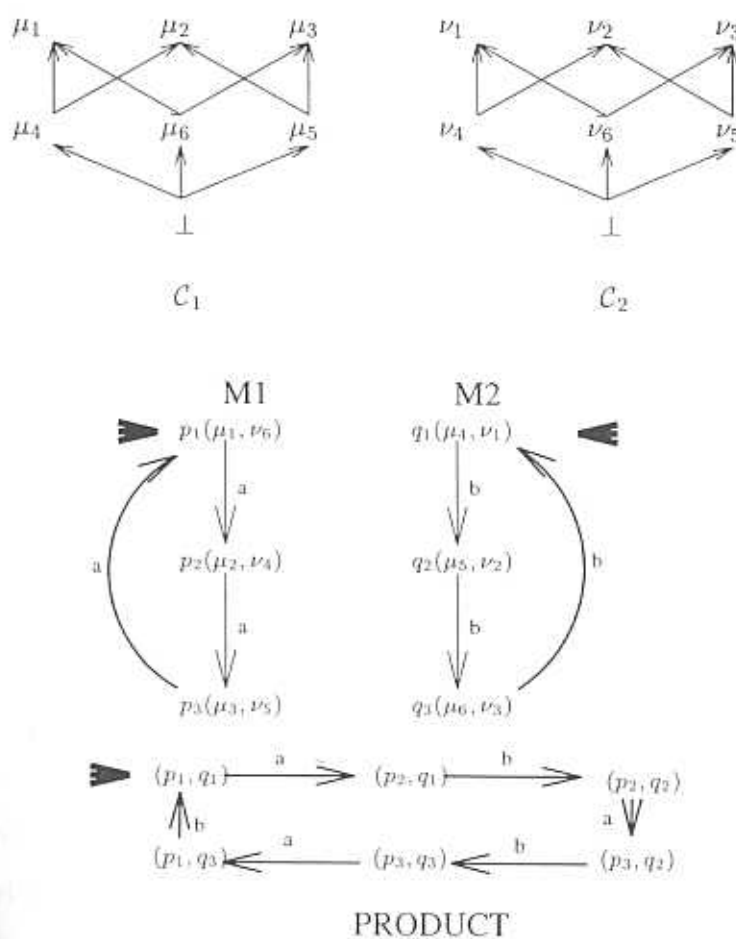


Figure 5.1: An example ACS accepting $(ab)^*$ over the alphabet $(\{a\}, \{b\})$.

Let $M = (M_1, M_2, \langle f_1, f_2 \rangle, F)$ be an ACS over $\tilde{\Sigma}$ with \mathcal{C} as the commit alphabet. Let $M_i = (P_i, \rightarrow_i, p_i^0, f_i)$ be the local AC-automata.

Similarly, let $N = (N_1, N_2, \langle g_1, g_2 \rangle, G)$ be an ACS over $\tilde{\Sigma}$ with \mathcal{D} as the commit alphabet. Let $N_i = (Q_i, \rightarrow_i, q_i^0, g_i)$ be the local AC-automata.

Assume, w.l.o.g, that the commit alphabets \mathcal{C} and \mathcal{D} are disjoint and so also are the states of M and N . We have used the same "arrows" to denote transitions of both M and N to avoid some extra notation. But the contexts disambiguate this overloading.

For convenience, let $f_1(p_1^0) = \langle \lambda_1, \lambda_2 \rangle$, $f_2(p_2^0) = \langle \nu_1, \nu_2 \rangle$, $g_1(q_1^0) = \langle \mu_1, \mu_2 \rangle$ and $g_2(q_2^0) = \langle \eta_1, \eta_2 \rangle$.

It is immediate that $M = (M_1, M_2, \langle f_1, f_2 \rangle, \overline{F})$, with $\overline{F} = \hat{Q} \setminus F$ accepts the complement of $L(M)$, so closure under complementation is proved.

5.2.1 ACS accepting union of $L(M)$ and $L(N)$.

Construct an ACS O over $\tilde{\Sigma}$ as follows.(Fig. 5.2)

Let the commit alphabet $\mathcal{E} = ((\mathcal{E}_1, \preceq_1), (\mathcal{E}_2, \preceq_2))$ where for each local commitment alphabet we introduce some new elements.

We define $\mathcal{E}_1 = \mathcal{C}_1 \cup \mathcal{D}_1 \cup \{(\lambda_1 \vee \mu_1), (\nu_1 \wedge \eta_1)\}$, where the lub and glb are new elements. Similarly we define $\mathcal{E}_2 = \mathcal{C}_2 \cup \mathcal{D}_2 \cup \{(\nu_2 \vee \eta_2), (\lambda_2 \wedge \mu_2)\}$, where the lub and glb are again new elements.

The orders in each \mathcal{E}_i is now the disjoint union of the orders of \mathcal{C}_i and \mathcal{D}_i plus some natural relations induced by the new elements. For example, Let λ be a new element as described above and let $l \in \mathcal{C}_i \cup \mathcal{D}_i$. Then, \preceq_i now includes all (λ, l) such that $\lambda = (l \wedge \mu)$ for some μ , and it also includes (l, λ) such that $\lambda = (l \vee \mu)$ for some μ .

Among the new elements, the order is defined as $(\lambda_2 \wedge \mu_2) \preceq_2 (\nu_2 \vee \eta_2)$ and $(\nu_1 \wedge \eta_1) \preceq_1 (\lambda_1 \vee \mu_1)$.

Let $O = (O_1, O_2, \langle h_1, h_2 \rangle, H)$ be an ACS over $\tilde{\Sigma}$ with \mathcal{E} as the commit alphabet. Let $O_i = (R_i, \rightarrow_i, r_i^0, h_i)$ be the local AC-automata, where

$$1. R_i = P_i \cup Q_i \cup \{r_i^0\},$$

$$\Rightarrow_i = \rightarrow_i^M \cup \rightarrow_i^N$$

$$2. \quad \cup \{ (r_i^0, a, p) \mid p_i^0 \xrightarrow{a} p \} \\ \cup \{ (r_i^0, a, q) \mid q_i^0 \xrightarrow{a} q \}.$$

$$3. \text{ For all } i \in \{1, 2\} \text{ and } r \in P_i \cup Q_i,$$

$$h_i(r) = \begin{cases} f_i(r) & \text{if } r \in P_i. \\ g_i(r) & \text{if } r \in Q_i. \end{cases}$$

In addition, $h_1(r_1^0) = \langle (\lambda_1 \vee \mu_1), (\lambda_2 \wedge \mu_2) \rangle$ and $h_2(r_2^0) = \langle (\nu_1 \wedge \eta_1), (\nu_2 \vee \eta_2) \rangle$.

$$4. H = F \cup G.$$

From the assumption maps, immediately we get that the initial state (r_1^0, r_2^0) is compatible, since $(\nu_1 \wedge \eta_1) \preceq_1 (\lambda_1 \vee \mu_1)$ and $(\lambda_2 \wedge \mu_2) \preceq_2 (\nu_2 \vee \eta_2)$.

Further, for any global state (r_1, r_2) where $p_i \neq r_i^0, i = 1, 2$, both r_i 's are in P_i or from Q_i , otherwise they will be incompatible. This is because their assumptions and commitments are now from different commitment alphabets which are disjoint by assumption.

Since the initial state does not have any in-coming transitions, any string x passes through only global states that are also global states for either \widehat{M} or \widehat{N} . This leads to the fact that P accepts the union of $L(M)$ and $L(N)$.

5.2.2 ACS's with multiple initial states

A much simpler construction results if we extend the class of ACS's to have multiple initial states. Then the introduction of the special state and the special transitions is rendered redundant: one just takes the component-wise disjoint union of the given ACS's.

An ACS with multiple initial states is given as $M = (M_1, \dots, M_n, \langle f_1, \dots, f_n \rangle, I, F)$ where the compatible product has the set of initial states I instead of a single initial state.

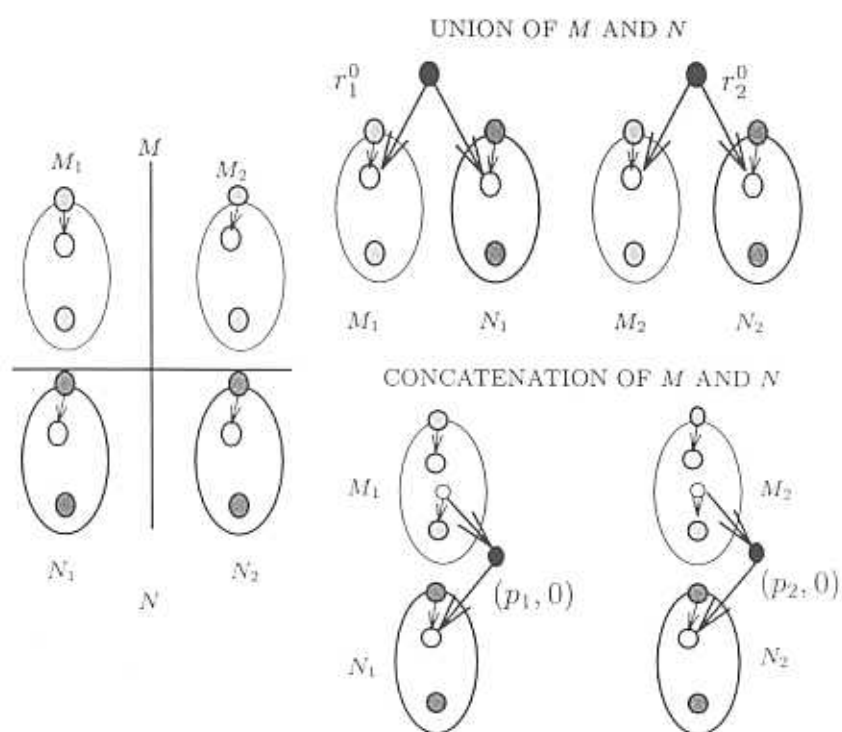


Figure 5.2: Union and concatenation of ACS's.

The above construction effectively shows that, from the point of view of language acceptance, the class of ACS's with multiple initial states is equivalent to the one having ACS's with single initial state. As in the case of Chapter 2, we also observe that ACS's with multiple final states also can be seen as union of ACS's each with single initial and single final state.

5.2.3 ACS accepting concatenation of $L(M)$ and $L(N)$.

We consider the case when M has a single final state (p_1^f, p_2^f) and N has a single initial state. Extension to multiple initial states and multiple final states is not difficult.

We construct the commitment alphabet exactly as we did in case of "union" above; except that in place of the initial state (p_1^0, p_2^0) , we consider the final state (p_1^f, p_2^f) .

Let the assumption map of the final states be as follows: $f_1(p_1^f) = \langle \lambda_1, \lambda_2 \rangle$, $f_2(p_2^f) = \langle \nu_1, \nu_2 \rangle$. Also let $g_1(q_1^0) = \langle \mu_1, \mu_2 \rangle$ and $g_2(q_2^0) = \langle \eta_1, \eta_2 \rangle$. Then, the commitment alphabet constructed is identical to that constructed above.

Let $O = (O_1, O_2, \langle h_1, h_2 \rangle, H)$ be an ACS over $\tilde{\Sigma}$ with \mathcal{E} as the commit alphabet. Let $O_i = (R_i, \Rightarrow_i, r_i^0, h_i)$ be the local AC-automata, where

1. $R_i = P_i \cup Q_i \cup \{(p_i, 0)\}$.
2. $r_i^0 = p_i^0$.
3. $\Rightarrow_i = \longrightarrow_i \cup \longrightarrow_i$
 $\cup \{(p, a, \langle p_i, 0 \rangle) \mid p \xrightarrow{a}_i p_1^f\}$
 $\cup \{((p_i, 0), a, q) \mid q_i^0 \xrightarrow{a}_i q\}$.
4. For all $i \in \{1, 2\}$ and $s \in P_i \cup Q_i$,

$$h_i(s) = \begin{cases} f_i(s) & \text{if } s \in P_i. \\ g_i(s) & \text{if } s \in Q_i. \end{cases}$$

In addition, $h_1(p_1, 0) = \langle (\lambda_1 \vee \mu_1), (\lambda_2 \wedge \mu_2) \rangle$ and $h_2(p_2, 0) = \langle (\nu_1 \wedge \eta_1), (\nu_2 \vee \eta_2) \rangle$.

5. $H = G$.

The construction of transition relation for P_i ensures that entry to N_i is only through the special states $(p_i, 0)$ and there are no transitions from N_i back to M_i .

The special state tuple $((p_1, 0), (p_2, 0))$ is compatible and hence it is in the compatible product. Crucially, it acts as a synchronizing point. This is because any state (s, t) with $s \in P_i$ and $t \in Q_j$, $i \neq j$ and $s \neq (p_i, 0), t \neq (p_j, 0)$ are incompatible because they are assigned assumptions from disjoint commitment alphabets \mathcal{C} and \mathcal{D} resp.

Hence any accepted string leads to $((p_1, 0), (p_2, 0))$ through the global states of \widehat{M} and then goes through the global states of \widehat{N} . Since reaching $((p_1, 0), (p_2, 0))$ is equivalent to reaching (p_1^f, p_2^f) of \widehat{M} , one gets that if a string is accepted by P then it is a concatenation of strings from M and N .

On the other hand a string x which is a concatenation of string from $L(M)$ and $L(N)$ will lead to (p_1^f, p_2^f) and then to a global final state of G , hence x will be accepted by P . Therefore, P accepts the concatenation of $L(M)$ and $L(N)$.

We summarize the findings of preceding sections in the theorem below.

Theorem 5.5 *Given $\tilde{\Sigma}$, ACS's over $\tilde{\Sigma}$ are closed under all boolean operations and concatenation.*

More importantly, from our observations about ACS's with multiple initial states, depending upon a given situation, we may freely choose ACS's with either single or multiple initial states.

5.3 ACS's and regular languages

The following theorem constitutes the central result of the chapter:

Theorem 5.6 $\mathcal{L}(ACS_{\tilde{\Sigma}}) = Reg_{\Sigma}$.

The inclusion $\mathcal{L}(ACS_{\tilde{\Sigma}}) \subseteq Reg_{\Sigma}$ 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 $\mathcal{L}(ACS_{\tilde{\Sigma}})$, 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.

5.3.1 Distributed state automaton

A Distributed State Automaton (DSA) on $\bar{\Sigma}$ is a tuple $\mathcal{A} = (A_1, \dots, A_n, \longrightarrow_{\mathcal{A}}, F)$, where

1. for every $i \in Loc$, $A_i = (Q_i, \longrightarrow_{A_i}, s_i^0)$ is the i -th TS on Σ_i ,
2. $\hat{Q} = \prod_{i \in Loc} Q_i$ is the state space of \mathcal{A} ,
3. (s_1^0, \dots, 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 $((p_1, p_2, \dots, p_n), a, (q_1, q_2, \dots, q_n)) \in \longrightarrow_{\mathcal{A}}$ then
 - (a) for all $i \in loc(a)$, $(p_i, a, q_i) \in \longrightarrow_{A_i}$, 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.

When $(\bar{p}, a, \bar{q}) \in \rightarrow_{\mathcal{A}}$ we write it as $\bar{p} \xrightarrow{a}_{\mathcal{A}} \bar{q}$. \mathcal{A} is *deterministic* if $\rightarrow_{\mathcal{A}}$ is a deterministic relation.

5.3.2 Properties of DSA

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

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

Recall also that for a non-empty string $x = a_1 \cdots a_k$, $last(x) \stackrel{\text{def}}{=} a_k$.

Now we define $\Downarrow : (\Sigma^* \times Loc) \rightarrow \Sigma^*$ as follows.

Definition 5.7 $x \Downarrow i \stackrel{\text{def}}{=} \begin{cases} \epsilon & \text{if } x \lceil i = \epsilon \\ y & \text{such that } \exists u \in \Sigma^* : x = yu, last(y) \in \Sigma_i \text{ and } u \lceil i = \epsilon \\ & \text{if } x \lceil i \neq \epsilon. \end{cases}$

$x \Downarrow i$ gives the maximal prefix of x ending in an i -action. For example, let $\bar{\Sigma} = \{\{a, c\}, \{b, c\}\}$. Then $ab \Downarrow 1 = a$, $abcaa \Downarrow 2 = abc$ and $abcaac \Downarrow 1 = abcaac$.

Note that this projection operator is different from the view-projection operator \downarrow of Chapter 3. The notion of causality via locations in \downarrow is completely absent in \Downarrow . Hence, for example, $ab \Downarrow 2 = ab$ while $ab \downarrow 2 = b$.

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 $x \Downarrow i = ya$, then for all $j \in loc(a)$, $x \Downarrow j = x \Downarrow i = ya$.

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

Definition 5.8 The events associated with strings over Σ are defined inductively as follows:

$$\text{event}(x) \stackrel{\text{def}}{=} \begin{cases} ((\epsilon)_{\mathcal{A}}, \epsilon, (\epsilon)_{\mathcal{A}}) & \text{if } x = \epsilon \\ ((y)_{\mathcal{A}}, a, (ya)_{\mathcal{A}}) & \text{if } x = ya \end{cases}$$

An event, say $\text{event}(x)$, 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 5.9 $\text{event}(u) \text{ } i \triangleright j \text{ } \text{event}(v)$ iff there is an $r \in \Sigma^*$ such that $r \Downarrow i$ is a prefix of $r \Downarrow j$, $\text{event}(u) = \text{event}(r \Downarrow i)$ and $\text{event}(v) = \text{event}(r \Downarrow j)$.

It is clear from this and definition of \Downarrow that $\text{event}(u) \text{ } i \triangleright j \text{ } \text{event}(v)$ iff there is an $r \in \Sigma^*$ such that $r = r \Downarrow j$, $\text{event}(v) = \text{event}(r)$ and $\text{event}(u) = \text{event}(r \Downarrow i)$.

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 5.10 For all $x \in \Sigma^*$, $(x \Downarrow i)_{\mathcal{A}}[i] = (x)_{\mathcal{A}}[i]$.

Proposition 5.11 Let $i, j \in \text{loc}(a)$. $\text{event}(xa) \text{ } i \triangleright j \text{ } \text{event}(ya)$ implies $(x)_{\mathcal{A}} = (y)_{\mathcal{A}}$.

Proof: Suppose $\text{event}(xa) \text{ } i \triangleright j \text{ } \text{event}(ya)$. By definition, there exists an $r \in \Sigma^*$ such that $\text{event}(r \Downarrow i) = \text{event}(xa)$ and $\text{event}(r \Downarrow j) = \text{event}(ya)$ and $r \Downarrow i$ is a prefix of $r \Downarrow j$. (See Fig. 5.3).

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 $(x)_{\mathcal{A}} = (r')_{\mathcal{A}} = (r'')_{\mathcal{A}} = (y)_{\mathcal{A}}$. ■

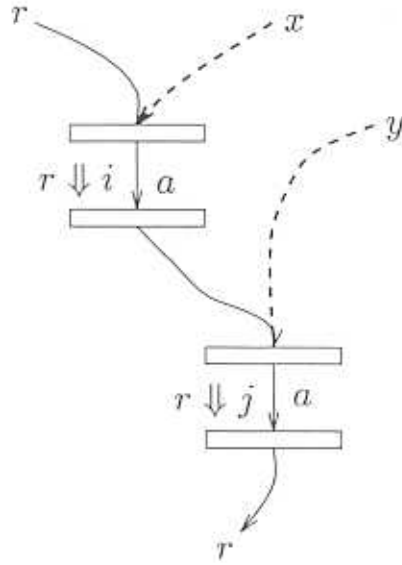


Figure 5.3: $event(xa) \text{ } i \triangleright j \text{ } event(ya)$

Definition 5.12 A DSA \mathcal{A} is **stutter-free** if for every $\bar{p}, \bar{q} \in \hat{Q}$,

$$\bar{p} \xrightarrow{a}_{\mathcal{A}} \bar{q} \text{ implies for all } i \in loc(a), \bar{p}[i] \neq \bar{q}[i].$$

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

Proposition 5.13 Suppose \mathcal{A} is stutter-free. Let $u, w \in \Sigma^*$, $a \in \Sigma_i$ and $k \in Loc$. Then it is never the case that both $event(ua) \text{ } i \triangleright k \text{ } event(w)$ and $event(u \downarrow i) \text{ } i \triangleright k \text{ } event(w)$.

Proof: Suppose that both $event(ua) \text{ } i \triangleright k \text{ } event(w)$ and $event(u \downarrow i) \text{ } i \triangleright k \text{ } event(w)$.

Claim: Let $x, y, z \in \Sigma^*$, $i, k \in Loc$ and suppose that both $event(x) \text{ } i \triangleright k \text{ } event(z)$ and $event(y) \text{ } i \triangleright k \text{ } event(z)$. 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 $event(x) \text{ } i \triangleright k \text{ } event(z)$. 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 5.10 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. \blacksquare

Definition 5.14 \mathcal{A} is **four-alternation-free** if for all $i, k \in Loc$, $a, c \in \Sigma_i \setminus \Sigma_k$, $b, d \in \Sigma_k \setminus \Sigma_i$ and $r \in \Sigma^*$, $(r \Downarrow i = r_1 a, r_1 \Downarrow k = r_2 b, r_2 \Downarrow i = r_3 c, r_3 \Downarrow k = r_4 d \text{ and } r_4 \Downarrow i = r_5)$ 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. 5.4).

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

Proposition 5.15 Suppose \mathcal{A} is four-alternation-free. Let $u, y \in \Sigma^*$, $a \in \Sigma$, $i \in loc(a)$, $k \notin loc(a)$. Then, $(event(ua) \dashv\dashv k event(y) \text{ and } event(y) \dashv\dashv i event(u \Downarrow i))$ implies $(y)_{\mathcal{A}}[k] = (u)_{\mathcal{A}}[k]$.

Proof: By definition of $\dashv\dashv$, we have

1. There is an $r \in \Sigma^*$ 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. there is an $r' \in \Sigma^*$ 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,

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

¹We really do not have any intuition to offer as to why only “four” works. Vaguely, this seems to be connected to some kind of minimal unfolding of the DSA necessary for the synthesis.

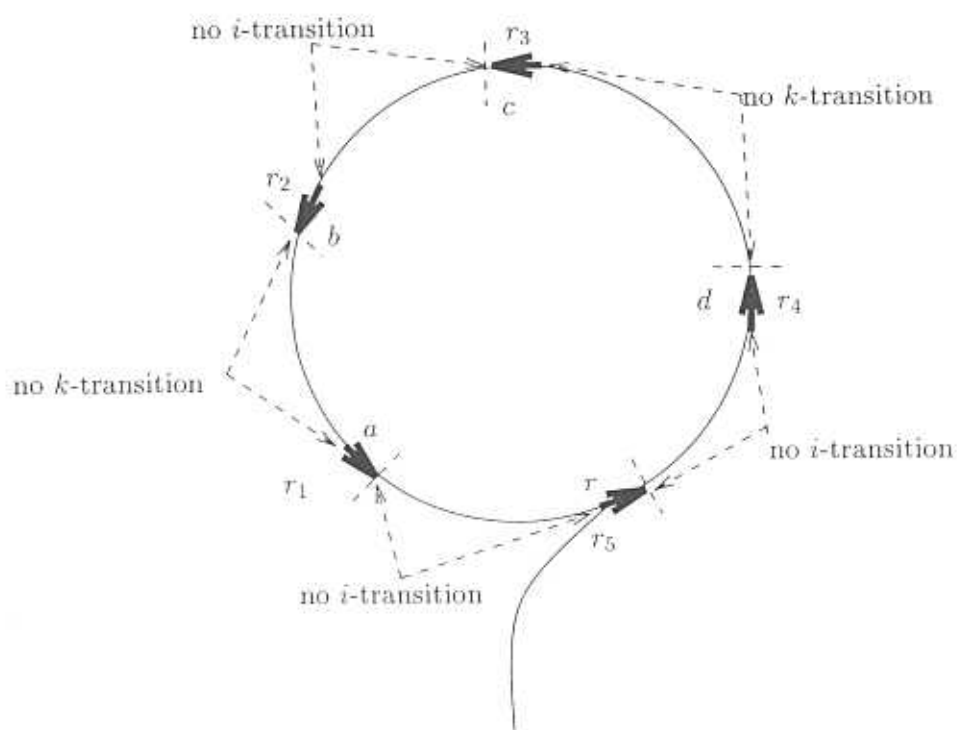


Figure 5.4: When \mathcal{A} is four-alternation-free, such loops are not present in \mathcal{A} .

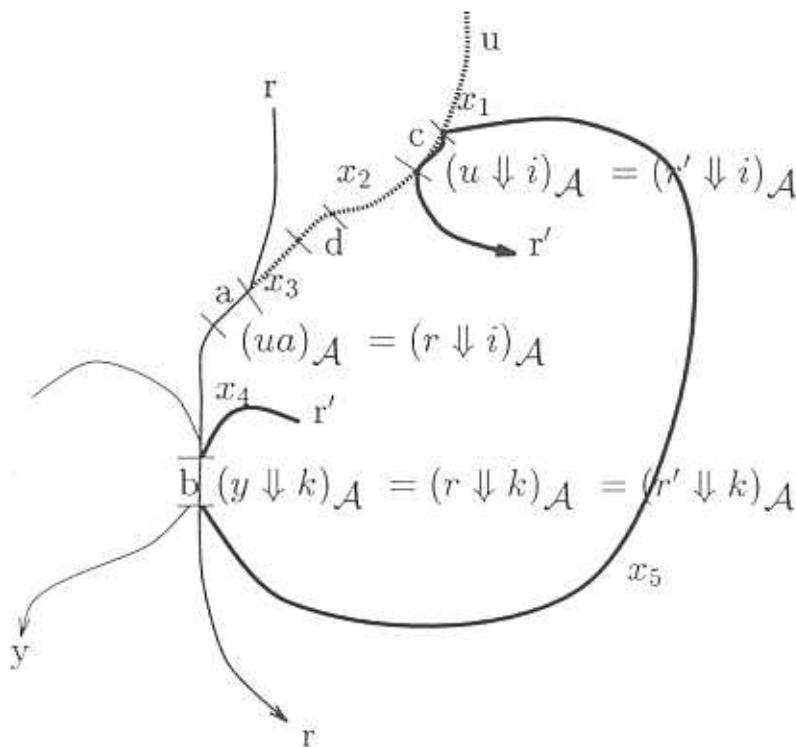


Figure 5.5: The case $u \Downarrow k \neq (u \Downarrow i) \Downarrow k$.

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,

$$\begin{aligned}
 (u)_{\mathcal{A}}[k] &= (u \Downarrow k)_{\mathcal{A}}[k] && \text{from Proposition 5.7} \\
 &= ((u \Downarrow i) \Downarrow k)_{\mathcal{A}}[k] && \text{by the claim} \\
 &= (u \Downarrow i)_{\mathcal{A}}[k] && \text{from Proposition 5.7 again} \\
 &= (r' \Downarrow i)_{\mathcal{A}}[k] && \text{from (2) above} \\
 &= ((r' \Downarrow i) \Downarrow k)_{\mathcal{A}}[k] && \text{Proposition 5.7 yet again} \\
 &= (r' \Downarrow k)_{\mathcal{A}}[k] && \text{from (2) above} \\
 &= (y)_{\mathcal{A}}[k] && \text{from (2) above}
 \end{aligned}$$

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. 5.5.

- 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 $u \Downarrow i = x_1 c$, $c \in \Sigma_i$, $d \in \Sigma_k \setminus \Sigma_i$.
- 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 $b \in \Sigma_k \setminus \Sigma_i$, 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

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

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

Hence, we get

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

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

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

Also, one verifies that

1. $y \Downarrow i = x_1c \cdot x_2d \cdot x_3a \cdot x_4b \cdot x_5e$, since $c \in \Sigma_i$. Let $y_1 = x_1c \cdot x_2d \cdot x_3a \cdot x_4b \cdot x_5$.
2. $y_1 \Downarrow k = x_1c \cdot x_2d \cdot x_3a \cdot x_4b$, since $b \in \Sigma_k$ and $x_5 \in (\Sigma \setminus \Sigma_k)^*$. Let $y_2 = x_1c \cdot x_2d \cdot x_3a \cdot x_4$.
3. $y_2 \Downarrow i = x_1c \cdot x_2d \cdot x_3a$, since $a \in \Sigma_i$ and $x_4 \in (\Sigma \setminus \Sigma_i)^*$. Let $y_3 = x_1c \cdot x_2d \cdot x_3$.
4. $y_3 \Downarrow k = x_1c \cdot x_2d$, since $d \in \Sigma_k$ and $x_3 \in (\Sigma \setminus \Sigma_k)^*$. Let $y_4 = x_1c \cdot x_2$.
5. Finally, $y_4 \Downarrow i = x_1c = y_5$, 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. \blacksquare

5.3.3 From DSA to ACS

The idea behind introducing the properties 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 lemma establishes the required correspondence between DSA's and ACS's over $\bar{\Sigma}$.

Theorem 5.16 *Let \mathcal{A} be a stutter-free and four-alternation-free deterministic DSA on $\bar{\Sigma}$. Then, there exists an ACS \tilde{M} on $\bar{\Sigma}$ such that $L(\mathcal{A}) = L(\tilde{M})$.*

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

Definition 5.17 *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 $\widetilde{M} = (M_1, \dots, M_n, \langle f_1, \dots, f_n \rangle, F)$ over $(\widetilde{\Sigma}, \widetilde{\mathcal{C}})$ from the DSA \mathcal{A} as follows.

For each $i \in Loc$, i -local TS $M_i = (Q_i, \xrightarrow{i}, q_i^0)$ where

- $Q_i = S_i$,
- $q_i^0 = event(\epsilon)$, and
- $p \xrightarrow{a}_i q$ 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

$$F = \{(event(x \Downarrow 1), \dots, event(x \Downarrow n)) \mid x \in L(\mathcal{A})\}.$$

Let $(\widehat{M} = (\widehat{Q}, \longrightarrow, (q_1^0, \dots, q_n^0)), F)$ be the compatible product automaton of \widetilde{M} where \widehat{Q} is the set of all compatible states. Since $L(\widehat{M}) \stackrel{\text{def}}{=} L(\widehat{M}, F)$, 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: By construction, $f_i(event(x))(i) = \{event(y \Downarrow i) \mid event(y \Downarrow i) = event(x)\} = \{event(x)\}$, because $x \Downarrow i = x$ for $event(x) \in Q_i$. ■

An immediate consequence is the following.

2. For all $i, j \in Loc$, $f_i(event(x))(j) \preceq_j f_j(event(y))(j)$ 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 $s = \text{event}(x \Downarrow j)$ and $q = \text{event}(x \Downarrow i)$. Since either $x \Downarrow j \preceq x \Downarrow i$ or vice-versa, we have either $s \text{ j} \triangleright \text{i} q$ or $q \text{ i} \triangleright \text{j} s$ and hence $s \in \text{RHS}$.

The other inclusion follows from the definition of assumption maps. ■

We now observe some properties of the compatible product.

1. For all $x \in \Sigma^*$, $(\text{event}(x \Downarrow 1), \dots, \text{event}(x \Downarrow n))$ is compatible.

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

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

Therefore, the property is proved by setting y to be x . ■

2. For all $x \in \Sigma^*$, $(\text{event}(x \Downarrow 1), \dots, \text{event}(x \Downarrow n)) \xrightarrow{a} (\text{event}(xa \Downarrow 1), \dots, \text{event}(xa \Downarrow n))$.

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.

(a) For $i \notin \text{loc}(a)$, $x \Downarrow i = xa \Downarrow i$, hence, $\text{event}(x \Downarrow i) = \text{event}(xa \Downarrow i)$.

(b) From the definition of \xrightarrow{i} , it directly follows that for all $i \in \text{loc}(a)$,

$\text{event}(x \Downarrow i) \xrightarrow{a} \text{event}(xa \Downarrow i)$, and we are done. ■

In the light of the above, by a simple induction on the length of strings we get for all $x \in \Sigma^*$, $(\text{event}(\epsilon), \dots, \text{event}(\epsilon)) \xrightarrow{x} (\text{event}(x \Downarrow 1), \dots, \text{event}(x \Downarrow n))$.

These observations immediately give us the following lemma proving one direction of Theorem 5.16.

Lemma 5.18 $L(\mathcal{A}) \subseteq L(\widetilde{M})$.

Proof: Let $x \in L(\mathcal{A})$. Then, $(\text{event}(x \Downarrow 1), \dots, \text{event}(x \Downarrow n)) \in F$ by construction. Also, from the observation above, $\text{event}(\epsilon), \dots, \text{event}(\epsilon) \xrightarrow{x} (\text{event}(x \Downarrow 1), \dots, \text{event}(x \Downarrow n))$. Hence, $x \in L(\widetilde{M})$.

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

Lemma 5.19 $L(\widetilde{M}) \subseteq L(\mathcal{A})$.

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

Claim: $\forall x \in \Sigma^*, (event(\epsilon), \dots, event(\epsilon)) \xrightarrow{x} (event(z_1), \dots, event(z_n))$ 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 $(event(\epsilon), \dots, event(\epsilon)) \xrightarrow{ya} (event(z_1), \dots, event(z_n))$. Then there is a state $(event(y_1), \dots, event(y_n)) \in \widehat{Q}$ such that

$$(event(\epsilon), \dots, event(\epsilon)) \xrightarrow{y} (event(y_1), \dots, event(y_n)), \text{ and} \\ (event(y_1), \dots, event(y_n)) \xrightarrow{a} (event(z_1), \dots, event(z_n)).$$

By induction hypothesis

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

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

Case 1: $i \notin loc(a)$. 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: $i \in loc(a)$. Since $ya \Downarrow i = ya$, we have to show $event(z_i) = event(ya)$.

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

$$event(y \Downarrow j) = event(u_j \Downarrow j), \text{ and} \quad (6)$$

$$event(u_j a) = event(z_j). \quad (7)$$

So it suffices to show that $event(u_i a) = event(y a)$. 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 $k \in loc(a)$. Because the state $(event(z_1), \dots, event(z_n))$ is compatible, $event(z_i) \in f_k(event(z_k))(i)$. Hence, by (7), $event(u_i a) \in f_k(event(u_k a))(i)$.

From the property of assumption maps, it follows that either

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

In both the cases, from Prop. 5.11, $(u_i)_{\mathcal{A}} = (u_k)_{\mathcal{A}}$. Hence $(u_i)_{\mathcal{A}}[k] = (u_k)_{\mathcal{A}}[k]$.

By (6) and using Proposition 5.7 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 $k \notin loc(a)$.

1. By compatibility of $(event(z_1), \dots, event(z_n))$, $event(z_i) \in f_k(event(z_k))(i)$.

Therefore, by Property of assumption maps,

$$[\text{either } event(z_i) \text{ i-}\triangleright\text{-}k \text{ event}(z_k) \text{ or } event(z_k) \text{ k-}\triangleright\text{-}i \text{ event}(z_i)].$$

2. By compatibility of $(event(y_1), \dots, event(y_n))$, $event(y_i) \in f_k(event(y_k))(i)$.

Therefore, by Property of assumption maps

$$[\text{either } event(y_i) \text{ i-}\triangleright\text{-}k \text{ event}(y_k) \text{ or } event(y_k) \text{ k-}\triangleright\text{-}i \text{ event}(y_i)].$$

We consider the possible cases. Remember that we are considering the case when $i \in loc(a)$ and $k \notin loc(a)$. Hence, in the following, $event(z_k) = event(y_k)$.

Case $event(z_k) \text{ k-}\triangleright\text{-}i \text{ event}(z_i)$: 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

$$\begin{aligned}
(y)_{\mathcal{A}}[k] &= (y \Downarrow k)_{\mathcal{A}}[k] \text{ by Proposition 5.7} \\
&= (y_k)_{\mathcal{A}}[k] \text{ by induction hypothesis (5)} \\
&= (r \Downarrow k)_{\mathcal{A}}[k] \text{ since } \text{event}(y_k) = \text{event}(r \Downarrow k) \\
&= (r)_{\mathcal{A}}[k] \text{ by Proposition 5.7} \\
&= (u_i a)_{\mathcal{A}}[k] \text{ since } \text{event}(u_i a) = \text{event}(r) \\
&= (u_i)_{\mathcal{A}}[k] \text{ since } k \notin \text{loc}(a),
\end{aligned}$$

which is the required result.

Case $\text{event}(z_i) \text{ i-}\triangleright\text{-k } \text{event}(z_k)$ and $\text{event}(y_i) \text{ i-}\triangleright\text{-k } \text{event}(z_k)$:

By (7), $\text{event}(z_i) = \text{event}(u_i a)$ and by (5) and (6) we get $\text{event}(y_i) = \text{event}(u_i \Downarrow i)$.

Hence, for this case, $\text{event}(u_i a) \text{ i-}\triangleright\text{-k } \text{event}(z_k)$ and $\text{event}(u_i \Downarrow i) \text{ i-}\triangleright\text{-k } \text{event}(z_k)$.

Then, since we assume that \mathcal{A} is *stutter-free*, by Prop. 5.13 this case is not possible.

Case $\text{event}(z_i) \text{ i-}\triangleright\text{-k } \text{event}(z_k)$ and $\text{event}(z_k) \text{ k-}\triangleright\text{-i } \text{event}(y_i)$

Recall that for the case under consideration (namely, $i \in \text{loc}(a)$, $k \notin \text{loc}(a)$),

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

Hence, $\text{event}(u_i a) \text{ i-}\triangleright\text{-k } \text{event}(y_k)$ and $\text{event}(y_k) \text{ k-}\triangleright\text{-i } \text{event}(u_i \Downarrow i)$. Then, since the DSA \mathcal{A} is four-alternation-free, from Proposition 5.15, we have $(u_i)_{\mathcal{A}}[k] = (y_k)_{\mathcal{A}}[k]$. Then, from (5), applying Proposition 5.7, we finally get $(u_i)_{\mathcal{A}}[k] = (y)_{\mathcal{A}}[k]$.

Thus we have proved that for all locations $k \in \text{Loc}$, $(u_i)_{\mathcal{A}}[k] = (y)_{\mathcal{A}}[k]$. Thereby we prove the claim and the lemma. ■

Lemmas 5.18 and 5.19 prove Theorem 5.16.

Theorem 5.16 *Let \mathcal{A} be a stutter-free and four-alternation-free deterministic DSA on $\tilde{\Sigma}$. Then, there exists an ACS \tilde{M} on $\tilde{\Sigma}$ such that $L(\mathcal{A}) = L(\tilde{M})$.*

5.3.4 Regular languages and DSA

Because of Theorem 5.16, in order to show that ACS's over $\tilde{\Sigma}$ characterize the class of all regular languages over Σ , it suffices to prove the following theorem.

Theorem 5.20 *Let the class of deterministic DSA's over $\tilde{\Sigma}$ that are stutter-free and four-alternation-free be denoted as FDSA. Then $\mathcal{L}(FDSA_{\tilde{\Sigma}}) = Reg_{\Sigma}$.*

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 $\mathcal{A}_{\tilde{\Sigma}}$ such that $L(\mathcal{A}_{\tilde{\Sigma}}) = L$ and $\mathcal{A}_{\tilde{\Sigma}}$ has the required properties.

Definition 5.21 (A labeling scheme) *Let $l : \Sigma^* \rightarrow (Loc \times Loc \rightarrow \{0, 1, 2\})$ be a labeling function defined inductively as follows: $l(\epsilon) \stackrel{\text{def}}{=} C$ where $C(i, j) = 0$ for every $i, j \in Loc$. Let $l(x) = C$. Then $l(xa) \stackrel{\text{def}}{=} C'$ where*

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

(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 Let $\tilde{\Sigma} = \{\{a, c\}, \{b, c\}\}$. 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 5.22 *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 $a \in \Sigma_i$: $x = x \downarrow i$ and hence the proposition holds.

Case $a \notin \Sigma_i$: 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 5.23 *If $x \downarrow j$ is a proper prefix of $x \downarrow i$ then $l(x)(i, j) = l(x)(j, i) + 1 \bmod 3$.*

Proof: By the preceding result, $l(x)(i, j) = l(x \downarrow i)(i, j)$ 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 $u[j] = \epsilon$. Hence, $x \downarrow j = (x \downarrow j \cdot u) \downarrow j$. Then from previous observation we can derive:

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

■

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

Lemma 5.25 *If (for all $x, y \in \Sigma^*$ and for all $i \in \text{Loc}$, $x \equiv_i y$) then $x \setminus L = y \setminus L$.*

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

Case $x = \epsilon$ and $y \neq \epsilon$: Let $j \in \text{loc}(\text{last}(y))$. 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 , $x \setminus L = y \setminus L$.

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

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

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, $x \setminus L = y \setminus L$.

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 5.23, $l(y) = C'$, where $C'_{ji} = C_{ij} + 1 \bmod 3$.

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 5.23, $C_{ij} = C'_{ji} + 1 \bmod 3$.

From the previous paragraph, we then get, $C_{ij} = C_{ij} + 2 \bmod 3$, which is a contradiction. Thus we prove the claim and the lemma. \blacksquare

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 5.26 (*i*-parity) Define $i\text{-parity} : \Sigma^* \rightarrow \{0, 1\}$ as: for all $x \in \Sigma^*$,

$$i\text{-parity}(x) \stackrel{\text{def}}{=} |x[i]| \bmod 2.$$

For example, let $\bar{\Sigma} = \{\{a, c\}, \{b, c\}\}$. Then, $1\text{-parity}(a) = 1$, $2\text{-parity}(bab) = 0$ and $1\text{-parity}(abab) = 0$.

The following proposition regarding *i*-parity of strings follows easily from the definition of \Downarrow .

Proposition 5.27 For all $x \in \Sigma^*$ and $i \in Loc$, $i\text{-parity}(x) = i\text{-parity}(x \Downarrow i)$. Therefore, for all $i \notin loc(a)$, $i\text{-parity}(x) = i\text{-parity}(xa)$.

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

Definition 5.28 For all $x, y \in \Sigma^*$, $i \in Loc$, $x \sim_i y$ iff $x \equiv_i y$ and $i\text{-parity}(x) = i\text{-parity}(y)$.

Let $[x]_i$ denote the equivalence class under \sim_i containing x .

The following results follow directly from definition of \sim_i and from Lemma 5.25

Corollary 5.29 1. $[xa]_i = [x]_i$ for all $i \notin \text{loc}(a)$.

2. If (for all $x, y \in \Sigma^*$ and for all $i \in \text{Loc}$, $x \sim_i y$) then $x \setminus L = y \setminus L$.

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

5.3.5 Properties of the constructed DSA \mathcal{A}

1. We observe that by construction, $\rightarrow_{\mathcal{A}}$ is deterministic.

2. \mathcal{A} is stutter-free.

Proof: Suppose $\bar{p} \xrightarrow{a}_{\mathcal{A}} \bar{q}$. We need to show that for all $i \in \text{loc}(a)$, $\bar{p}[i] \neq \bar{q}[i]$.

By our construction, there is an $x \in \Sigma^*$ such that $\bar{p} = ([x]_1, [x]_2, \dots, [x]_n)$ and $\bar{q} = ([xa]_1, [xa]_2, \dots, [xa]_n)$. For all $i \in \text{loc}(a)$, $i\text{-parity}(x) \neq i\text{-parity}(xa)$, hence $[x]_i \neq [xa]_i$. Therefore, $\bar{p}[i] \neq \bar{q}[i]$. ■

3. \mathcal{A} is four-alternation-free.

Proof: Let $r \in \Sigma^*$ be such that $r \Downarrow i = r_1a$, $r_1 \Downarrow k = r_2b$, $r_2 \Downarrow i = r_3c$, $r_3 \Downarrow k = r_4d$ 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 $l(r_5)(i, k) = c \bmod 3$. We know that for all $x \in \Sigma^*$, $i, j \in \text{Loc}$, $l(x)(i, j) = l(x \Downarrow i)(i, j)$. We repeatedly

use this observation below.

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

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

Lemma 5.30 $\forall x \in \Sigma^* : (([e]_1, 0), ([e]_2, 0), \dots, ([e]_n, 0)) \xrightarrow{\mathcal{F}_A} ([x]_1, [x]_2, \dots, [x]_n)$.

Proof: By the construction, \longrightarrow_A is deterministic and by Corollary 5.29.1 \longrightarrow_A satisfies the *asynchrony* property. Hence we get the result by induction on $|x|$. ■

The following proposition now gives us the required conclusion.

Proposition 5.31 $L(\mathcal{A}) = L$.

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

$$([\epsilon]_1, [\epsilon]_2, \dots, [\epsilon]_n) \Rightarrow_{\mathcal{A}} ([x]_1, [x]_2, \dots, [x]_n) \in F.$$

Hence $x \in L(\mathcal{A})$.

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

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

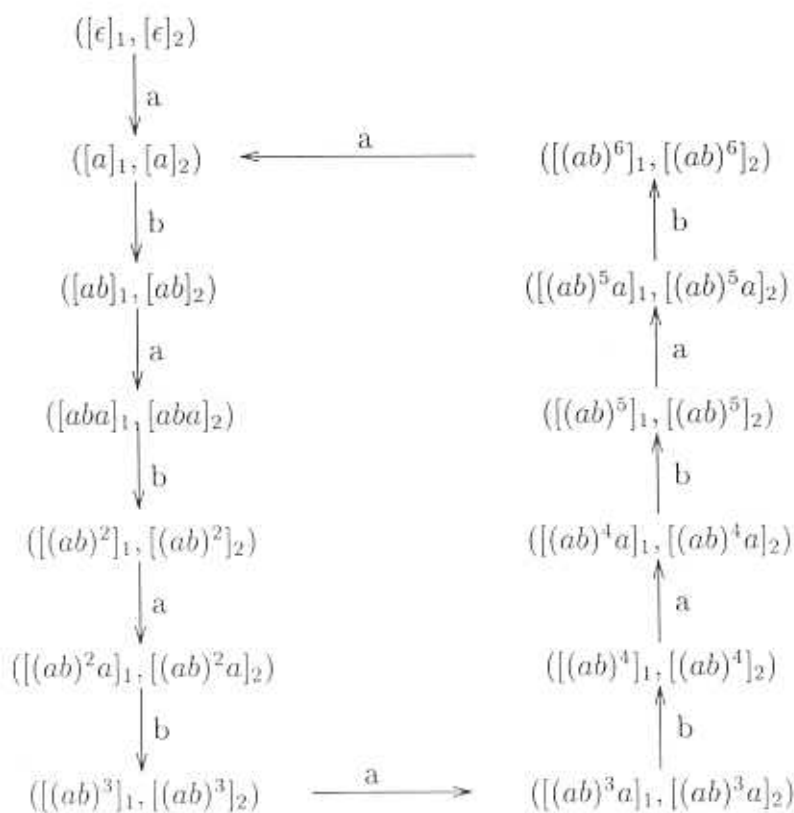
for some $y \in L$.

By Corollary 5.29.2, $x \setminus L = y \setminus L$. So since $y \in L$, $x \in L$. ■

Example Let $\tilde{\Sigma} = \{\{a\}, \{b\}\}$. In Fig. 5.6 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 $Q_2 = \{[\epsilon]_2\} \cup \{[(ab)^m]_2 \mid m \leq 6\}$.



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

$$(ab)^m \downarrow 2 = (ab)^m$$

$$(ab)^ma \downarrow 1 = (ab)^ma$$

$$(ab)^ma \downarrow 2 = (ab)^m$$

Figure 5.6: A DSA for the language $(ab)^*$.

5.4 Compatible shuffle and Kleene's theorem

A product operation on finite state automata corresponds to a shuffle operation on regular languages. Thus we can ask, what manner of shuffle 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 $\tilde{\Sigma} = (\Sigma_1, \dots, \Sigma_n)$ and a finite commit alphabet $\tilde{\mathcal{C}}$. 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.

5.4.1 Extending alphabets

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 $\langle a, \phi \rangle$ where $\phi \in \Phi$. Notice that these definitions differ from similar-looking ones in case of AT-systems of Chapter 4.

Definition 5.32 *Given a distributed alphabet, $\tilde{\Sigma}$ and a commit alphabet $\tilde{\mathcal{C}}$, we define extended alphabets as follows:*

$$\Sigma_i^c \stackrel{\text{def}}{=} \{\langle a, \phi \rangle \mid a \in \Sigma_i, \phi \in \Phi\}; \quad \tilde{\Sigma}^c \stackrel{\text{def}}{=} \langle \Sigma_1^c, \dots, \Sigma_n^c \rangle; \quad \Sigma^c \stackrel{\text{def}}{=} \bigcup_{i \in Loc} \Sigma_i^c.$$

5.4.2 Compatible shuffle of strings

We want to define n -way shuffle for strings x_1, \dots, 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, $\Xi = \{\xi \mid \xi : Loc \rightarrow \Phi\}$. 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 5.33 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 5.34 $\Sigma^\Xi = \{ \langle a, \xi \rangle \mid a \in \Sigma \text{ and } \xi \in \Xi \}$.

We use \hat{x}, \hat{y}, \dots 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. Naturally, the strings can not be arbitrary. They have to somehow capture the compatibility condition internally. We call these strings *good*. Formally,

Definition 5.35 Let $\hat{x} = \langle a_1, \xi_1 \rangle \langle a_2, \xi_2 \rangle, \dots, \langle a_k, \xi_k \rangle \in \Sigma^{\Xi*}$. Then, \hat{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.

By this definition, ϵ 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 will need two kinds of projection maps. The first is the *commit erasure* map: $\sigma : \Sigma^{c*} \rightarrow \Sigma^*$ defined as : $\sigma(\langle a_1, \phi_1 \rangle \dots \langle a_k, \phi_k \rangle) \stackrel{\text{def}}{=} a_1 \dots a_k$. 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: $\widehat{\lceil}: (\Sigma^{\Xi^*} \times Loc) \rightarrow \Sigma^{c^*}$ defined by:

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

Recall that \lceil is the component projection map: $\lceil: (\Sigma^* \times Loc) \rightarrow \Sigma^*$ defined as:

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

Definition 5.36 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 $x_i \in \Sigma_i^{c^*}$, $i \in \{1, \dots, n\}$. A string $x \in \Sigma^*$ is said to be **generated** by (x_1, x_2, \dots, 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, \dots, 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 5.37 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:

$$(L_1 || L_2 || \dots || L_n)_\xi \stackrel{\text{def}}{=} \{x \in \Sigma^* \mid x \text{ is generated by a tuple } (x_1, x_2, \dots, x_n) \text{ under } \xi, \text{ where for all } i, x_i \in L_i\}.$$

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

Definition 5.38 Let $\mathcal{L}(AC - shuffle)_\Sigma$ denote the least class that includes the set $\{L \subseteq \Sigma^* \mid \text{for some commit alphabet } \tilde{C}, \text{ there exist regular languages } L_i \subseteq \Sigma_i^{c^*} \text{ such that } L = L_1 || \dots || L_n\}$ and is closed under union.

5.4.3 ACS's and commitment structure

There is a fairly obvious association between states and runs of an ACS and good strings over Σ^\pm . We make it explicit in the following. Let $\tilde{M} = (M_1, M_2, \dots, M_n, \langle f_1, \dots, f_n \rangle, F)$ be an ACS and $\widehat{M} = (\widehat{Q}, \longrightarrow, \langle q_0^1, \dots, q_0^n \rangle, F)$ be the compatible product of \tilde{M} .

Definition 5.39 Let $(q_1, \dots, q_n) \in \widehat{q}$. Then, $\text{env}(q_1, \dots, q_n) \stackrel{\text{def}}{=} \xi$, where $\xi(i) = f_i(q_i)$, for all $i \in \text{Loc}$.

Let (q_1, \dots, q_n) be a compatible state and $\xi = \text{env}(q_1, \dots, q_n)$. Then ξ is feasible. This is because of the following reason. Since (q_1, \dots, q_n) is compatible, for all $i, j \in \text{Loc}$, $f_i(q_i)(j) \preceq_j f_j(q_j)(j)$. Then by the construction, for all $i, j \in \text{Loc}$, $\xi(i)(j) \preceq_j \xi(j)(j)$. This implies feasibility of ξ .

We can associate strings over Σ^{c*} with runs of \widehat{M} in a canonical fashion. Fix $x = a_1 a_2 \dots a_k$, and a run $\rho = (q_1^0, \dots, q_n^0) \xrightarrow{a_1} (q_1^1, \dots, q_n^1) \dots \xrightarrow{a_k} (q_1^k, \dots, q_n^k)$ on x in \widehat{M} . Define $c(\rho) = \langle a_1, \xi_1 \rangle \dots \langle a_k, \xi_k \rangle \in \Sigma^{c*}$ by: for $1 \leq l \leq k$, $\xi_l = \text{env}(q_1^l, \dots, q_n^l)$.

Note that when $k = 0$, that is, when $x = \epsilon$, $c(\rho) = \epsilon$. The initial environment associated with the run is defined as $\text{env}((q_1, \dots, q_n))$. Note that $\text{env}(q_1, \dots, q_n)$ is a feasible assumption environment (because (q_1, \dots, q_n) is compatible.)

Proposition 5.40 Let ρ be a run in $(q_1, \dots, q_n) \xrightarrow{x} (p_1, \dots, p_n)$. Then, $c(\rho)$ is a witness for x under $\text{env}(q_1, \dots, q_n)$.

Proof: By definition, $\sigma(c(\rho)) = x$. Hence it suffices to show that $c(\rho)$ is good w.r.t. $\text{env}(q_1, \dots, q_n)$.

Let the run ρ be $\langle q_0^1, \dots, q_0^n \rangle \xrightarrow{a_1} (q_1^1, \dots, q_n^1) \dots \xrightarrow{a_k} (q_1^k, \dots, q_n^k)$.

Fix l such that $1 \leq l \leq k$. For all $j \notin \text{loc}(a_l)$, $q_j^{l-1} = q_j^l$ 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. $\text{env}(q_1, \dots, q_n)$. ■

The above propositions give us an important result regarding runs in the product and their projections.

Proposition 5.41 *Let ρ denote a run $(q_1, \dots, q_n) \xrightarrow{x} (p_1, \dots, p_n)$ in \widehat{M} . Then for all $i \in Loc$, there exist $x_i \in \Sigma_i^{c*}$ such that $q_i \xrightarrow{x_i}^c p_i$ and x is generated by the tuple (x_1, x_2, \dots, x_n) under $env(q_1, \dots, q_n)$.*

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

Proposition 5.42 *Let $\hat{x} = \langle a_1, \xi_1 \rangle \langle a_2, \xi_2 \rangle, \dots, \langle a_k, \xi_k \rangle \in \Sigma^{\Xi*}$ be good w.r.t. $\xi_0 = env(q_1, \dots, q_n)$ such that $q_i \xrightarrow{\hat{x} \upharpoonright i}^c p_i$. Then, $\xi_k(i) = f_i(p_i)$. (Since ξ_k is feasible this implies that (p_1, \dots, p_n) is compatible.)*

Proof: Let l be the last i -action in \hat{x} . If $l = 0$ (meaning $\hat{x} \upharpoonright i$ is empty), then $p_i = q_i$ and by goodness of \hat{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 $q_i \xrightarrow{y}^c r_i \xrightarrow{\langle a_l, \xi_l(i) \rangle}^c p_i$, where $\hat{x} \upharpoonright i = y \cdot \langle a_l, \xi_l(i) \rangle$. Hence, $f_i(p_i) = \xi_l(i)$. By goodness of \hat{x} , $\xi_l(i) = \xi_k(i)$. Therefore, $f_i(p_i) = \xi_k(i)$ and we are done. ■

Proposition 5.43 *Suppose $x \in \Sigma^*$ and for all $i \in Loc$, there exist $x_i \in \Sigma_i^{c*}$ such that $q_i \xrightarrow{x_i}^c p_i$ and x is generated by the tuple (x_1, x_2, \dots, x_n) under $env(q_1, \dots, q_n)$. Then $(q_1, \dots, q_n) \xrightarrow{x} (p_1, \dots, p_n)$ 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 $q_i \xrightarrow{x_i}^c p_i$ and there exists a witness $\hat{x} \in \Sigma^{\Xi*}$ of x under $env(q_1, \dots, q_n)$ such that $\hat{x} \upharpoonright i = x_i$. We have to show that $(q_1, \dots, q_n) \xrightarrow{x} (p_1, \dots, p_n)$. Note that since \hat{x} is good w.r.t. $env(q_1, \dots, q_n)$, (q_1, \dots, q_n) is compatible from the definition of goodness. From Proposition 5.42 (p_1, \dots, p_n) is also compatible. Hence both (q_1, \dots, q_n) and (p_1, \dots, p_n) are in \widehat{M} .

Let $x = \epsilon$. Then the witness \hat{x} must be ϵ since $\sigma(\hat{x}) = x$. This implies, for all $i \in Loc$ $x_i = \epsilon$. Hence, $q_i = p_i$ for all i . Then, it is obvious that $(q_1, \dots, q_n) \xRightarrow{x} (p_1, \dots, p_n)$.

For the induction step, let $x = ya$. Then the witness \hat{x} for x must be of the form $\hat{x} = \hat{y} \cdot \langle a, \xi \rangle$, where $\sigma(\hat{y}) = y$. Since \hat{x} is a witness under $env(q_1, \dots, q_n)$ it is good w.r.t. $env(q_1, \dots, q_n)$. Hence, \hat{y} must also be good w.r.t. $env(q_1, \dots, q_n)$. Therefore, \hat{y} is a witness of y under $env(q_1, \dots, q_n)$. Let $\hat{y}[i] = y_i$ for all $i \in Loc$. Then, \hat{y} is generated by (y_1, \dots, y_n) under $env(q_1, \dots, q_n)$.

It is given that for all $i \in Loc$, $\hat{x}[i] = x_i$. Hence for every $i \notin loc(a)$, $x_i = \hat{x}[i] = \hat{y}[i] = y_i$ and for every $i \in loc(a)$, $x_i = y_i \cdot \langle a, \xi(i) \rangle$.

It is also given that for all $i \in Loc$, $q_i \xRightarrow{x_i} p_i$. Hence, for all $i \notin loc(a)$, $q_i \xRightarrow{y_i} p_i$ and for all $i \in loc(a)$, there is an $r_i \in Q_i$ such that $q_i \xRightarrow{y_i} r_i \xrightarrow{\langle a, \xi(i) \rangle} p_i$. Therefore, $f_i(p_i) = \xi(i)$.

Since we have:

1. for all $i \notin loc(a)$, $q_i \xRightarrow{y_i} p_i$ and
2. for all $i \in loc(a)$, $q_i \xRightarrow{y_i} r_i$,

by induction hypothesis, $(q_1, \dots, q_n) \xRightarrow{y} (s_1, \dots, s_n)$, where for $i \notin loc(a)$, $s_i = p_i$ and for $i \in loc(a)$ $s_i = r_i$.

By asynchrony, from the fact that for $i \in loc(a)$, $s_i = r_i \xrightarrow{\langle a, \xi(i) \rangle} p_i$, it follows that $(s_1, \dots, s_n) \xrightarrow{a} (p_1, \dots, p_n)$. Therefore, finally, $(q_1, \dots, q_n) \xRightarrow{ya} (p_1, \dots, p_n)$, as required. ■

5.4.4 Equivalence of AC-Shuffle and AC-systems

In the following, we 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 a Moore machine) along with the labels of transition so that languages accepted by local automata are actually over Σ_i^c . For example, if $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2$, q_0 is the initial state and q_2 is the final state, then we say that this is an accepting path for $\langle a_1, f_i(q_1) \rangle \cdot \langle a_2, f_i(q_2) \rangle$.

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 the language acceptance (in the above sense) of AC-automata. Let $(M_i = (Q_i, \rightarrow_i, q_i^0), f_i)$ be an AC-transition system over Σ_i . We transform M_i into a TS $M'_i = (Q'_i, \rightarrow_i^c, q_i^{c0})$ over Σ_i^c where, $Q'_i = Q_i$, $q_i^{c0} = q_i^0$ and $\rightarrow_i^c \subseteq Q'_i \times \Sigma_i^c \times Q'_i$ is defined as follows: $p \xrightarrow{<a, \phi>}_i^c q$ iff $p \xrightarrow{a}_i q$ and $\phi = f_i(q)$. This one step transition can be extended to \Rightarrow_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 5.44 $L \in \text{Reg}_{\Sigma_i^c}$ 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 $L \in \text{Reg}_{\Sigma_i^c}$. Since L is regular over Σ_i^c , there exist some FSA $(A = (Q, \rightarrow, q^0), F)$ 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 $M = (P, \Rightarrow, p^0)$ where,

- $P = \{q^0\} \cup \bigcup_{q \in Q} \{(q, \phi) \mid \text{there is a transition } p \xrightarrow{<a, \phi>} q \text{ in } A\}$,
- $p^0 = q^0$,
- $(p, \phi_1) \xRightarrow{a} (q, \phi_2)$ 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 $(M^c = (Q^c \xrightarrow{c} q^{c0}), F^c)$ over Σ_i^c , where

- $Q^c = Q$,
- $q^{c0} = p^0$,
- $(p, \phi_1) \xrightarrow{c} (q, \phi_2)$ iff $(p, \phi_1) \xrightarrow{a} (q, \phi_2)$ and $\phi = f((q, \phi_2)) = \phi_2$,
- Finally, $F^c = G$.

Since, by definition, $L_m(M, G) = L(M^c, F^c)$, it suffices to show that $L(M^c, F^c) = L(A, F)$.

We note that M'_i simulates A via Θ where $\Theta((p, \phi)) = p$ and $\Theta(q^0) = q^0$ and $F' = \{(p, \phi) \mid \Theta((p, \phi)) = p \in F\}$. Then by simulation theorem, we get the language equality and hence the proof. ■

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

Proposition 5.45 *Suppose, for all $i \in Loc$, $L_i = L_m((M_i, f_i), F_i)$. Take an AC-system $\tilde{M} = (M_1, M_2, \dots, M_n, \langle f_1, \dots, f_n \rangle, \Pi_{i \in Loc} F_i)$ with the initial state (q_1^0, \dots, q_n^0) . Let ξ_0 denote $env(q_1^0, \dots, q_n^0)$. Then $L(\tilde{M}) = (\parallel L_i)_{\xi_0}$.*

Proof: (\supseteq ;) Let $x \in L(\tilde{M})$. Then there is a final state $(q_1^f, \dots, q_n^f) \in \Pi_{i \in Loc} F_i$ such that $(q_1^0, \dots, q_n^0) \xrightarrow{x} (q_1^f, \dots, q_n^f)$. By Proposition 5.41, there exist $x_i \in \Sigma_i^{c*}$ such that $q_i^0 \xrightarrow{x_i}^c q_i^f$ and x is generated by the tuple (x_1, x_2, \dots, x_n) under $\xi_0 = env(q_1^0, \dots, q_n^0)$. Since each q_i^f is in F_i , each x_i is in L_i . Hence by definition of shuffle, $x \in (\parallel L_i)_{\xi_0}$. Therefore, $L(\tilde{M}) \subseteq (\parallel L_i)_{\xi_0}$.

(\supseteq ;) Let $x \in (\parallel L_i)_{\xi_0}$. 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, \dots, 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 $q_i^0 \xrightarrow{x_i} q_i^f$. By Proposition 5.43 we have $(q_1^0, \dots, q_n^0) \xrightarrow{x} (f_1, \dots, f_n) \in \Pi_{i \in Loc} F_i$. Hence $x \in L(\tilde{M})$. Therefore, $(\| L_i)_{\xi_0} \subseteq L(\tilde{M})$. ■

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

Theorem 5.46 $\mathcal{L}(AC\text{-}Shuffle_{\tilde{\Sigma}}) = Reg_{\Sigma} = \mathcal{L}(ACS_{\tilde{\Sigma}})$.

Proof: Consider $L \in \mathcal{L}(ACS_{\tilde{\Sigma}})$. Then $L = L(\tilde{M})$ for some ACS $\tilde{M} = (M_1, M_2, \dots, M_n, < f_1, \dots, f_n >, F)$. One can then write L as $L = \bigcup_{q^f \in F} \mathcal{L}(\tilde{M}_{q^f})$ where $\tilde{M}_{q^f} = (M_1, \dots, M_n, < f_1, \dots, f_n >, \{q^f\})$. Let $q^f = (q_1^f, q_2^f, \dots, q_n^f)$ 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 $L(\tilde{M}_{q^f}) = (\| L_i)_{\xi_0}$. This places $L(\tilde{M}_{q^f})$ in $\mathcal{L}(AC\text{-}Shuffle_{\tilde{\Sigma}})$. Then, since $\mathcal{L}(AC\text{-}Shuffle_{\tilde{\Sigma}})$ is closed under union, L also is placed in it.

Let $L = (\| L_i)_{\xi_0}$. We show that $L \in Reg_{\Sigma}$. Since L_i 's are regular over Σ_i^c , by Proposition 5.44, there are AC-automata $((M_i, f_i), F_i)$ such that $L_i = L_m((M_i, f_i), F_i)$. By Proposition 5.45, the ACS $M = (M_1, \dots, M_n, < f_1, \dots, f_n >, \Pi_{i \in Loc} F_i)$ accepts L . Since M is an FA over Σ , $L \in Reg_{\Sigma}$. Since any language in $AC\text{-}shuffle_{\tilde{\Sigma}}$ is either an AC-shuffle or a union of AC-shuffle languages, $\mathcal{L}(AC\text{-}shuffle_{\tilde{\Sigma}}) \subseteq Reg_{\Sigma}$. ■

5.4.5 A syntax for ACS's and a Kleene theorem

We now consider the question of syntax for languages in $\mathcal{L}(ACS_{\tilde{\Sigma}})$. The syntax is given in two layers, one for 'local' expressions and another for parallel composition. Fix a distributed alphabet $\tilde{\Sigma}$, a commit alphabet $\tilde{\mathcal{C}}$, and the associated extended alphabet.

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

The semantics of these expressions is given as follows: for each $i \in Loc$, we have a map $\llbracket \cdot \rrbracket_i : ACREG_i \rightarrow 2^{\Sigma_i^{c*}}$, and globally a map $\llbracket \cdot \rrbracket : ACREG \rightarrow 2^{\Sigma^*}$. These maps are defined

by structural induction:

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

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, $\mathcal{L}(ACREG)_{\tilde{\Sigma}} = \{L \subseteq \Sigma^* \mid \text{for some commit alphabet } \bar{C}, \text{ there is an } R \in ACREG \text{ over } \tilde{\Sigma}^c \text{ such that } L = [R]\}$.

Then, from the semantics and Theorem 5.46, we get the following characterization.

Theorem 5.47 $\mathcal{L}(ACS)_{\tilde{\Sigma}} = \mathcal{L}(ACREG)_{\tilde{\Sigma}} = Reg_{\Sigma}$.

5.5 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):

$$FACREG_i ::= \emptyset \mid a \in \Sigma_i \mid \phi \in \Phi \mid p + q \mid p; q \mid p^*$$

$$FACREG ::= r_1 \parallel r_2 \parallel \dots \parallel r_n, r_i \in FACREG_i \mid R_1 + R_2, R_i \in FACREG$$

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.

Natural semantics of $FACREG_i$ (call the semantic function $\llbracket \cdot \rrbracket_i$) expressions give regular languages over $\Sigma \cup \Phi$. We translate these languages to languages over Σ_i^c in some 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 end. 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, $\perp \cdot \langle a_1, \phi_2 \rangle \cdot \langle a_2, \perp \rangle \cdot \langle a_3\phi_3 \rangle \cdot \langle a_4\perp \rangle$.

For some other examples see that,

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

It should be clear that this translation is actually a function, call it h such that $h : (\Sigma_i \cup \Phi)^* \rightarrow \Phi \cdot \Sigma_i^c^*$. 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 5.48 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: $(L_1 \parallel \cdots \parallel L_n)_{(\phi_1, \dots, \phi_n)}$ which is the AC-shuffle of the L_i 's with the initial environment (ϕ_1, \dots, ϕ_n) .

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

Definition 5.49 Let $R = (r_1 \parallel_F \cdots \parallel_F r_n) \in FACREG$. Let also $\langle r_i \rangle_i = \bigcup_{k \in \Gamma_i} \phi_{ik} L_{ik}$. Then,

$$\langle R \rangle \stackrel{\text{def}}{=} \bigcup_{j_i \in \Gamma_i, i \in Loc} (L_{1j_1} \parallel \cdots \parallel L_{nj_n})_{(\phi_{1j_1}, \dots, \phi_{nj_n})}.$$

By the very definition, $\mathcal{L}(FACREG_{\tilde{\Sigma}}) \subseteq \mathcal{L}(ACREG_{\tilde{\Sigma}})$.

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 $d : \Sigma_i^c \rightarrow (\Sigma_i \cdot \Phi)$ defined as $d(\langle a, \phi \rangle) = a\phi$ 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 5.50 $\mathcal{L}(ACREG_{\tilde{\Sigma}}) = \mathcal{L}(FACREG_{\tilde{\Sigma}})$.

5.6 ACS's for ω -regular languages

Automata

Finite behaviour of ACS's is given via compatible product with a set of global final states. A natural extension of this notion to capture infinite behaviour of ACS's is to have a Muller acceptance condition with the compatible product.

Fix a distributed alphabet $\tilde{\Sigma} = (\Sigma_1, \dots, \Sigma_n)$ and a commit alphabet $\tilde{\mathcal{C}} = \langle (\mathcal{C}_1, \preceq_1), \dots, (\mathcal{C}_n, \preceq_n) \rangle$.

Definition 5.51 *An ACS with Muller condition (ACS_M) over $(\tilde{\Sigma}, \tilde{\mathcal{C}})$ is given by a tuple $\tilde{M} = (M_1, \dots, M_n, \langle f_1, f_2, \dots, f_n \rangle, \mathcal{T})$, where*

- *for each $i \in Loc$, (M_i, f_i) is an AC-TS over $(\Sigma_i, \tilde{\mathcal{C}})$, and*
- *$\mathcal{T} \subseteq 2^{\tilde{Q}}$, where \tilde{Q} is the set of compatible global states of \tilde{M} .*

An infinite string $x \in \Sigma^\omega$ is accepted by \tilde{M} if there is an infinite run ρ on x in the compatible product of \tilde{M} and a set $G \in \mathcal{T}$ such that $\text{Inf}(\rho) = G$.

The class of languages accepted by these ACS's is denoted as $\mathcal{L}(\omega\text{ACS}_{\tilde{\Sigma}})$.

Compatible shuffle of ω -languages

The definitions of compatible shuffle of finite strings over Σ_i^c (Section 5.4) directly generalizes to those of infinite strings over Σ_i^c without any change whatsoever. Hence, we use the same notation $\hat{\parallel}$ to denote compatible shuffle of ω -languages over Σ_i^c .

Then $\mathcal{L}(\omega\text{-AC-shuffle})_{\tilde{\Sigma}}$ denotes the least class that includes the set $\{L \subseteq \Sigma^\omega \mid \text{for some commit alphabet } \tilde{\mathcal{C}}, \text{ there exist } \omega\text{-regular languages } L_i \subseteq \Sigma_i^{c\omega} \text{ such that } L = L_1 \hat{\parallel} \dots \hat{\parallel} L_n\}$ and is closed under union and complementation.

Syntax

The syntax is also a smooth generalization, mirroring the way we construct *ACREG* expressions. We now have three layers:

$$\begin{aligned}
ACREG_i &:= \epsilon \mid \langle a, \phi \rangle \in \Sigma^c_i \\
&\mid p + q \mid p; q \mid p^* \quad p, q \in ACREG_i \\
\omega ACREG_i &::= r \cdot s^\omega \quad r, s \in ACREG_i \\
&\mid R_1 + R_2 \quad R_1, R_2 \in \omega ACREG_i \\
\omega ACREG &::= (R_1 \parallel \dots \parallel R_n)_\xi \quad R_i \in \omega ACREG_i \text{ and } \xi \in Xi \\
&\mid \neg X \mid X_1 + X_2 \quad X, X_i \in \omega ACREG
\end{aligned}$$

Thus, at the local level, we have ω -regular languages over extended alphabets and at the global level we have parallel composition and their boolean combination. The semantics for $\omega ACREG$ expressions is given by compatible shuffle of ω -languages. The class of languages accepted by $ACREG$ expressions then is denoted as $\mathcal{L}(\omega ACREG)$.

Discussion

The significant departure from the earlier syntax is that now we have a global complementation operator. In the finite case, we saw that the global union was only *syntactic sugar* and could be eliminated giving us only parallel composition at the global level. Even the global complementation in that case could be transformed into parallel composition. But in the infinite case, such a transformation seems quite difficult. Thus, while the finite case yielded to a completely local presentation (syntactically), this does not seem to be so in the infinite case.

Results

Since the definitions of automata, syntax and compatible shuffle generalize straightforwardly from the finite case, we now show that the results asserting their equivalence (in terms of language acceptance) also generalize smoothly. We summarize the results in the following, which essentially gives the Kleene theorem for ω -Regular languages.

Theorem 5.52 $\omega Reg_\Sigma = \mathcal{L}(\omega ACS_{\bar{\Sigma}}) = \mathcal{L}(\omega\text{-}AC\text{-}shuffle)_{\bar{\Sigma}} = \mathcal{L}(\omega ACREG_{\bar{\Sigma}})$.

5.6.1 Proof of the theorem

The proof of the theorem can be done in three stages as follows:

1. $\omega Reg_{\Sigma} = \mathcal{L}(\omega ACS_{\tilde{\Sigma}})$.
2. $\omega Reg_{\Sigma} = \mathcal{L}(\omega\text{-AC-shuffle})_{\tilde{\Sigma}}$.
3. $\mathcal{L}(\omega\text{-AC-shuffle})_{\tilde{\Sigma}} = \mathcal{L}(\omega ACREG_{\tilde{\Sigma}})$.

Lemma 5.53 $\omega Reg_{\Sigma} = \mathcal{L}(\omega ACS_{\tilde{\Sigma}})$.

Proof: The inclusion $\mathcal{L}(\omega ACS_{\tilde{\Sigma}}) \subseteq \omega Reg_{\Sigma}$ follows easily because, if L is accepted by \tilde{M} where $\tilde{M} \in ACS_M$, then L is actually accepted by the compatible product of \tilde{M} (which is an FSA over Σ) with a Muller condition. Hence, $L \in \omega Reg_{\Sigma}$.

For the other inclusion $\omega Reg_{\Sigma} \subseteq \mathcal{L}(\omega ACS_{\tilde{\Sigma}})$, let $L \in \omega Reg_{\Sigma}$. We know that there is a deterministic Muller automaton (N, \mathcal{T}) such that $L = L(N, \mathcal{T})$. We need to show that there is a commit alphabet \mathcal{C} and an ωACS on $(\tilde{\Sigma}, \tilde{\mathcal{C}})$ such that it accepts L .

For this we use proof of Theorem 5.6 in Chapter 5 which essentially shows that there is an ACS $\tilde{M} = (M_1, \dots, M_n, \langle f_1, f_2, \dots, f_n \rangle)$ such that the product \widehat{M} simulates N via some Θ . Let \hat{Q} be the set of all compatible global states of \widehat{M} .

Using Θ , define $\mathcal{T}' = \{F' \subseteq \hat{Q} \mid \text{there exists } F \in \mathcal{T} \text{ such that } \Theta(F') = F\}$. Then by simulation theorem, $L(\widehat{M}, \mathcal{T}') = L(N, \mathcal{T}) = L$. Hence, the ACS $\tilde{M} = (M_1, \dots, M_n, \langle f_1, f_2, \dots, f_n \rangle, \mathcal{T}')$ accepts L . This proves the inclusion and hence the lemma is proved.

■

Lemma 5.54 $\omega\text{-AC-shuffle} \subseteq \omega Reg_{\Sigma}$.

Proof: Let $L \in \omega\text{-AC-shuffle}$ and let $L = (L_1 \parallel \dots \parallel L_n)_{\xi}$, where L_i are regular over Σ_i^c . It suffices to show that $L \in \omega Reg_{\Sigma}$ since ωReg_{Σ} is closed under boolean operations.

In the proof of Proposition 5.44 in Chapter 5, we showed that for a given $L \subseteq \Sigma_i^{c*}$, there is an AC-automaton (M, f) over Σ_i such that $L = L_m((M, f), F)$ for some set F of final states.

We observe that the construction is, in fact, independent of acceptance conditions. Hence, following that proof technique, one can show here that for all $L_i \in \Sigma_i^{c\omega}$, there is an AC-automaton with Büchi condition $((M_i, f_i), \mathcal{B}_i)$ over Σ_i such that $L_i = L_m((M_i, f_i), \mathcal{B}_i)$.

For the proof of the lemma it is advantageous to have a slight variation of ω ACS's. This time we assign multiple Büchi conditions to the product. As observed in Section 2.5.1 of Chapter 2, automata with multiple Büchi conditions still accept only ω regular languages. We recall the definition as it applies to the compatible product of an ACS.

Definition 5.55 *An ACS with multiple Büchi condition over $(\tilde{\Sigma}, \tilde{\mathcal{C}})$ is given by a tuple $\tilde{M} = (M_1, \dots, M_n, \langle f_1, f_2, \dots, f_n \rangle, \{\mathcal{B}_1, \dots, \mathcal{B}_k\})$, where*

1. *for each $i \in Loc$, (M_i, f_i) is an AC-TS over $(\Sigma_i, \tilde{\mathcal{C}})$, and*
2. *$\mathcal{B}_i \subseteq \hat{Q}$, $i \in \{1, \dots, k\}$, where \hat{Q} is the set of compatible global states of \tilde{M} .*

An infinite string $x \in \Sigma^\omega$ is accepted by \tilde{M} if there is an infinite run ρ on x such that for all $i \in \{1, \dots, k\}$, $Inf(\rho) \cap \mathcal{B}_i \neq \emptyset$.

Claim: *Let $L_i = L_m(M_i, \mathcal{B}_i)$, $i \in \{1, \dots, n\}$ where (M_i, \mathcal{B}_i) are Büchi automata over alphabets Σ_i^c . Then, there is an ACS \tilde{M} with multiple Büchi conditions $\{\mathcal{G}_1, \dots, \mathcal{G}_n\}$ such that $(\bigparallel_i L_i)_{\xi_0} = L(\tilde{M}, \{\mathcal{G}_1, \dots, \mathcal{G}_n\}) = L$, where $\xi_0 = env(q_1^0, \dots, q_n^0)$.*

Assuming the claim, we immediately get that $L \in \omega Reg_\Sigma$ and hence the lemma is proved, pending the claim.

Proof of claim: Define $M = (M_1, \dots, M_n, \langle f_1, \dots, f_n \rangle, \{\mathcal{G}_1, \dots, \mathcal{G}_n\})$ where the global Büchi conditions are $\mathcal{G}_i = \{(p_1, \dots, p_n) \in \hat{Q} \mid p_i \in \mathcal{B}_i\}$. We show that $L = L(\tilde{M}, \{\mathcal{G}_1, \dots, \mathcal{G}_n\})$.

(\supseteq .) Let $x \in L(M)$. Then, there is a ρ on x in the compatible product \tilde{M} such that

$$\text{for all } i \in Loc, \text{ } inf(\rho) \cap \mathcal{G}_i \neq \emptyset. \quad (8)$$

Recall how we assigned strings over Σ^ω with runs of a product system in Section 5.4.3. Let a run $\rho = (q_1^0, \dots, q_n^0) \xrightarrow{a_1} (q_1^1, \dots, q_n^1) \dots$ on some $x \in \Sigma^\omega$ in \tilde{M} . Define $c(\rho) = \langle a_1, \xi_1 \rangle \dots \in \Sigma^{\omega}$ by: for $1 \leq l \leq k$, $\xi_l = env(q_1^l, \dots, q_n^l)$.

Let $x_i = c(\rho) \upharpoonright i$, for all $i \in Loc$. Then, each $x_i \in \Sigma_i^{\omega}$. It is obvious to see that $\rho \upharpoonright i$ is a path on x_i in M_i . With (8) it implies that $inf(\rho_i) \cap \mathcal{B}_i \neq \emptyset$. Therefore, for all $i \in Loc$, $x_i \in L_i$.

Similar to the proof of Proposition 5.41, we can show that x is generated by (x_1, \dots, x_n) under ξ_0 . Hence, by definition of shuffle, $x \in (\hat{\parallel} {}_i L_i)_{\xi_0}$.

(\subseteq ;) Let $x \in (\hat{\parallel} {}_i L_i)_{\xi_0}$. By definition, for every $i \in Loc$, there exist $x_i \in \Sigma_i^{c\omega}$ such that $x_i \in L_i$ and x is generated by (x_1, \dots, x_n) under ξ_0 . Since $x_i \in L_i$, for all $i \in Loc$, there is some path ρ_i for x_i in M_i such that $\inf(\rho_i) \cap \mathcal{B}_i \neq \emptyset$.

Then, one can construct a path ρ for x inductively from the set of paths ρ_1, \dots, ρ_n in \widehat{M} such that $\rho[i] = \rho_i$. This construction is the same as in Proposition 5.43; it ensures that if one starts from a compatible initial state, the very choice of x_i 's ensure that one steps through only compatible states and never gets stuck. But this implies that hence $\inf(\rho) \cap \mathcal{G}_i \neq \emptyset$ for all $i \in Loc$. Hence $x \in L(M)$. Thus we prove the claim and the lemma.

■

Lemma 5.56 $\omega Reg_{\Sigma} \subseteq \omega\text{-}AC\text{-}shuffle$.

Proof: Let $L \in \omega Reg_{\Sigma}$. By Lemma 5.53, there is an ACS M with global Muller conditions such that $L = L(M, \mathcal{T})$. Let the global Muller condition be $\mathcal{T} = \langle F_1, \dots, F_k \rangle$ where $F_i \subseteq \widehat{Q}$. Then, one can rewrite the language accepted by M as a boolean combination of $wACS$'s each with a Büchi condition. Further more, all the Büchi conditions are of cardinality one.

Proposition 5.57 (McNaughton) *Let (M, \mathcal{T}) be an FSA with Muller condition $\mathcal{T} = \{F_1, \dots, F_k\}$. Then,*

$$L(M, \mathcal{T}) = \bigcup_{j \in \{1 \dots k\}} (\cap_{\bar{q} \in F_j} L(M, \{\bar{q}\}) \cap \neg \cup_{\bar{q} \notin F_j} L(M, \{\bar{q}\})).$$

In the light of the above proposition, in order to express the given languages as (boolean combination of) compatible shuffle languages, it suffices to show that any language accepted by an ωACS with a Büchi condition of cardinality one can be expressed as a compatible shuffle of infinite languages over Σ_i^c .

For this we introduce the notion of ACS's with local Büchi conditions.

An ACS with local Büchi condition ($ACSLB$) over $(\tilde{\Sigma}, \tilde{\mathcal{C}})$ is given by a tuple $\tilde{M} = (M_1, \dots, M_n, \langle f_1, f_2, \dots, f_n \rangle, \langle B_1, \dots, B_n \rangle)$, where

1. for all $i \in Loc$, (M_i, f_i) is an AC-TS over $(\Sigma_i, \tilde{\mathcal{C}})$, and
2. for all $i \in Loc$, $B_i \subseteq Q_i$.

An infinite string $x \in \Sigma^\omega$ is accepted by \tilde{M} if there is an infinite run ρ on x such that for all $i \in Loc$, $Inf_i(\rho) \cap B_i \neq \emptyset$, where $Inf_i(\rho) = \{q \in Q_i \mid \exists^\infty j : \rho(j) = s \text{ and } s[i] = q\}$.

Claim: Let L be accepted by an ω ACS with $\{(s_1, \dots, s_n)\}$ as the Büchi condition. Then, there is an ω ACS with local Büchi condition $\tilde{M} = (M_1, \dots, M_n, \langle f_1, f_2, \dots, f_n \rangle, \langle B_1, \dots, B_n \rangle)$, such that $L = L(\tilde{M})$.

Assuming the claim, from the proposition above, the lemma follows.

Proof of claim: To simplify the presentation, let $n = 2$. Let $O = (M, N, \langle f, g \rangle, \{(s_M, s_N)\})$, where

- $M = (P, \longrightarrow, p^0, f)$
- $N = (Q, \longrightarrow, q^0, g)$.

Since we will not be using the transition relation very much, we use the same arrow to denote the transitions of both M and N .

Construct $M' = (Q', \longrightarrow', p^{0'}, f')$ as follows. (Construction of N' from N is similar).

- $Q' = (Q \cup \{sp_M\}) \times \{0, 1\}$ where $(sp_M, 0)$ and $(sp_M, 1)$ are "special" local states.
-

$$\begin{aligned}
 \longrightarrow' &= \{((p, 0), a, (q, 0)) \text{ and } ((p, 1), a, (q, 1)) \mid p \xrightarrow{a} q\} \\
 &\cup \{((p, 0), a, (sp_M, 0)) \mid p \xrightarrow{a} s_M\} \\
 &\cup \{((sp_M, 0), a, (q, 1)) \mid s_M \xrightarrow{a} q\} \\
 &\cup \{((p, 1), a, (sp_M, 1)) \mid p \xrightarrow{a} s_M\} \\
 &\cup \{((sp_M, 1), a, (q, 0)) \mid s_M \xrightarrow{a} q\}
 \end{aligned}$$

- $p^{0'} = (p^0, 0)$

The commitment alphabet \mathcal{C}' is defined as: for all $i \in Loc$, $\mathcal{C}'_i = \mathcal{C}_i \times \{0, 1, \top_1, \top_2\}$.

The order among elements of \mathcal{C}'_i is defined as: $(\nu_1, bit_1) \preceq'_i (\nu_2, bit_2)$ iff $\nu_1 \preceq_i \nu_2$ and either $(bit_1 = bit_2)$ or $(bit_1 \in \{0, 1\} \text{ and } bit_2 \in \{\top_1, \top_2\})$.

The commitment maps are defined for the local states as follows.

For $bit \in \{0, 1\}$ and $j \in \{1, 2\}$,

$$f'(p, bit)(j) = \begin{cases} (f(p)(j), bit) & \text{when } p \neq sp_M. \\ (f(sp_M)(j), \top_{bit}) & \text{when } p = sp_M. \end{cases}$$

Construction of M' from M is shown pictorially in Fig. 5.7.

Let $O' = (M', N', < f', g' >, < \{(sp_M, 1)\}, \{(sp_N, 1)\} >)$ be an ACS with local Büchi condition.

By virtue of the construction, the product has some special properties. We observe these in the following. (When $bit = 0$ (resp. 1), $\overline{bit} = 1$ (resp. 0)).

Consider any global state $((p_1, bit_1), (p_2, bit_2))$. If it is compatible, then it is one of the following forms:

1. $p_1 \neq sp_M$, $p_2 \neq sp_N$ and bit_i 's are either all 0 or all 1. This is because in this case $f'(p_1, bit)$ is $(f(p_1), bit)$ and hence (p_1, bit) is incompatible with any other state (p_2, \overline{bit}) .
2. $p_1 = sp_M$, $p_2 = sp_N$ and bit_i 's are either all 0 or all 1. This follows directly from the new assumption map f' .

We show in the following that $L(O') = L(O)$. This is done by showing that the product of O' denoted as $\widehat{O'}$ simulates the product \widehat{O} of O .

Define $\Theta_M : P' \rightarrow P$ as :

$$\Theta_M(p, i) = \begin{cases} p & \text{if } p \neq sp_M \\ sp_M & \text{if } p = sp_M \end{cases}$$

Similarly define $\Theta_N : Q' \rightarrow Q$. Then define $\Theta : \widehat{O'} \rightarrow \widehat{O}$ as : $\Theta(p_1, p_2) = (\Theta_M(p_1), \Theta_N(p_2))$.

For $i \in \{0, 1\}$, denote by SP^i the state $((sp_M, i), (sp_N, i))$.

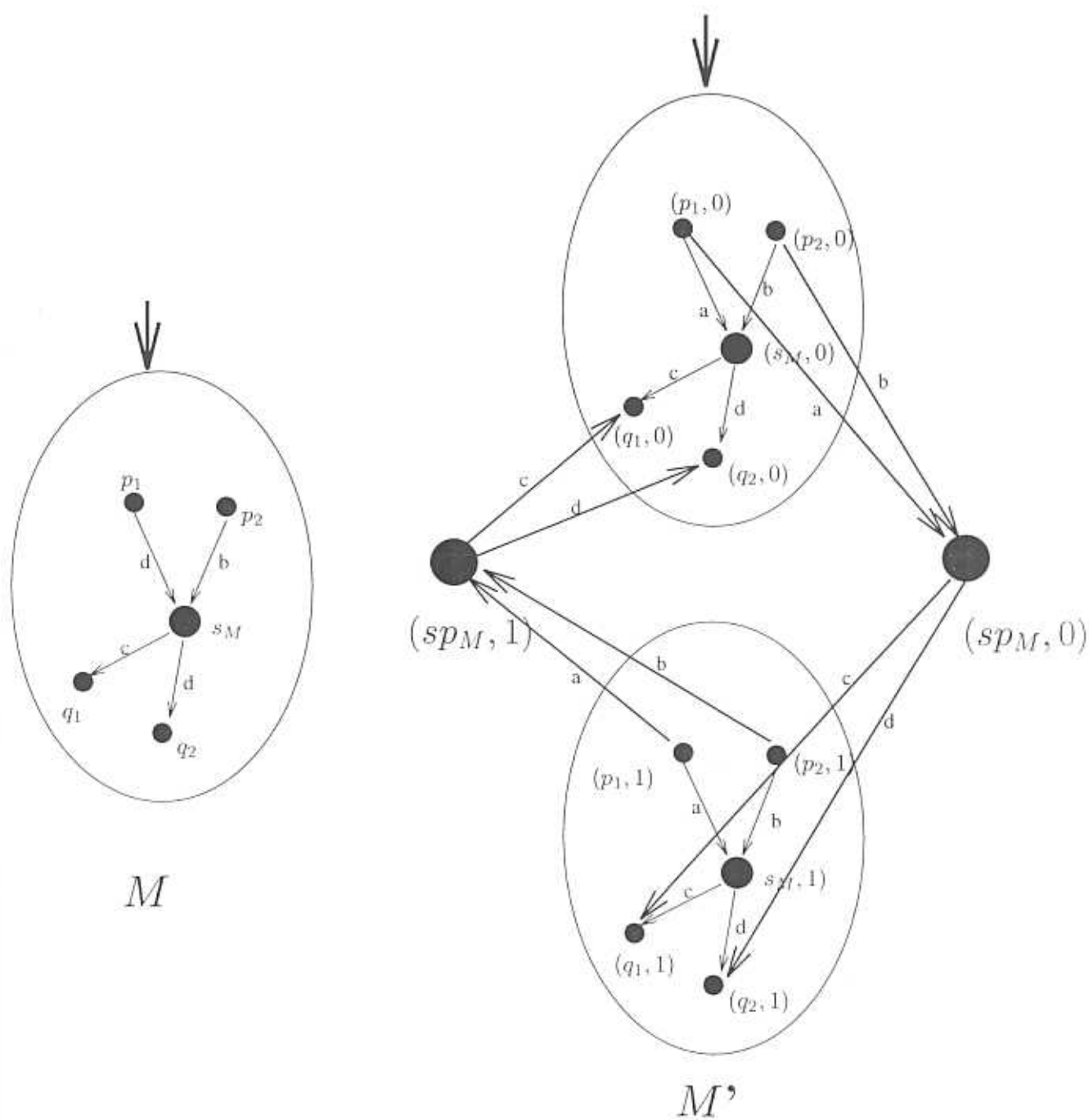


Figure 5.7: Constructing M' from M .

Let $x \in L(O)$. Then there is an infinite run ρ on x in \widehat{O} such that (s_M, s_N) occurs infinitely often. By simulation theorem, there is a run ρ' on x in \widehat{O}' such that there exists $(s_1, s_2) \in \widehat{O}'$ with $\Theta(s_1, s_2) = (s_M, s_N)$. In which case, (s_1, s_2) can only be SP^0 or SP^1 . Since the only paths from SP^0 to SP^0 must pass through SP^1 , both these states occur infinitely often in ρ' . Hence, $\text{inf}_M(\rho') = \{(sp_M, 0), (sp_M, 1)\}$. Similarly, $\text{inf}_N(\rho') = \{(sp_N, 0), (sp_N, 1)\}$, and hence $x \in L(O')$.

Let $x \in L(O')$. Then there is a run ρ' in \widehat{O}' such that $\text{inf}_M(\rho') \cap \{(sp_M, 1)\} \neq \emptyset$. This implies $(sp_M, 1)$ occurs infinitely often in ρ . We show that this implies both SP^0 and SP^1 occur infinitely often on ρ' . By simulation theorem, there is an infinite run ρ on x in \widehat{O} such that (s_M, s_N) occurs infinitely often on $\Theta(\rho')$ which implies $x \in L(O)$.

Notice that whenever $(sp_M, 0)$ (resp. $(sp_M, 1)$) occurs, at some point in the future SP^0 (resp. SP^1) occurs. This ensures that the local automata necessarily synchronize at these states. Informally, since these states actually simulate the given (single) Büchi condition, we are ensured that even in the distributed nature of computation, the product visits the Büchi condition infinitely often. Now, we give a proof of this claim in the following.

Suppose, $\rho'(t)[1] = (sp_M, 0)$. Then $\rho'(t)[2]$ is either $(sp_N, 0)$ in which case we have SP^0 , or $(q_N, 0)$ for some $q_N \in Q$. At this point M' can not move to the second copy because then it will be in a state $(p_M, 1)$ and hence the global state will be incompatible. Neither can M' move into the first copy because by construction, there is no such transition. Hence, M' waits in the state $(sp_M, 0)$. Eventually, N' must come to $(sp_N, 0)$ since it has to visit the state $(sp_N, 1)$ infinitely often and the only way to $(sp_N, 1)$ from the first copy is through $(sp_N, 0)$. Hence, eventually, SP^0 occurs.

By the argument above and the fact that $(sp_M, 0)$ (resp. $(sp_M, 1)$) occur infinitely often, we get that both SP^0 and SP^1 occur infinitely often. This completes the proof of the claim and the lemma. ■

Lemma 5.58 $\mathcal{L}(\omega\text{-}AC\text{-}shuffle)_{\widehat{\Sigma}} = \mathcal{L}(\omega\text{ACREG}_{\widehat{\Sigma}})$.

Proof: The right-to-left inclusion is trivial because the semantics of ωACREG expressions is given in terms of $\omega\text{-}AC\text{-}shuffle$ and this class of languages is closed under boolean operations,

by definition.

For the left-to-right inclusion, observe that by Lemma 5.54, $\omega\text{-AC-shuffle} \subseteq \omega\text{Reg}_\Sigma$. By Lemma 5.53, for every $L \in \omega\text{Reg}_\Sigma$, we can construct an ωACS with Muller condition \widetilde{M} accepting L . Now, as in the proof of Lemma 5.56, L can be expressed as a boolean combination of languages in $\mathcal{L}(\omega\text{-AC-shuffle})_{\widetilde{\Sigma}}$. Since we have a global complementation operator in the syntax, it suffices to find syntactic expressions for languages in $\mathcal{L}(\omega\text{-AC-shuffle})_{\widetilde{\Sigma}}$. The proof of the same lemma shows that each of these languages can be accepted by an ωACS \widetilde{M}' with local Büchi conditions. Constructing an ωACREG expression for \widetilde{M}' is now straightforward. (This is where local Büchi conditions come of use). ■

/

Assumption-compatible systems with restrictions

In Chapters 2 and 3, we have described local presentations for synchronized shuffle of regular languages (*RSL*) and regular consistent languages (*RCL*) respectively. In Chapter 5 we saw that for any given distributed alphabet $\tilde{\Sigma}$, assumption-compatible systems can characterize the class of all regular languages over Σ . The natural question to ask is: whether both *RSL* and *RCL* can be locally presented via subclasses of ACS's. In the following we show that indeed this is possible.

6.1 ACS's for regular shuffle languages

We know that $RSL_{\tilde{\Sigma}}$ is characterized by the class of product systems $PS_{\tilde{\Sigma}}$. Recall the definition of product systems.

Definition 6.1 *A Product System over $\tilde{\Sigma}$ is a tuple*

$$\tilde{M} = (M_1, \dots, M_n, < F_1, \dots, F_n >), \text{ where}$$

1. for all $i \in Loc$, (M_i, F_i) is an FA over Σ_i , and
2. the product automaton of \tilde{M} is (\tilde{M}, F) where \tilde{M} is the complete product TS of \tilde{M} and $F = \prod_{i=1}^n F_i$.

The global states of product of \tilde{M} are all possible tuples of local states and the global transition relation is derived completely from the local transitions.

Since non-trivial use of compatibility is to filter out global states and thereby control global behaviour, for the case in hand, we really do not need to impose the notion. However, if one insists, one may have a notion of compatibility which allows *all* the global states to be compatible. This, perhaps, can be done by various means. We describe two ways in which it can be done.

6.1.1 ACS's with null assumption

Given $\tilde{\Sigma}$, take a commitment alphabet \mathcal{C} where for all $i \in Loc$, $\mathcal{C}_i = \{\perp\}$. Take the class of ACS's on $(\tilde{\Sigma}, \tilde{\mathcal{C}})$. Call the class $NACS_{\tilde{\Sigma}}$.

Then, if $\tilde{M} = (M_1, \dots, M_n, F)$ is an $NACS$, then for all $i, j \in Loc$ and $q \in Q_i$, $f_i(q)(j) = \perp$. Thus the very nature of the commitment alphabet ensures that local automata do not make any non-trivial assumptions about other's states. One immediately sees that this allows all global states to be compatible. Hence, we get the following result.

Theorem 6.2 $\mathcal{L}(NACS_{\tilde{\Sigma}}) = RSL_{\tilde{\Sigma}}$.

6.1.2 ACS's with static assumption

Consider the following condition which is seemingly stronger.

Definition 6.3 A $SACS$ $\tilde{M} = (M_1, \dots, M_n, F)$ over $(\tilde{\Sigma}, \tilde{\mathcal{C}})$ is an ACS, where for every $i \in Loc$, $p \xrightarrow{a} q$ implies for all $j \in Loc, j \neq i$, $f_i(p)(j) = f_i(q)(j)$.

In other words, in $SACS$'s the assumptions of an agent about others do not change during local transitions. This way, any agent can only assume a fixed set of states possible for another. Once we restrict our attention to only these states, all the global states generated are compatible and hence it is equivalent to that of having null assumptions on all local states. Hence we get the following theorem.

Theorem 6.4 $\mathcal{L}(SACS_{\tilde{\Sigma}}) = RSL_{\tilde{\Sigma}}$.

6.2 ACS's capturing regular consistent languages

The condition that we consider now is both on the commit alphabet and on the joint transitions in the product.

Definition 6.5 *An ACS is called synchronizing if the product satisfies the following assumption synchronization condition alongwith asynchrony.*

$$(p_1, p_2, \dots, p_n) \xrightarrow{a} (q_1, q_2, \dots, q_n) \text{ implies for all } i, j \in \text{loc}(a), f_i(q_i) = f_j(q_j).$$

Informally, immediately after an action has taken place, agents participating in the action have complete knowledge of the commitments of other participating agents and have the same assumption about the non-participating agents. This is very similar to the *perfect exchange* condition of view-based systems of Chapter 3 (except that in case of view-based systems, the local states of participating agents were the same after a transition.)

We denote this class of ACS's as *SyncACS's* and the languages accepted by them is denoted as $\mathcal{L}(\text{SyncACS})$. The asynchrony condition immediately gives us the easy inclusion $\mathcal{L}(\text{SyncACS})_{\tilde{\Sigma}} \subseteq \text{RCL}_{\tilde{\Sigma}}$. To show the other inclusion, take a deterministic Zielonka automaton \mathcal{A} accepting $L \in \text{RCL}_{\tilde{\Sigma}}$. Recall that, for all $x \in \Sigma^*$,

$$\text{event}(x) = \begin{cases} ((\epsilon)_{\mathcal{A}}, \epsilon, (\epsilon)_{\mathcal{A}}) & \text{if } x = \epsilon, \\ ((y)_{\mathcal{A}}, a, (x)_{\mathcal{A}}) & \text{if } x = ya. \end{cases}$$

The commit alphabet \mathcal{C} is taken as follows: $\mathcal{C}_i = \{\text{event}(x \downarrow i) \mid x \in \Sigma^*\}$. The ordering \preceq_i is given by: $\text{event}(x) \preceq_i \text{event}(y)$ iff there exist $u, v \in \Sigma^*$, $u \ll v$ and $\text{event}(x) = \text{event}(u)$ and $\text{event}(y) = \text{event}(v)$.

By this definition, $\text{event}(x \downarrow i) \preceq_i \text{event}(xa \downarrow i)$ for all $i \in \text{Loc}$.

Let $\gamma(x) \stackrel{\text{def}}{=} (\text{event}(x \downarrow 1), \dots, \text{event}(x \downarrow n))$.

Define the SyncACS $\tilde{M} = (M_1, \dots, M_n, F)$ as follows. For all $i \in \text{Loc}$, $M_i = (Q_i, \longrightarrow_i, q_i^0, f_i)$ where,

- $Q_i = \{\gamma(x \downarrow i) \mid x \in \Sigma^*\}$.
- $q_i^0 = \gamma(\epsilon)$.

- $p \xrightarrow{a}_i q$ if there is some $u \in \Sigma^*$ such that $p = \gamma(u \downarrow i)$ and $q = \gamma(ua)$.
- $f_i(\gamma(x)) = \gamma(x)$.

Set $F = \{(\gamma(x \downarrow 1), \dots, \gamma(x \downarrow n)) \mid x \in L(\mathcal{A})\}$.

Notice that except for the assumption maps, \widetilde{M} is defined exactly the same way as the view-based system in Chapter 3. Notice that the assumption maps and assumption synchronization condition ensure the perfect exchange of transitions, which implies that $L(\widetilde{M}) \subseteq L(\mathcal{A})$.

For the other inclusion $L(\mathcal{A}) \subseteq L(\widetilde{M})$, it suffices to check that the assumption maps ensure the following.

1. For all $x \in \Sigma^*$, $(\gamma(x \downarrow 1), \dots, \gamma(x \downarrow n))$ is compatible.
2. $(\gamma(x \downarrow 1), \dots, \gamma(x \downarrow n)) \xRightarrow{a} (\gamma(xa \downarrow 1), \dots, \gamma(xa \downarrow n))$.

But these are easily proved from the definition of γ and some simple properties of views. Thus, we get the required characterization of RCL 's via $SyncACS$'s.

Theorem 6.6 $\mathcal{L}(SyncACS)_{\widetilde{\Sigma}} \subseteq RCL_{\widetilde{\Sigma}}$.

6.3 ACS's with a monotone condition

We see in the preceding sections that some familiar languages that we saw in earlier chapters could be captured by subclasses of ACS's. In the following, for any given $\widetilde{\Sigma}$, we give an example of a class of languages different from RSL 's and RCL 's that can be captured by suitable condition on ACS's. Perhaps neither this class of automata, nor the class of languages accepted, has any claim to intrinsic interest. They are presented here only to demonstrate the flexibility of the assumption-commitment framework in automata.

We now define a class of ACS's where the assumptions of local states never *decrease*.

Definition 6.7 A *MonACS* $\widetilde{M} = (M_1, \dots, M_n, F)$ is an ACS, where for every $i \in Loc$, $p \xrightarrow{a}_i q$ implies for all $j \in Loc$, $f_i(p)(j) \preceq_i f_i(q)(j)$.

A crucial consequence of the monotone condition is that along every path in M_i assumptions increase monotonically. Hence, for all $i \in Loc$, if $p \xrightarrow{a}_i q$ and $f_i(p)(j) \prec_i f_i(q)(j)$ (a strict increase) then there is no path from q to p in M_i .

In the following we show that *MonACS*'s characterize a class of languages which properly subsumes *RSL*'s but is *different* from the class of *RCL*'s.

6.3.1 Languages

Consider the least class of languages *BFCRSL* (boolean combination of finite concatenation of regular shuffle languages) such that

1. *BFCRSL* includes all regular shuffle languages, and
2. if L_1, L_2 are in *BFCRSL* then $L_1 \cdot L_2$ and $L_1 \cup L_2$ also are in *BFCRSL*.

From the definition, it trivially follows that $RSL_{\tilde{\Sigma}} \subseteq \mathcal{L}(BFCRSL_{\tilde{\Sigma}})$ and that the inclusion is strict.

Now, take the language $L = [ab][cd]$ on the distributed alphabet $\Sigma_1 = \{a, c\}$ and $\Sigma_2 = \{b, d\}$. Since $[ab]$ and $[cd]$ are both *RSL*'s over $\tilde{\Sigma}$, $L \in BFCRSL_{\tilde{\Sigma}}$. But clearly, it is not closed under \sim because, for example, while $abcd \in L$, $acbd \notin L$. Hence $L \notin RCL_{\tilde{\Sigma}}$. This shows that $BFCRSL_{\tilde{\Sigma}} \not\subseteq RCL_{\tilde{\Sigma}}$.

The other non-inclusion $RCL_{\tilde{\Sigma}} \not\subseteq BFCRSL_{\tilde{\Sigma}}$ also holds. But now the argument is slightly involved. We have seen in Section 2.3.6 that union of *RSL*'s do not give us *RCL*'s. Here we have to show that union of finite concatenations of *RSL*'s also is not sufficient to capture *RCL*'s. We take the same example of the language $L = ([ab]c + [aabb]c)^*$.

Let $L_k = \{x \in L \mid x \text{ has } k \text{ number of } c\text{'s}\}$. In Section 2.3.7, we saw that if we consider finite union of sets, since there are a large number of strings in L_k for a sufficiently large k we are forced us to put many different strings from L_k in some set, which entails that this particular set can not be an *RSL*.

We argue that for some large enough k , this particular set can not be expressed as a concatenation of *RSL*'s either.

Since we can have only finite number of concatenations of *RSL*'s, $L_k = L_1 \cdots L_l$ (l is the maximum number of concatenations we have.) and hence one has to divide any string $x \in L_k$ into at most l pieces s.t. $x = x_1 \cdot x_2 \cdots x_l$. Let $\sharp(x_i)$ denote the number of c 's in x_i .

Suppose for some $x, y \in L_k$, and $j, 1 \leq j \leq l$, $\sharp(x_j) = \sharp(y_j)$, $x = u_1cu_2cu_3$ and $y = v_1cv_2cv_3$ where u_1, u_3, v_1 and v_3 do not have any occurrence of c . Then, $u_2 = v_2$. This is because L_j is an *RSL* and if $u_2 \neq v_2$, we could have bad strings in L_j and hence in L_k . For example if $u_2 = abcab$ and $v_2 = abcabab$, L_j will have strings like $u_1abcaabcu_3$.

Define $x \equiv y$ iff for all $j, 1 \leq j \leq l$, $\sharp(x_j) = \sharp(y_j)$. Since the number of equivalence classes of \equiv depend upon only k and l (where l is fixed), one can increase k such that some equivalence class will have a large number of strings from L_k . Call this class S .

Consequently, since for each $x \in S$, $\sharp(x_i)$ is the same, if we have a large number of strings in S , there will be strings x and y such that $x = u_1cu_2cu_3$ and $y = v_1cv_2cv_3$ but $u_2 \neq v_2$. This will imply that L_j is not an *RSL*, a contradiction. ■

6.3.2 *MonACS*'s and *BFCRSL*

In the following we show that *MonACS*'s exactly characterize the class *BFCRSL*. In Fig. 6.1 we draw a *MonACS* for the language $L = [ab][cd]$ over the alphabet $(\Sigma_1 = \{a, c\}, \Sigma_2 = \{b, d\})$.

Theorem 6.8 $\mathcal{L}(\text{MonACS}) = \text{BFCRSL}$.

Proof: (\subseteq ;) Let $L \in \mathcal{L}(\text{MonACS})$. Then there is an *ACS* \widehat{M} satisfying the monotone condition and $L = L(\widehat{M})$. Let $\widetilde{M} = (\widehat{Q}, \longrightarrow, \overline{q}, F)$ be the product TS of \widehat{M} .

Definition 6.9 For all $\bar{p}, \bar{q} \in \widehat{Q}$, let $\widehat{M}(\bar{p}, \bar{q})$ be a sub-TS of \widehat{M} with \bar{p} as the initial state, \bar{q} as the final state and having all the paths of \widehat{M} starting from \bar{p} and reaching \bar{q} .

Definition 6.10 A transition $\bar{p} \xrightarrow{a} \bar{q}$ is called a cut-edge of \widehat{M} , if it is not a part of any cyclic path in \widehat{M} .

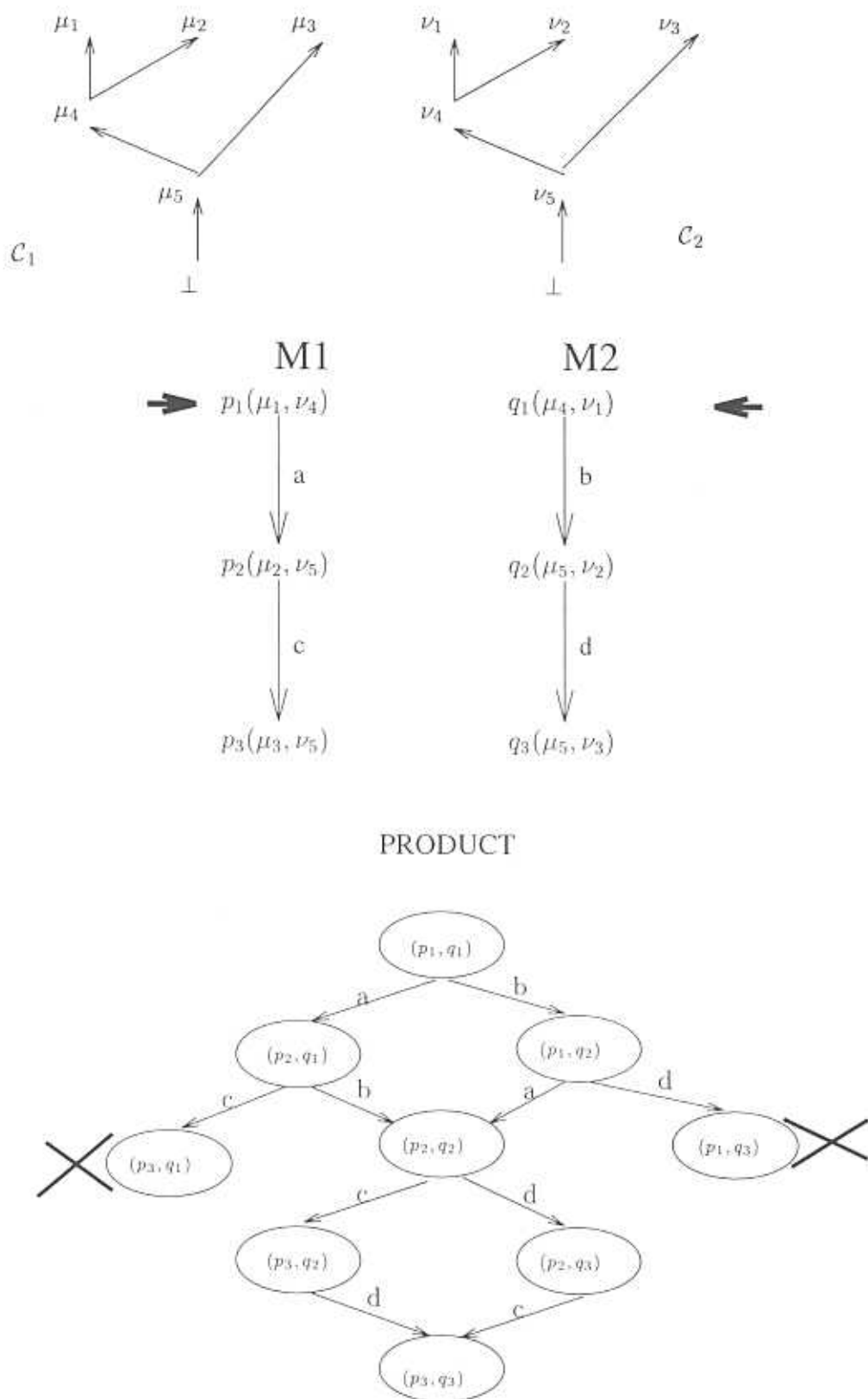


Figure 6.1: *MonACS* for the language $L = [ab][cd]$.

Observation 6.11 1. Suppose for every state \bar{r} and \bar{s} in the sub-TS $\widehat{M}(\bar{p}, \bar{q})$, $f_i(r) = f_i(s)$ for all $i \in \text{Loc}$, i.e., if there is no increase of assumptions for any automaton. Then, it follows from the discussion about SACS's of section 6.1 that $L(\widehat{M}(\bar{p}, \bar{q}))$ is in *RSL*.

2. For any two states \bar{p} and \bar{q} in \widehat{M} , let $p_i = \bar{p}[i]$ and $q_i = \bar{q}[i]$.

If $\bar{p} \xrightarrow{a} \bar{q}$ and for some i, j , $f_i(p_i)(j) \prec_j f_i(q_i)(j)$, then the transition $(\bar{p} \xrightarrow{a} \bar{q})$ is a cut-edge of \widehat{M} since if there was a path from \bar{q} to \bar{p} in the product (thus forming a cycle), there would be a path from p_i to q_i in M_i which would violate monotonicity.

We prove the required inclusion by induction on the number of cut edges in the product. It suffices to consider products having a single initial and a single final state because languages accepted by products with multiple initial and final states can be expressed as union of languages accepted by products with single initial and final state.

Let \bar{q}^f be the final state. Since \bar{q}^0 is the initial state the language accepted by \widehat{M} is $L = L(\widehat{M}(\bar{q}^0, \bar{q}^f))$.

If there are no cut-edges, then it means that the assumptions are static throughout, hence from observation 6.11 (1), we know that L is in *RSL* and hence in *BFCRSL*.

Suppose there are $k(\geq 1)$ cut edges in the product. This implies every path from the initial to the final node must pass through some cut edge. Take the set of cut-edges

$$S = \{ \bar{p}_i \xrightarrow{a_i} \bar{q}_i \mid \text{for all } i \in \{1, \dots, l\} \widehat{M}(\bar{q}^0, \bar{p}_i) \text{ does not have any cut-edge} \}.$$

Then, $L(\bar{q}^0, \bar{q}^f) = \bigcup_{i \in \{1, \dots, l\}} L_i$, where $L_i = L(\bar{q}^0, \bar{p}_i) \cdot a_i \cdot L(\bar{q}_i, \bar{q}^f)$. Notice that $L(\bar{q}^0, \bar{p}_i)$ is an *RSL*, $\{a_i\}$ is an *RSL* trivially and by induction hypothesis, since there is at least one less cut-edge, $L(\bar{q}_i, \bar{q}^f)$ is in *BFCRSL*. Hence, L can be expressed as union of concatenation of *BFCRSL*'s. This concludes the proof.

(\supseteq .) Note that from the previous section, *RSL*'s are accepted by *SACS*'s which are a special case of *MonACS*. For the inclusion it suffices to show the closure of *MonACS*'s with respect to union and concatenation.

Let $L_1 = L(M)$ and $L_2 = L(N)$ where,

- $M = (M_1, M_2, \langle f_1, f_2 \rangle, F)$ be an ACS over $\tilde{\Sigma}$ with \mathcal{C} as the commit alphabet. Let $M_i = (P_i, \longrightarrow_i, p_i^0, f_i)$ be the local AC-automata.
- $N = (N_1, N_2, \langle g_1, g_2 \rangle, G)$ be an ACS over $\tilde{\Sigma}$ with \mathcal{D} as the commit alphabet. Let $N_i = (Q_i, \longrightarrow_i, q_i^0, g_i)$ be the local AC-automata.

Let $f_1(p_1^0) = \langle \lambda_1, \lambda_2 \rangle$, $f_2(p_2^0) = \langle \nu_1, \nu_2 \rangle$, $g_1(q_1^0) = \langle \mu_1, \mu_2 \rangle$ and $g_2(q_2^0) = \langle \eta_1, \eta_2 \rangle$.

Proposition 6.12 $L_1, L_2 \in \mathcal{L}(\text{MonACS})$ implies $L_1 \cup L_2 \in \mathcal{L}(\text{MonACS})$.

Proof:

Our goal is to show that there is a *MonACS* $O = (O_1, O_2, \langle h_1, h_2 \rangle, H)$ over \mathcal{E} as the commit alphabet, where $O_i = (R_i, \Longrightarrow_i, r_i^0, h_i)$, that accepts the union $L_1 \cup L_2$.

We could construct O as we did in Chapter 5. We recall that for this, we had introduced a special state r_i^0 for each local automaton P_i . Note that $R_i = P_i \cup Q_i \cup \{r_i^0\}$. The assumption maps h_i are defined on $P_i \cup Q_i$ as follows.

$$h_i(r) = \begin{cases} f_i(r) & \text{if } r \in P_i. \\ g_i(r) & \text{if } r \in Q_i. \end{cases}$$

In addition, $h_1(r_1^0) = \langle \lambda_1 \vee \mu_1, \lambda_2 \wedge \mu_2 \rangle$, and $h_2(r_2^0) = \langle \nu_1 \wedge \eta_1, \nu_2 \vee \eta_2 \rangle$.

Unfortunately, the construction above for union does not work because the special initial state introduced leads to nonmonotonicity of assumptions. This is because : if $r_2^0 \xrightarrow{a} p_2$ and $p_2 \neq r_2^0$, then $h_2(r_2^0)(1) = (\nu_1 \wedge \eta_1)$ and $h_2(p_2)(1) = g_2(p_2)(1) \in \mathcal{C}_1$ and they can, in general, be unrelated in the ordering thus violating the monotonicity condition.

Hence, we resort to the class of *MonACS*'s with multiple initial states. In this case no special initial state is necessary, and one takes the (component-wise) disjoint union of the two ACS's maintaining the same assumption maps. Obviously, monotonicity of assumptions is maintained and the simple construction for union as mentioned above gives us the desired result. ■

Proposition 6.13 $L_1, L_2 \in \mathcal{L}(\text{MonACS})$ implies $L_1 \cdot L_2 \in \mathcal{L}(\text{MonACS})$.

Proof: It suffices to consider the case when M has a single final state (p_1^f, p_2^f) and N has a single initial state, because then the concatenation in general can be expressed as union of concatenation of ACS's with single initial and single final state.

Again, as in Chapter 5, we can construct O accepting $L_1 \cdot L_2$. Recall the introduction of a special state $(p_i, 0)$ for all $i \in \text{Loc}$ and the definition of the assumption maps h_i .

$$h_i(r) = \begin{cases} f_i(r) & \text{if } r \in P_i, \\ g_i(r) & \text{if } r \in Q_i, \end{cases}$$

Also $h_1(p_1, 0) = \langle (\lambda_1 \vee \mu_1), (\lambda_2 \wedge \mu_2) \rangle$, and $h_2(p_2, 0) = \langle (\nu_1 \wedge \eta_1), (\nu_2 \vee \eta_2) \rangle$.

For the same reason as in the case of “union”, this construction also does not work because for the states $q \in Q_1$ such that $(p_1, 0) \xrightarrow{a}_1 q$, $h_1(q)(2) \in \mathcal{D}$ whereas $h_1(p_1, 0)(2)$ is a new element and they may be unrelated.

To remedy this, we redefine h_1 as : $h_1(q)(i) = h_1(p_1, 0)(i) \vee g_1(q)(i)$ for all $q \in Q_1$ while keeping the assumption maps of P_1 same as before. Now P is a monotone ACS because by assumption both M and N were monotone. We notice that states of N_i are reachable only via the special states $(p_i, 0)$ and hence, internal states of M_i and N_j ($i \neq j$) never form a *reachable* global state, hence the extra assumption acts only like a dummy and the compatible states are the same as there were originally. Then, the proof that P accepts $L_1 \cdot L_2$ goes through. Thus the inclusion $\mathcal{L}(\text{MonACS}) \supseteq \text{BFCSRSL}$ holds and the theorem is proved. ■

7 Conclusion

7.1 Summary

In this thesis, we have addressed the problem of how the behaviour of finite state distributed systems can be given *local presentations*. Recall that in Chapter 1, we suggested the following notion as constituting local presentation:

a class of distributed systems is **locally presented** if it is modeled as a set of components, one for each process, and the global behaviour is completely defined by a **fixed** set of construction rules **universal** to that class.

The model of distributed systems studied was that of finite state automata which communicate by handshake synchronization. Behaviour was given in terms of language acceptance, using finite words or infinite words. Products of automata provided the framework by which the (global) system was built from ‘local specifications’. This meant that a syntax for such systems also reflect such ‘top level parallel’ behaviour.

In Chapter 2, we first observed that synchronized products of automata are expressively weak (Corollary 2.14 and Theorem 2.28), and raised the issue of obtaining richer behaviour, namely that of regular consistent languages (Definition 2.19) in terms of products. This was studied in Chapter 3, where **view-based** systems were proposed (Definition 3.2) and shown to characterize regular consistent languages (Theorem 3.3). The proof also revealed the way processes in distributed systems update their partial views of global states

by exchanging view information during synchronization. However, no satisfactory syntax could be provided for the systems.

Given that view based systems offered a way to store and update partial global information in local states of processes, the next natural question was how much farther this could be taken. Is there a way of capturing **all** regular behaviours in terms of products by suitably enriching local state information? This led to the class of *Assumption-Compatible Systems* studied in Chapter 5 (Definition 5.2). These systems are of independent interest, coming from a paradigm extensively studied in the formal specification and verification of distributed systems, and we illustrated the use of the paradigm in Chapter 4.

In a sense, the central result of the thesis is Theorem 5.6, which asserts that every regular language over a finite alphabet Σ can be given a local presentation over a specified distributed alphabet $\bar{\Sigma}$ by choosing an appropriate structure of assumptions and commitments between the n processes. This is further supported by Theorem 5.47, offering a top-level parallel syntax, reflecting the distributed nature of the systems. We also showed that the generalization of these results to infinite behaviours could be carried out smoothly. (A caveat here is that the syntax for infinite behaviours required global complementation operation, diminishing the distributed nature somewhat.)

Assumption-compatible systems are not only the most expressive local presentations, but subsume other candidate presentations in the thesis: in Chapter 6, we showed that the other classes of behaviours studied in Chapters 2 and 3 could be obtained by simple restrictions on commit alphabets. Moreover, other restrictions showed classes of languages strictly between regular consistent languages and the class of all regular languages (over an alphabet).

Having said this, we now turn to what has **not** been done in the thesis, but nevertheless is of interest in the context of local presentations for finite state distributed systems.

7.2 Automata theoretic issues

Having chosen automata theory as the tool for this study, a number of questions are immediately posed:

- We have studied systems in which the number of processes is *fixed*. This can be at once generalized to **open** systems, where the number of agents is finite but unbounded. The syntax for this class is obvious:

$$REGSYS ::= r \in REG_{\Sigma} \mid r_1 \parallel r_2$$

Since the semantics of \parallel always gives a commutative and associative operation (the way we have done throughout), we get a top-level parallel composition of (finite but unboundedly many) regular expressions, the alphabet of each process being determined by the letters in Σ syntactically occurring in it.

Generalizing assumption-commitment in an appropriate fashion for such open systems seems feasible and interesting. In terms of the distribution problem, the constraint on distribution must be formulated differently: for instance, the size of each local alphabet may be bounded by some number.

- A more challenging study is that of dynamic networks of processes. In terms of syntax, this corresponds to \parallel occurring in the scope of other operators.

$$PREG ::= a \in \Sigma \mid p + q \mid p; q \mid p^* \mid p_1 \parallel p_2$$

However, the meaning of synchronization is more complicated now, and several semantic issues are involved [Old]. One semantic approach is to consider $r_1; (r_2 \parallel r_3)$ as a *forking* of two processes in the style of Unix systems. An interesting issue is whether there is any need of an iterated parallel operator, like “*” iterates over “;” [LW]. As in the study of *dot depth*, we believe that *parallel depth* can shed interesting insight into local presentations of behaviour.

- A great strength of automata theory is in *algebraic* study of behaviour. In the spirit of local presentations, we expect that ACS's can be characterized using local right congruences of finite index and an associated global congruence. Algebraic theories of distributed decomposition are needed for a clear understanding of this subject.

7.3 Complexity

It is clear from the proof of Theorem 5.6 in Chapter 5 that given a finite state automaton over Σ , an equivalent ACS over $\tilde{\Sigma}$ can be constructed *effectively*. However, we have not studied the complexity of the construction, and particularly, issues of minimizing the number of local states.

This immediately brings in succinctness questions: for a given finite behaviour what is the best (in terms of least size of local automata and commitment alphabet) distribution? How does the distribution of the alphabet affect sizes of local automata? Are there minimal local presentations? These questions are of paramount importance if one wants to use ACS's to model finite state distributed systems.

7.4 Model-checking

In terms of applications, the most immediate relevance of the paradigm studied here is in the area of *automata-theoretic methods in verification*, collected under the rubric of **model-checking** [VW].

Briefly, we have a finite state system S , a property α , and ask whether it is the case that every infinite run of S satisfies α . The property α is stated in a formal logic – say, the propositional temporal logic of linear time. Models of α are again infinite sequences, those that satisfy the property. It can be shown that given any such α , there is a Büchi automaton \mathcal{A}_α such that $Models(\alpha) = L_\omega(\mathcal{A}_\alpha)$. Now, when S is presented as a Büchi automaton, we construct $\mathcal{A}_{\neg\alpha}$, and check whether $L_\omega(S) \cap L_\omega(\mathcal{A}_{\neg\alpha})$ is empty. The emptiness check exactly corresponds to the verification problem we started out with. Such a model checking

procedure runs in time $O(m.2^k)$, where m is the number of states in S and k is the length of the formula α (since the size of the automaton associated with α is exponential in the size of α).

The main need of automatic model checking comes in the verification of systems which exhibit nontrivial concurrency. For instance, in hardware, a chip is a system with a few million parallel components, each of which can be comprehensively modelled as an automaton with a small number of states. Here, each of the automata is easy to study, but the global system cannot be studied manually at all. But then the verified system called S above is the product of these automata, and the size of the product is exponential in the number of components. This is often referred to as the **state explosion problem**.

Now consider the situation when the system S is a locally presented ACS with n components. If the property α is similarly structured, say in the form, $\alpha_1 @ 1 \wedge \dots \wedge \alpha_n @ n$ (read α at 1 etc.) (where α_i includes specification of assumptions that process i makes), one can expect to generalize the methodology above for *local* model checking of components S_i against α_i . This may be a pragmatic approach to solving the state explosion problems, at least in situations where the property being verified lends itself to such distribution.

One place where the problem may manifest itself is in the construction of local AC-automata. The construction in Chapter 5 clearly shows that what we are doing is a kind of *unfolding* of the global automaton and then taking projections to get local AC-automata. This suggests a strong relationship to the net-unfolding techniques proposed by of McMillan [Mcm] and developed by Esparza [Esp1, Esp2, ERV]. If one has to unfold many times, then the constructed local AC-automata will be very large and obviously any advantages of local model checking will be lost. Hence the question of how much unfolding is necessary for the decomposition is an important one.

Bibliography

- [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.
- [BC] Boudol, G. and Castellani, I., "Flow models of distributed computations: event structures and nets", *Rapports de Recherche 1482*, INRIA, July 1991.
- [BKP] Barringer, H., Kuiper, R. and Pnueli, A., "Now you may compose temporal logic specifications", *Proc. of STOC*, 1984, 51-63.
- [BP] Beauquier, J. and Petit, A., "Distribution of sequential processes", *Proc. of MFCS in LNCS 324*, 1988, 180-189.

- [Bed] Bednarczyk, M.A., "Categories of Asynchronous Systems", *PhD Thesis*, University of Sussex, 1987.
- [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.
- [Dro] Droste, M., "Event structures and domains", *Theo. Comp. Sc.*, vol. 68, 1989, 37-47.
- [Esp1] Esparza, J., "Model checking using net unfoldings", *Sci. of Comp. Prog.*, vol. 23, 1994, 151-195.
- [Esp2] Esparza, J., "Decidability of model checking for infinite-state concurrent systems", *Acta Inf.*, vol. 34, 1997, 85-107.
- [ERV] Esparza, J., Römer, S. and Vogler, W., "An improvement of McMillan's unfolding algorithm", *TACAS*, 1996, 87-106.
- [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*, American Mathematical Society, 1967, 19-32.
- [FP] Francez, N., and Pnueli, A., "A proof method for cyclic programs", *Acta Inf.*, vol. 9, 1978, 138-158.
- [Hoa] Hoare, C.A.R., *Communicating sequential processes*, Prentice-Hall International, Englewood Cliffs, 1985.
- [HU] Hopcroft, J.E. and Ullman, J.D., *Introduction to the theory of automata, languages and computation*, Addison-Wesley, 1979.

- [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.
- [Lam1] Lamport, L., "An Assertionl Correctness Proof of a Distributed Algorithm". *Sci. of Comp. Prog.*, vol. 2, 1982, 175-206.
- [Lam2] Lamport, L., "Specifying Concurrent Program Modules". *TOPLAS*, vol. 5, 1983, 190-222.
- [LL] Lamport, L. and Lynch, N., "Distributed computing: models and methods", *Handbook of Theoretical Computer Science*, vol. B, ed: van Leeuwen, Elsevier, 1990.
- [LP] Lichtenstein, O. and Pnueli, A., "Checking that finite state concurrent programs satisfy their linear specification". *POPL*, 1985, 97-107.
- [LPRT] Lodaya, K., Parikh, R., Ramanujam, R. and Thiagarajan, P.S., "A logical study of distributed transition systems", *Inf. and Comp.*, vol. 119, 1995, 91-118.
- [LW] Lodaya, L. and Weil, P., "Series-parallel posets: algebra, automata and languages", *Proc. STACS '98 in LNCS 1373*, 1998, 555-565.
- [Maz] Mazurkiewicz, A., "Basic notions of trace theory", *LNCS 354*, 1989, 285-363.
- [Mem] McMillan, K.L., "Using unfolding to avoid the state explosion problem in the verification of asynchronous circuits, *Proc. of CAV'92*, Montreal, 1992, 164-174.
- [MC] Misra, J., and Chandy, M., "Proofs of networks of processes", *IEEE Trans. on Soft. Engg.*, vol. 7, 1981, 417-426.
- [Mil] Milner, A.R.G., *Communication and concurrency*, Prentice-Hall International, Englewood Cliffs, 1989.

- [MP] Manna, Z. and Amir Pnueli, A., *Temporal logic of reactive systems: specification*, Springer-Verlag, New York, 1992.
- [MR1] Mohalik S.K. and Ramanujam, R., "Distributing globally specified automata", *Manuscript*, December 1995.
- [MR2] Mohalik S.K. and Ramanujam, R., "Assumption-Commitment in automata", *Proc. of FST & TCS '97 in LNCS 1346*, Kharagpur, 1997, 153-168.
- [MR3] 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 .
- [Muk] Mukund, M., "Petri nets and step transition systems", *Int. Jl. Found. Comp. Sci.*, vol. 3, 1992, 443-478.
- [NRT] Nielsen, M., Rozenberg, G. and Thiagarajan, P.S., "Behavioural notions for elementary net systems", *Dist. Comp.*, vol. 4, 1990, 45-57.
- [Och] Ochmanski, E., "Regular behaviour of concurrent systems", *Bulletin of the EATCS*, vol. 27, 1985, 56-67.
- [Old] Olderog, .*Nets, terms and formulas*, Cambridge Tracts in Theoretical Computer Science 23, Cambridge University Press, 1991.
- [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.

- [Pnu] Pnueli, A., "Application of temporal logic to the specification and verification of reactive systems: A survey of current trend", *LNCS 224*, 1986, 510-584.
- [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.
- [Tho] Thomas, W., "Automata on infinite objects", *Handbook of Theoretical Computer Science, vol. B*, ed: van Leeuwen, Elsevier, 1990, 133-191.
- [Var] Vardi, M.Y., "Verification of open systems", Invited talk in *Proc. of FST & TCS '97 in LNCS 1346*, Kharagpur, 1997, 250-266.
- [VW] Vardi, M.Y. and Wolper. P., "An automata-theoretic approach to automatic program verification", *Proc. of the First Symp. on Logic in Computer Science*, Cambridge, 1986, 138-266.
- [WN] Winskel, G. and Nielsen, M., "Models for concurrency", in *Handbook of Logic in Computer Science, vol. 4*, eds: Abramsky, Gabbay and Maibaum, Oxford Science Publications, New York, 1992, 1-148.
- [Zie] Zielonka, W., "Notes on finite asynchronous automata", *RAIRO-Inf. Theor. et Appli.*, vol. 21, 1987, 99-135.