

# **Automatic Synthesis of Distributed Transition Systems**

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik  
der Universität Stuttgart zur Erlangung der Würde eines  
Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

**Alin Ștefănescu**

aus Slatina, Rumänien

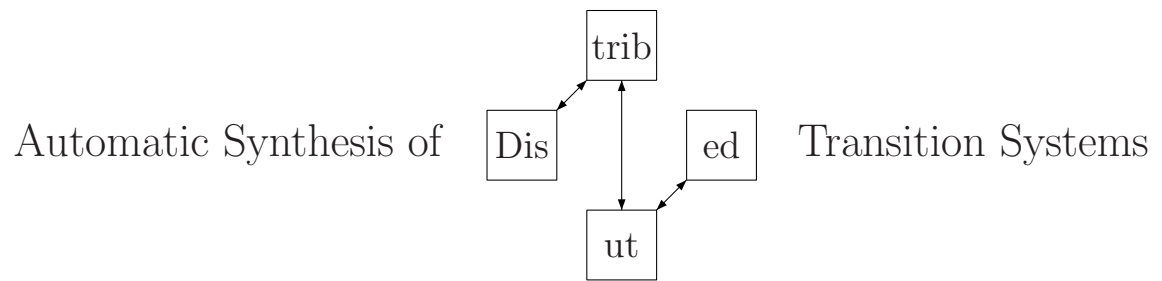
Vorsitzender des Prüfungsausschusses: Prof. Dr. Volker Diekert  
Hauptberichter: Prof. Dr. Javier Esparza  
Mitberichter: Prof. Dr. Colin Stirling

Tag der mündlichen Prüfung: 13.2.2006

Institut für Formale Methoden der Informatik  
Universität Stuttgart

2006





Alin Ștefănescu

March 7, 2006



## Abstract

This thesis investigates the synthesis problem for two classes of distributed transition systems: synchronous products and asynchronous automata. The underlying structure of these models consist of local automata synchronizing on common actions. The synthesis problem discussed is as follows: Given a global specification as a transition system  $TS$  and a distribution pattern  $\Delta$ , find a distributed transition system over  $\Delta$  whose global state space is ‘equivalent’ to  $TS$ . As criteria for the correctness of the (distributed) implementation vs. the specification (i.e., their ‘equivalence’) we use: transition system isomorphism, language equivalence, and bisimilarity respectively. In particular, the synthesis of asynchronous automata modulo language equivalence is a notoriously hard problem solved by Zielonka at the end of the 80s. One of the motivations behind our work was to bring this theory closer to practical applications.

From the theoretical point of view, we conduct a detailed analysis of the synthesis problem for both models of distributed systems, look at effective algorithmic approaches and draw a map of computational complexity results. E.g., we provide several matching lower and upper complexity bounds for the distributed implementability problem.

From the practical perspective, we provide prototype implementations for most of the synthesis algorithms discussed in the thesis. Moreover, we offer assistance when a given specification is not distributable by trying to modify this specification such that distributed synthesis can be applied. By using several heuristics to overcome the classical state space explosion, we are able to automatically generate small distributed algorithms for problems such as mutual exclusion.



## Zusammenfassung

Diese Dissertation erforscht das Syntheseproblem für zwei verschiedene Arten von verteilten Transitionssystemen: synchrone Produkte und asynchrone Automaten. Diese Modelle bestehen aus lokalen Automaten, die sich über gemeinsame Aktionen synchronisieren. Das betrachtete Syntheseproblem ist Folgendes: Für ein gegebenes Transitionssystem  $TS$  und eine Verteilungsstruktur  $\Delta$  sucht man ein verteiltes Transitionssystem über  $\Delta$  mit einem globalen Zustandsraum "äquivalent" zu  $TS$ . Als Kriterien für die Korrektheit der Implementierung bezüglich der Spezifikation (d.h., ihre "Äquivalenz"), verwenden wir Transitionssystem-Isomorphismus, Sprachenäquivalenz bzw. Bisimilarität. Insbesondere ist die Synthese der asynchronen Automaten Modulo-Sprachenäquivalenz ein bekanntes hartes Problem, das von Zielonka Ende der 80iger Jahre gelöst wurde. Einer der Beweggründe hinter unserer Arbeit war diese Theorie näher an praktische Anwendungen zu holen.

Aus theoretische Sicht leiten wir eine ausführliche Analyse des Syntheseproblems für beide Modelle der verteilten Systeme, betrachten wirkungsvolle algorithmische Verfahren und erarbeiten eine Übersicht der Berechnungskomplexität. So z.B. stellen wir einige untere und obere Komplexitätsgrenzen für das verteilte Implementierungsproblem zur Verfügung.

Von einer praktischen Ansicht liefern wir Prototypimplementierungen für die meisten Synthesealgorithmen, die in diese Dissertation besprochen werden. Außerdem bieten wir Unterstützung an, falls eine gegebene Spezifikation nicht verteilbar ist und versuchen, sie zu ändern, so daß verteilte Synthese möglich wird. Indem wir einige Heuristiken verwenden, zum der klassischen Zustandsraumsexplosion zu überwinden, sind wir in der Lage, kleine verteilte Algorithmen für Probleme, wie z.B. der gegenseitige Ausschluß (Mutex), automatisch zu erzeugen.





## Acknowledgments

In the first place I would like to thank to my supervisor Javier Esparza, for all the kindness he has showed, the model he has been, and all the help he has provided to make this work come into being.

Then, there is a long list of people I am indebted to regarding this work and my formation. I will mention here only a few of them: Keijo Heljanko, Barbara König, Vitali Kozioura, Ioana and Laurențiu Leuştean, Rémi Morin, Anca Muscholl, Claus Schröter, Stefan Schwoon, Gheorghe Ștefănescu, Ljiljana and Joseph Spadavecchia, Marin Toloși. Moreover, to all the friends in München, Edinburgh, and Stuttgart I am grateful for the moments we shared. Also, I thank Stefan Leue for the support during the last part of the writing-up process and Colin Stirling for accepting to referee this thesis.

I owe a deep debt of gratitude to my family for love, support, and understanding.



## Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

*(Alin Ștefănescu)*



This thesis is dedicated to my daughter Anastasia  

---

Because of Anastasia, the thesis may have a couple of sections less.  
Because of the thesis, Anastasia had her father around less.

This thesis is also dedicated to my wife Cornelia  

---

Because of Cornelia, I have now both the thesis and Anastasia.



# CONTENTS

---

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	The Synthesis Problem . . . . .	2
1.2	A Small Example . . . . .	3
1.3	The Contribution of this Thesis . . . . .	7
<b>2</b>	<b>Preliminaries</b>	<b>9</b>
2.1	Basic Notions and Notations . . . . .	9
2.1.1	Set Theory . . . . .	9
2.1.2	Algebraic Notions . . . . .	11
2.1.3	Formal Languages . . . . .	11
2.1.4	Graph Theory . . . . .	16
2.2	Transition Systems . . . . .	18
	Discussion . . . . .	23
<b>3</b>	<b>Distributed Transition Systems and the Synthesis Problem</b>	<b>25</b>
3.1	Trace Theory . . . . .	25
3.2	Distributed Transition Systems . . . . .	31
3.2.1	Distributions . . . . .	32
3.2.2	Synchronous Products of Transition Systems . . . . .	34
3.2.3	Asynchronous Automata . . . . .	37
3.3	Shapes . . . . .	40
3.3.1	Diamonds <i>et al.</i> . . . . .	41
3.3.2	Synchronous Products of Transition Systems . . . . .	44
3.3.3	Asynchronous Automata . . . . .	47
3.4	Languages . . . . .	50
3.4.1	Final States . . . . .	51
3.4.2	Traces of Diamonds . . . . .	51
3.4.3	Synchronous Products of Transition Systems . . . . .	55
3.4.4	Asynchronous Automata . . . . .	63
3.4.5	Comparative Expressiveness . . . . .	65
3.5	The Synthesis Problem . . . . .	68
	Discussion . . . . .	70

<b>4</b>	<b>The Complexity of the Distributed Implementability Test</b>	<b>71</b>
4.1	The Distributed Implementability Problem . . . . .	72
4.2	Implementability modulo Isomorphism . . . . .	74
4.2.1	Synchronous Products of Transition Systems . . . . .	74
4.2.2	Asynchronous Automata . . . . .	83
4.2.3	Implementability for Concurrent Alphabets . . . . .	92
4.3	Implementability modulo Language Equivalence . . . . .	99
4.3.1	Synchronous Products of Transition Systems . . . . .	99
4.3.2	Asynchronous Automata . . . . .	105
4.3.3	Non-regular specifications . . . . .	113
4.4	Implementability modulo Bisimulation . . . . .	114
4.4.1	Synchronous Products of Transition Systems . . . . .	115
4.4.2	Asynchronous Automata . . . . .	116
4.5	Relaxed Implementability . . . . .	117
4.5.1	Language Inclusion . . . . .	118
4.5.2	Isomorphic Embedding Heuristic . . . . .	121
	Discussion . . . . .	126
<b>5</b>	<b>Synthesis of Distributed Transition Systems</b>	<b>129</b>
5.1	Synchronous Products of Transition Systems . . . . .	129
5.1.1	Synthesis modulo Isomorphism . . . . .	129
5.1.2	Synthesis modulo Language Equivalence . . . . .	130
5.2	Asynchronous Automata . . . . .	131
5.2.1	Synthesis modulo Isomorphism . . . . .	131
5.2.2	Synthesis modulo Language Equivalence . . . . .	132
5.2.3	Alternative Constructions for Special Cases . . . . .	138
	Discussion . . . . .	152
<b>6</b>	<b>Implementations and Case Studies</b>	<b>153</b>
6.1	Motivating Example: Mutual Exclusion . . . . .	156
6.1.1	A Classical Solution for Mutual Exclusion . . . . .	157
6.1.2	Mutual Exclusion Modeled in Our Framework . . . . .	159
6.1.3	Mutual Exclusion Revisited . . . . .	164
6.1.4	Parametrized Mutual Exclusion . . . . .	170
6.1.5	Dining Philosophers . . . . .	171
6.2	Implementation for Synchronous Products of Transition Systems . . . . .	173
6.3	Implementations for Asynchronous Automata . . . . .	178
6.3.1	Synthesis modulo Isomorphism . . . . .	178
6.3.2	Heuristics to Construct Under-Approximations . . . . .	182
6.3.3	A Heuristic using Unfoldings for Zielonka's Construction . . . . .	187
	Discussion . . . . .	196
<b>7</b>	<b>Conclusions</b>	<b>199</b>
<b>A</b>	<b>Appendix</b>	<b>201</b>
A.1	Implementability Test modulo Isomorphism for Asynchronous Automata . . . . .	201



**Bibliography**

**203**

**Index**

**213**



# LIST OF FIGURES

---

1.1	Synthesis at work for a simple Producer-Consumer problem . . . . .	4
1.2	A diagrammatic flow of the synthesis approach followed in this thesis . . .	7
2.1	Example of a graph [Fig93a] . . . . .	17
2.2	A house-like transition system . . . . .	20
3.1	The dependence graph of the concurrent alphabet of the house-like transi- tion system . . . . .	26
3.2	A detail in the proof of Proposition 3.7 . . . . .	29
3.3	The synchronization of two workers . . . . .	36
3.4	A ‘futuristic ambient’ asynchronous automaton that is not a synchronous product . . . . .	39
3.5	The independent and forward diamond properties . . . . .	42
3.6	A transition system satisfying ID and FD, but not isomorphic to an asyn- chronous automaton . . . . .	42
3.7	The relation between conditions $SP_1$ – $SP_3$ and the definition of synchronous products . . . . .	45
3.8	An asynchronous automaton that is not a synchronous product . . . . .	46
3.9	Two transition systems satisfying FD whose languages are not forward-closed	53
3.10	Deterministic finite automaton satisfying FD whose language is not forward- closed . . . . .	55
3.11	Nondeterministic synchronous product with local final states accepting a finite union of regular product languages . . . . .	61
3.12	Deterministic synchronous product with final states accepting $\{\varepsilon, a, b\}$ for $a  b$ . . . . .	62
3.13	Comparison between the classes of languages of distributed transition systems	66
3.14	Comparison between the classes of languages of <i>acyclic</i> distributed transi- tion systems . . . . .	67
3.15	Comparison between the classes of languages of distributed transition sys- tems with <i>final states</i> . . . . .	68
3.16	Comparison between the classes of languages of acyclic distributed transi- tion systems with <i>final states</i> . . . . .	68
3.17	Transition system for the language of Example 3.69 [Zie87] . . . . .	69
3.18	The synthesis problem for distributed transition systems . . . . .	69

4.1	The transition system $TS$ associated to $\phi = (x_1 \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3})$ . . . . .	75
4.2	The transition system $TS$ associated to $\phi = (x_1 \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3})$ (without the states and transitions associated to clause $c_0$ ) . . . . .	86
4.3	Example related to the construction for Lemma 4.17 . . . . .	95
4.4	Visual aid for the inductive proof of Lemma 4.17 . . . . .	98
4.5	A schematic representation of the reduction in the proof of Lemma 4.22 . .	103
4.6	A schematic representation of the reduction in the proof of Theorem 4.26 .	106
4.7	A schematic representation of the reduction in the proof of Proposition 4.27	110
4.8	The transition system $TS$ associated to $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})$ . . .	123
5.1	The partial order of a trace . . . . .	133
5.2	The backward diamond property . . . . .	139
5.3	A <i>cyclic</i> transition system satisfying ID, FD, and BD that is not isomorphic to an asynchronous automaton . . . . .	141
5.4	Details for Step 2 of the proof of Theorem 5.14 . . . . .	143
5.5	Details for Step 3 of the proof of Theorem 5.14 . . . . .	145
5.6	Exemplifying unfolding using copies . . . . .	147
6.1	The synthesis flow followed in this thesis . . . . .	154
6.2	The life-cycle of a local process involved in mutual exclusion . . . . .	156
6.3	A classical synthesized solution to mutual exclusion problem [CE82] . . . .	158
6.4	Synthesis cascade for the mutual exclusion problem $Mutex_1$ . . . . .	161
6.5	Synthesis cascade for the revisited mutual exclusion problem $Mutex_2$ . . .	166
6.6	Synthesized mutual exclusion algorithm for $Mutex_2$ using (two) shared variables . . . . .	167
6.7	Implementation for the synthesis of deterministic synchronous products . .	176
6.8	A heuristic using unfoldings and an implementability test . . . . .	188
6.9	Example of the unfolding compared with Zielonka's construction . . . . .	193

# LIST OF TABLES

---

4.1	Complexity results for the implementability of synchronous products of transition systems with one initial state ( $ I  = 1$ ) . . . . .	72
4.2	Complexity results for the implementability of asynchronous automata (with multiple initial states) . . . . .	73
4.3	The equivalence classes constructed by Algorithm 4.1 . . . . .	79
4.4	Details for the satisfaction of the $\mathbf{SP}_3$ property . . . . .	79
4.5	Details for the satisfaction of the $\mathbf{SP}_2$ property (only the cases not solved already by Table 4.4) . . . . .	88
4.6	The equivalence classes constructed by Algorithm 4.2 . . . . .	89
4.7	Details for the satisfaction of the $\mathbf{AA}_3$ property . . . . .	90
4.8	Details for the satisfaction of the $\mathbf{AA}_2$ property (only cases not solved already by Table 4.7) . . . . .	101
4.9	Complexity results for the implementability of synchronous products of transition systems with multiple initial states ( $ I  \geq 1$ ) . . . . .	111
6.1	The (sizes of the) benchmarks used in this chapter . . . . .	171
6.2	Synthesis of deterministic synchronous products (modulo language equivalence) . . . . .	177
6.3	Synthesis of asynchronous automata modulo isomorphism . . . . .	181
6.4	Construction of under-approximations satisfying ID and FD . . . . .	184
6.5	Constructing an under-approximation that is isomorphic to an asynchronous automaton . . . . .	186
6.6	Synthesis of (deterministic) asynchronous automata . . . . .	195



# LIST OF ALGORITHMS

---

4.1	Construction of local equivalences for the second part of the proof of Theorem 4.3 . . . . .	78
4.2	Construction of local equivalences for the second part of the proof of Theorem 4.11 . . . . .	88
4.3	Construction of the least family of equivalences satisfying the $DAA_1$ and $DAA_2$ rules (of Theorem 4.19) . . . . .	96
4.4	A test whether the language of a transition system is a product language .	101
4.5	A test whether the language of a regular expression is prefix-closed . . . .	112
4.6	A polynomial test if a transition system is bisimilar to a deterministic asynchronous automaton . . . . .	117
6.1	The reduction of the synthesis problem for deterministic synchronous products to the non-reachability of 1-safe Petri nets . . . . .	175
6.2	An unfolding approach for the synthesis of deterministic asynchronous automata (based on Zielonka's equivalence) . . . . .	190





---

We are at the very beginning of time for the human race. It is not unreasonable that we grapple with problems. But there are tens of thousands of years in the future. Our responsibility is to do what we can, learn what we can, improve the solutions, and pass them on.

*Richard Feynman*

# CHAPTER 1

## INTRODUCTION

---

THE design of distributed systems has always been a challenging and error-prone task. The difficulty resides in the multitude of possible interactions between the concurrent components of the system. In this thesis, we study automatic procedures of generating distributed implementations from global specifications.

There are two complementary classic approaches to the problem of finding a (distributed) model for a given specification: *verification* and *synthesis* (both proposed in the seminal works [CE82, MW84]). The verification procedure checks if a model (provided by the user) satisfies the specification, whereas in the synthesis approach the model is directly generated from the specification. Regarding verification, if the model fails to fulfill the specification, it should be iteratively improved and checked again by the user. In the latter case of synthesis, the system is correct by construction, and therefore no need for further verification. Despite this advantage and the fact that the approaches have similar computational complexities, automatic synthesis has so far been less successful than verification.

A possible explanation to this fact is that both approaches were mostly studied using temporal logics like LTL or CTL for the specification. A shortcoming of these logics is their inability of expressing properties about casual *independences* between the different actions of the system, hence these properties cannot play a rôle in the synthesis procedure. Although a couple of distributed versions of temporal logics have been proposed over the last decade [TH98, Wal98, AS02, GM02, Wal02], they have not proved to be really better in practice and a good candidate of a local temporal logic able to easily express natural properties of distributed systems is still to be discovered.

*Asynchronous automata* are a well-known formal model that embed an independence relation between actions. They were proposed by Zielonka [Zie87] as a natural generalization of finite state automata to concurrent systems and, loosely speaking, they are sequential automata communicating through ‘rendez-vous’. Asynchronous automata have been introduced to provide a model of computation for *trace languages*, which are languages closed under an explicit independence relation between actions. (The trace languages were introduced by Mazurkiewicz [Maz77] as a simple yet powerful formal language tool to capture distributed behavior.) Zielonka gives a construction that accepts as input a regular trace language and a distribution pattern and outputs an asynchronous automaton accepting the given language.

Little has been done to use the powerful theory of asynchronous automata to more practical applications and to turn it into a reliable computer science tool: To the best of our knowledge no steps towards implementation of it have been taken. A possible explanation is the fact that the Zielonka's algorithm is complicated and computationally involved despite attempts over the years to simplify it. This thesis challenges the synthesis of asynchronous automata from global specifications closed under independence. Since we aim towards practical applications, we conduct a careful computational complexity study of the problem together with heuristics targeting smaller solutions.

## 1.1 The Synthesis Problem

We study the problem of automatic synthesis of distributed systems. In system synthesis we transform a specification into a system that is guaranteed to satisfy the specification. This problem has been investigated in various frameworks and generally proved to be a rather difficult question. In the following, we mention seminal works, which generated the mainstreams in the area, and try to motivate our research.

**Temporal Logics** Most related work was carried out using temporal logic for the specification. In [CE82], Emerson and Clarke proposed a synthesis procedure using the branching time temporal logic CTL. Of similar nature, but using the linear time temporal logic LTL, is the approach by Manna and Wolper in [MW84]. The main problem with approaches based on (classical) temporal logics is that such logics are not able to express the independences between the involved actions, thus the procedure synthesizes a global transition system and not a concurrent one. Moreover, the constructed transition system might not be distributable (i.e., there is no distributed system exhibiting the same behavior).

Further research on synthesis using temporal logics was generated by Pnueli and Rosner [PR89], who studied the synthesis of reactive modules (open systems interacting with their environment). In this setting, there is a module and its environment that alternate their moves as in a game and the goal is to find a winning strategy for the module (in other words, the module should satisfy the specification irrespective of how the environment behaves). The problem received further attention (see for example [KV01] for recent acquisitions in the area), but the specification does not incorporate the notion of independence of actions and the result of synthesis is a module and not a distributed system. Another important disadvantage is that the problem is in most cases undecidable.

Recently distributed games were also considered [MW03] and the results look more promising, since they are able to subsume other frameworks in the literature (for instance, [MT02]). This still does not solve the high undecidability of the synthesis problem in general, but it is hoped that the model will contribute to the identification of interesting decidable fragments.

**Petri nets** Petri nets [Pet62] are well established models for concurrent systems. Ehrenfeucht and Rozenberg introduced in [ER90], the *theory of regions* as tool to synthesize a Petri net isomorphic to a given transition system. The approach was success-

ful [NRT92, BD98, CKLY98, BCD02] and it has inspired applications in the automatic synthesis of asynchronous circuits [CKK<sup>+</sup>02] and synthesis of distributed transition systems [Mor98, CMT99, Muk02]. Nonetheless, the approach still lacks a very flexible specification language, independence between actions is not explicit and has no abstract notion of action such that different system actions correspond to the same abstract action.

**Mazurkiewicz traces and asynchronous automata** The notion of *trace* was proposed by Mazurkiewicz [Maz77] for the study of concurrent systems and constitutes now a classical subject (see [DR95] for the extensive research over the years on traces). A concurrent behavior is captured by enriching a set of actions with the information about *independence* of actions, denoted  $\parallel$ . (The notation  $\parallel$  suggests that two independent actions can be executed in parallel and therefore can appear in a computation in any order.) The executions of a distributed system can then be naturally grouped together into equivalence classes, where two computations are equated in case they are two different interleavings of the same partial order stretch of behavior. A *trace* is just such an equivalence class of computations. Further, a *trace language* is a language closed under the independence relation  $\parallel$ . A trace language is *regular* if it is accepted by a conventional finite state machine. (Note that the behavior of a distributed system exhibiting an independence relation on the actions is always a trace language.)

A problem debated for a long time in the 80s, was to find a distributed model exactly characterizing the class of regular trace languages. Zielonka [Zie87] solved it, introducing the concept of *asynchronous automata*. An asynchronous automaton consists of a set of local automata that periodically synchronize according to a communication structure in order to process the input.

In this thesis, we choose the models of traces and asynchronous automata as theoretical foundations for the proposed synthesis of distributed systems. There are meaningful results characterizing their relation and they are naturally modeling the behavior of a concurrent system involving independence between actions.

**Synchronous products of transition systems** The asynchronous automata are in fact a generalization of a well-known model of distributed transitions systems: *the synchronous products of transitions systems* [Arn94]. Although strictly less expressive than the asynchronous automata, they enjoy a simpler synthesis procedure and also a better understanding. We will also consider them as models of concurrent systems and compare them with the asynchronous automata.

## 1.2 A Small Example

Just to provide an illustration of how the synthesis procedure works in our framework, we synthesize a solution for a simplified version of the Producer-Consumer problem. We will give a distributed alphabet together with a global specification of the problem over the chosen alphabet and then we automatically construct a distributed implementation for the given specification. Along with the example, we introduce the reader to some of the concepts used in the thesis.

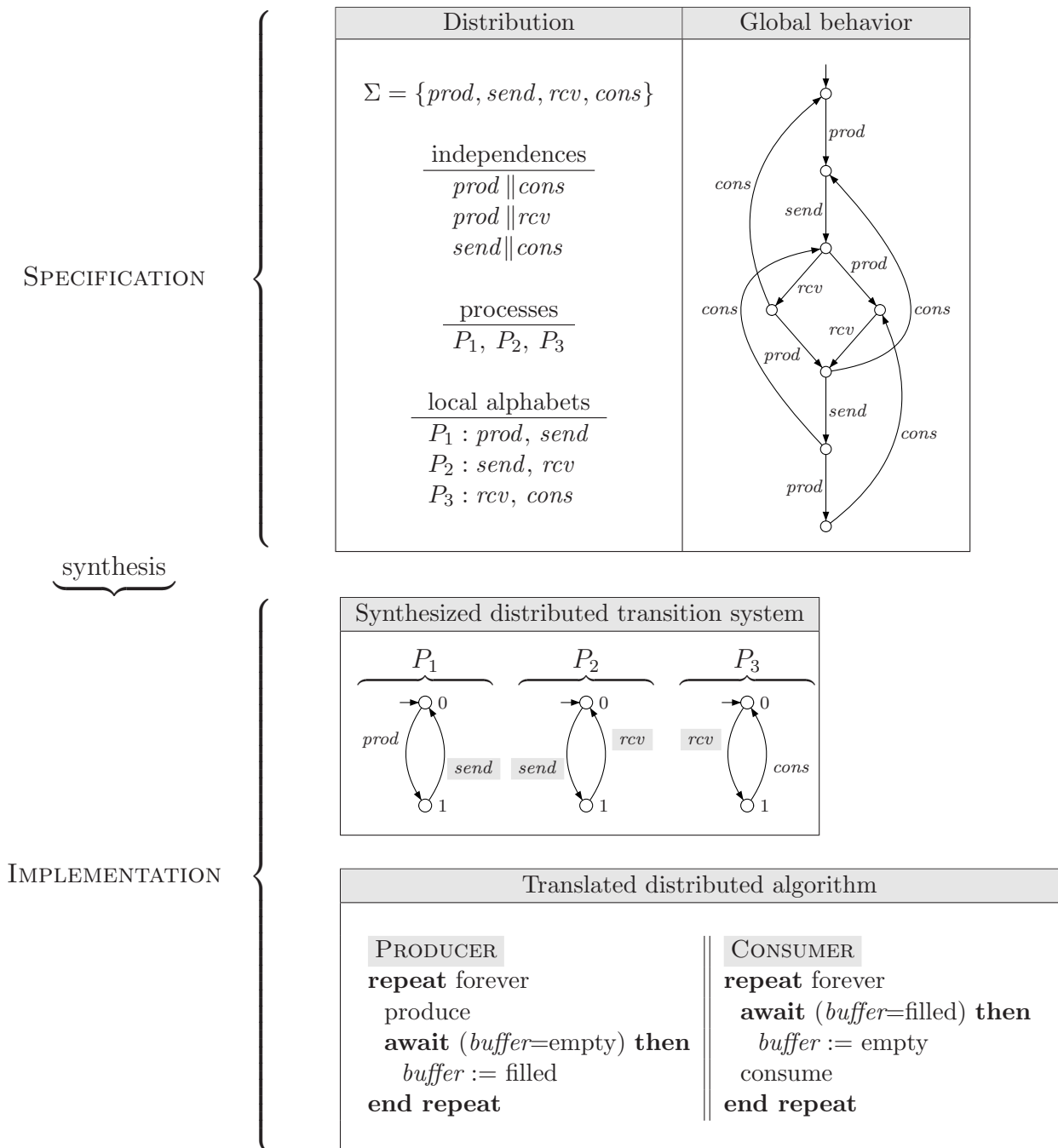


Figure 1.1: Synthesis at work for a simple Producer-Consumer problem

We first suppose that the actions involved in a trade between a *producer* and a *consumer* consist of the atomic actions of *producing*, *sending*, *receiving*, and *consuming*. So, first we fix the alphabet:

$$\Sigma = \{prod, send, rcv, cons\}.$$

Next, we specify a binary *independence* relation over the alphabet  $\Sigma$ :

$$\parallel \subseteq \Sigma \times \Sigma.$$

For instance, the independence relation is used to enforce that the actions of producing and consuming can and will be executed in parallel in the intended distributed implementation. A natural choice for our problem is to require that the following pairs of actions are independent:

$$prod \parallel rcv, \quad prod \parallel cons, \quad send \parallel cons.$$

The complement with respect to  $\Sigma \times \Sigma$  of the independence relation is called the *dependence* relation and is denoted by  $\not\parallel$ . Hence, we have  $\not\parallel = (\Sigma \times \Sigma) \setminus \parallel$ . The dependence relation is usually visualized using its associated *dependence graph*. The dependence graph is an undirected graph with the actions of  $\Sigma$  as vertices and the dependence relation  $\not\parallel$  as the edge relation. For our example, the dependence graph is:

$$\begin{array}{cc} prod & cons \\ | & | \\ send & \text{---} rcv \end{array}$$

Since we want to synthesize a distributed system, we need to specify also a *distribution pattern* that our distributed implementation must comply with. More precisely, we give a set of names of processes (or agents), denoted by  $Proc$ , together with a local alphabet  $\Sigma_{loc}(p) \subseteq \Sigma$  associated to each process  $p \in Proc$ . Once we have an independence relation we can generate a distribution pattern such that two actions are independent if and only if there is no process containing both actions in his local alphabet. This can be achieved by choosing the local alphabets to be a covering by (maximal) cliques<sup>1</sup> of the dependence graph. For our example, the only covering by cliques of the dependence graph is given by choosing one clique for each edge. This means, we have three processes

$$Proc := \{P_1, P_2, P_3\}$$

with the local alphabets

$$\begin{aligned} \Sigma_{loc}(P_1) &:= \{prod, send\}, \\ \Sigma_{loc}(P_2) &:= \{send, rcv\}, \\ \Sigma_{loc}(P_3) &:= \{rcv, cons\}. \end{aligned}$$

The core of the synthesis problem is to associate a local (labeled) transition system to each process name such that the global synchronization on common actions of all the local transition systems is consistent with a given global behavior. For our example, we choose a global (regular) behavior as follows:

$$Spec := \text{Prefix}(S_1) \cap \text{Prefix}(S_2),$$

where

---

<sup>1</sup>A *clique*  $C$  of a graph  $G = (V, E)$  is by definition a subset of vertices  $C \subseteq V$  such that  $C \times C \subseteq E$ .

- $S_1 = \text{Shuffle}((\text{prod} \cdot \text{send})^*, (\text{rcv} \cdot \text{cons})^*)$ ,  
i.e., we allow the interleaving/shuffle of the behaviors of the ‘producer’ and ‘consumer’,
- $S_2 = (T^* \cdot \text{send} \cdot T^* \cdot \text{rcv} \cdot T^*)^*$ , with  $T = \Sigma \setminus \{\text{send}, \text{rcv}\}$ ,  
i.e., the consumer can only receive something that was previously sent, and
- $\text{Prefix}(L)$  is a notation for the prefix-closure of the language  $L$ ,  
i.e., the set of all the prefixes of the words of  $L$ .  
(We observe all the partial runs of the system.)

The upper part of Figure 1.1 shows all the elements of the specification, i.e., the distribution pattern (top-left position) and the transition system exhibiting the global behavior<sup>1</sup> described above (top-right position).

The synthesis procedure will test whether the specification can be distributed and, if this is the case, will construct a distributed implementation (see also Figure 1.2). For simplicity, we choose as model of distributed implementation the *synchronous products of transition systems* [Arn94]. Informally, a synchronous product of transition systems over a given distribution consists of a set of local transition systems associated to each process which synchronize on *common actions*. For a given global specification, it is decidable to check whether there exists a synchronous product of transition systems accepting the same behavior as the global specification. The idea behind the distributability test is that of *projection* onto the local alphabets. More precisely, for each process  $p$  we construct a projection of the global specification onto the local alphabet  $\Sigma_{loc}(p)$  (that is, we ignore all the actions not in  $\Sigma_{loc}(p)$  by turning them into  $\varepsilon$ 's). Then, we have that the specification is distributable if and only if the synchronization on common actions of the projections is behaviorally equivalent to the specification.

For our example, it turns out that the specification is indeed distributable. The projections of the specification onto the three local alphabets are depicted in the middle of Figure 1.1. It can be easily verified that their global synchronization on common actions has the same behavior as the given specification. The distributed implementation will then be the synchronous product of the three local component each of them having two local states.

For instance, the run  $\text{prod} \cdot \text{send} \cdot \text{rcv} \cdot \text{cons}$  can be executed by the synchronous product in the following way: We start in the global initial state  $(0, 0, 0)$ . First,  $P_1$  can locally execute a *prod* action and change its local state to 1, so the new global state is  $(1, 0, 0)$ . Then,  $P_1$  and  $P_2$  can synchronize on the their *send* actions (marked in the picture) changing in the same step their local states and we obtain  $(0, 1, 0)$ . Similarly,  $P_2$  and  $P_3$  are both able to execute a *rcv* action, so they synchronize changing the global state to  $(0, 0, 1)$ . Finally,  $P_3$  will locally execute a *cons* action, and by doing so, the distributed system returns to its initial global state  $(0, 0, 0)$ .

Once we have a distributed implementation, we can go further and translate the distributed transition system into other formalisms. At the bottom of Figure 1.1 we show such a possible translation that interprets  $P_1$  and  $P_3$  as two processes PRODUCER and

<sup>1</sup>Here by the *behavior* of a system we understand the set of all possible runs (sequences of consecutive actions) from the initial state of the system.

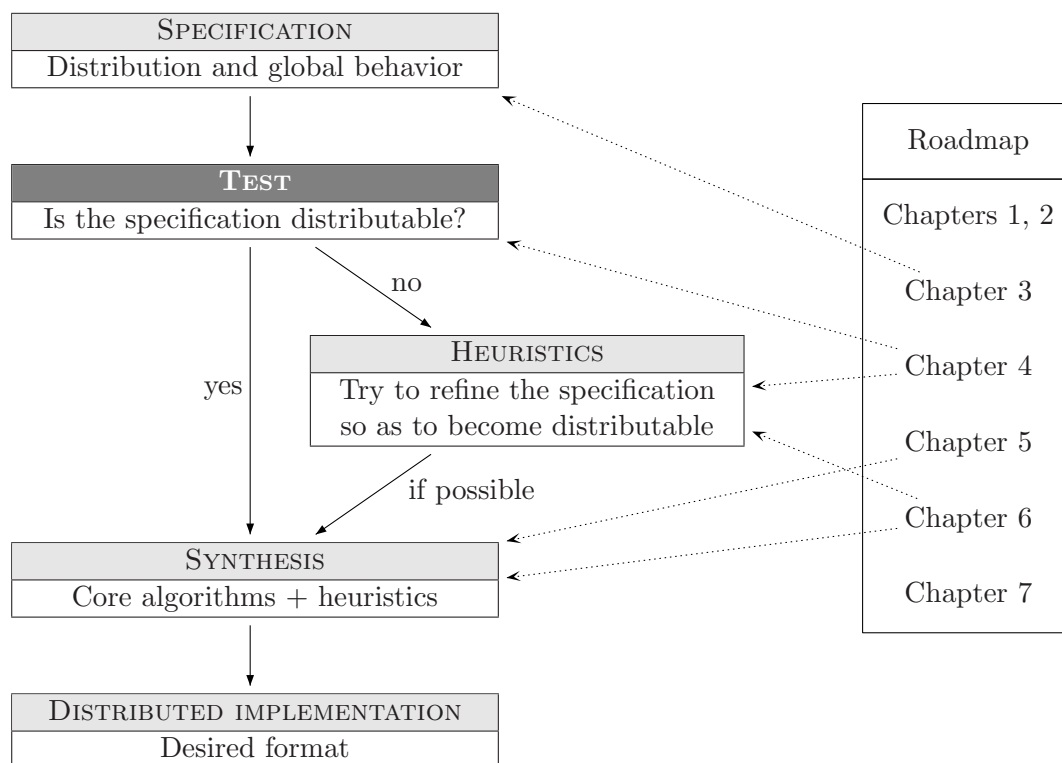


Figure 1.2: A diagrammatic flow of the synthesis approach followed in this thesis

CONSUMER communicating via  $P_2$  which models a buffer of capacity 1 (the state 0 of  $P_2$  reads ‘buffer empty’, while state 1 reads ‘buffer filled’; we assume that the buffer is initially empty). The PRODUCER process will loop forever through the sequence of the following atomic actions: First it produces, then it waits for the buffer to be emptied (i.e.,  $P_2$  moves to state 0). When the buffer is empty, the PRODUCER will fill it (note that we abstracted away the information/material being traded between the PRODUCER and the CONSUMER). On the opposite site, the CONSUMER will wait until the capacity 1 buffer is filled and only thereafter will empty it. Then, it will locally execute its ‘consume’ action.

### 1.3 The Contribution of this Thesis

The structure of the thesis follows the synthesis flow proposed in Figure 1.2. We discuss it below profiling some outcomes of our work:

**Chapter 2** We start with the prerequisites and some notations.

**Chapter 3** Then, we introduce the theoretical models (traces, asynchronous automata, synchronous products), together with various properties and characterizations used in the following chapters for the synthesis problem.

**Chapter 4** There exists the possibility that the given specification cannot be distributively implemented (this may be due to human error or incomplete specifications).



The first major step is to check whether the two parts of the specification (the distribution and the global behavior) are consistent, i.e., test whether there exists indeed a distributed implementation for the given specification.

We study this implementability test from computational complexity point of view and offer some heuristics to deal with the case when the test fails (i.e., we try to shape the initial global specification taking concurrency into account).

**Chapter 5** Assuming the implementability test is positive, the next step is the synthesis procedure itself.

In some cases, checking the implementability of the specification may give a distributed implementation for free (e.g., for synchronous products). However, in other cases we have to go through the notoriously hard construction of Zielonka. Although we invested a great deal in trying to simplify this procedure, we managed to offer some alternatives only in some special cases.

**Chapter 6** We have prototype implementations for most of the algorithms proposed in the thesis. We used several heuristics that try to construct smaller asynchronous automata, which worked very well on the benchmarks used.

In particular, we *automatically* synthesized new distributed algorithms for classical problems like mutual exclusion and dining philosophers.

**Chapter 7** We end the thesis with a short account on what was achieved and possible improvements.

**Publications** Some of the results included in this thesis appeared already in the following papers: [Ste02, SEM03, HŞ04, HŞ05].





---

I will not define time, space, place  
and motion, as being well known to  
all.

*Isaac Newton*  
(Principia Mathematica)

## CHAPTER 2

# PRELIMINARIES

---

IN this chapter, we present definitions and results frequently used in the rest of the thesis. Proofs will be sparsely given and those presented are there mainly for pedagogical reasons. For the missing proofs, the reader is referred to the cited papers.

We start with basic notions and notations (from set, formal language, and graph theory) in Section 2.1. In Section 2.2 we introduce the classic concept of transition system and its accepting language together with some properties needed in the subsequent chapters. We present the finite automata as transition systems with a refined acceptance condition.

The index at the end of the thesis will help at a later time to relocate the definitions and notations of this chapter.

### 2.1 Basic Notions and Notations

In this section we provide most of the basic notation conventions used throughout the thesis. We start with basic set theory. We continue with basic definitions on the algebraic structure of monoids. Then we introduce the basic concepts of formal languages over alphabets of actions used to generate words (alias executions) and languages (alias behaviors). We end with some graph theoretical concepts needed by the exposition.

#### 2.1.1 Set Theory

When dealing with *sets*, we apply the following usual conventions taken from the classic set theory:

- The names for the sets start with a capital letter ( $A, B, C, \dots$ ), while their elements with small letters ( $a, b, c, \dots$ ). We use indices ( $a_1, a_2, a_3, \dots$ ) or the prime symbol ( $a'$ ) to distinguish between elements of the same type.
- The *cardinality* of a given set  $S$ , i.e., the number of distinct elements of  $S$ , is denoted by  $|S|$ .
- The *operations* on sets are respectively denoted as follows:

- The *empty set* is denoted by  $\emptyset$ .
- The *union* of two sets is denoted by  $\cup$ .
- The *intersection* of two sets is denoted by  $\cap$ .
- The *strict* or *proper* inclusion of one set into another is denoted by  $\subsetneq$ .
- The *cartesian product* of two sets is denoted by  $\times$ . Given a set  $I$  of indices and a family  $(S_i)_{i \in I}$  of sets, the cartesian product of them is denoted by  $\prod_{i \in I} S_i$ .
- The *difference* of two sets is denoted by  $\setminus$ . (By definition,  $A \setminus B := \{x \in A \mid x \notin B\}$ .)
- If we suppose that a set  $B$  is a subset of a set  $A$ , then the *complement* of  $B$  with respect to  $A$  is defined as  $A \setminus B$  and denoted by  $\complement B$  when  $A$  can be deduced from the context or directly  $A \setminus B$ , otherwise.
- The *power set* of a set  $A$ , i.e., the set of all subsets of  $A$ , is denoted by  $\mathcal{P}(A)$ .
- The *set of all functions* from a set  $A$  to a set  $B$  is denoted by  $B^A$ .
- A *singleton* is a set with only one element,  $X = \{x\}$ . To simplify notation, we may sometimes drop the braces around  $x$ .

Given a class of sets  $\mathcal{C}$  and an operation on sets  $\mathcal{O}$ , we say that the class  $\mathcal{C}$  is *closed* under operation  $\mathcal{O}$  if and only if the result of the application of  $\mathcal{O}$  to the elements of  $\mathcal{C}$  is still in  $\mathcal{C}$ . For example, the class of sets with at most 3 elements is closed under intersection, but not under union.

- The set of *natural numbers* is denoted by  $\mathbb{N}$ . For two natural numbers  $i, j \in \mathbb{N}$  with  $i \leq j$ , we denote by  $[i..j]$  the set  $\{k \mid i \leq k \leq j\}$ .
- A (binary) *relation* is a set of pairs. If  $R$  is a relation and  $(a, b)$  is a pair in  $R$ , then we mostly write  $aRb$ .

Suppose  $\mathcal{P}$  is a set of properties of relations. The  $\mathcal{P}$ -*closure* of a relation  $R$  is the smallest relation  $R'$  that includes all the pairs of  $R$  and possesses the properties in  $\mathcal{P}$ .

For example, the *transitive closure* of  $R$ , denoted  $R^+$ , is defined by:

- 1) If  $(a, b)$  is in  $R$ , then  $(a, b)$  is in  $R^+$ .
  - 2) If  $(a, b)$  is in  $R^+$  and  $(b, c)$  is in  $R$ , then  $(a, c)$  is in  $R^+$ .
  - 3) Nothing is in  $R^+$  unless it so also follows from (1) and (2).
- An *equivalence* relation on a set  $S$  is a binary relation  $\sim \subseteq S \times S$  that satisfies the properties of *reflexivity* ( $\forall a \in S : a \sim a$ ), *symmetry* ( $\forall a, b \in S : a \sim b \Rightarrow b \sim a$ ), and *transitivity* ( $\forall a, b, c \in S : a \sim b$  and  $b \sim c \Rightarrow a \sim c$ ). For a set  $S$  and an equivalence  $\sim$  on it, we have:
    - The *equivalence class* of an element  $a \in S$  is defined as  $\{b \in S \mid b \sim a\}$  and denoted by  $[a]_{\sim}$ .

- The *quotient set* of  $S$  by  $\sim$  is defined as the set of all equivalence classes of  $\sim$ , that is,  $\{[a]_\sim \mid a \in S\}$  and is denoted by  $S/\sim$ .
- An equivalence  $\sim$  on  $S$  is said to have *finite index*, if the quotient set  $S/\sim$  is finite.
- A *partial order* relation on a set  $S$  is a binary relation  $\sqsubseteq$  on  $S$  that satisfies the properties of *reflexivity* ( $\forall a \in S : a \sqsubseteq a$ ), *antisymmetry* ( $\forall a, b \in S : a \sqsubseteq b$  and  $b \sqsubseteq a \Rightarrow a = b$ ), and *transitivity* ( $\forall a, b, c \in S : a \sqsubseteq b$  and  $b \sqsubseteq c \Rightarrow a \sqsubseteq c$ ).

A *partially ordered set*, alias *poset*, is a set equipped with a partial order relation.

### 2.1.2 Algebraic Notions

A *monoid* is a tuple  $(M, \cdot, 1)$ , where  $M$  is a set,  $\cdot$  is a associative binary operation, and  $1 \in M$  is the *identity* element. (When confusion does not arise, we can use  $xy$  to indicate  $x \cdot y$ .)

A *congruence relation*  $\sim$  over a monoid  $(M, \cdot, 1)$  is an equivalence over  $M$  that is compatible with the binary operation of the monoid, i.e.,  $\forall x, y, x', y' \in M : x \sim x'$  and  $y \sim y' \Rightarrow xy \sim x'y'$ . The *quotient* of a monoid  $(M, \cdot, 1)$  under a congruence  $\sim$  is the monoid  $(M/\sim, \circ, [1]_\sim)$ , where  $[x]_\sim \circ [y]_\sim := [x \cdot y]_\sim$ . (The fact that the equivalence  $\sim$  is a congruence is used to prove that the given definition is well-defined.)

A *morphism*  $\phi$  from a monoid  $(M, \cdot_M, 1_M)$  to another monoid  $(S, \cdot_S, 1_S)$  is a function  $\phi : M \rightarrow S$  such that  $\phi(1_M) = 1_S$  and  $\phi(x \cdot_M y) = \phi(x) \cdot_S \phi(y)$ , for any  $x, y \in M$ . The surjective morphism  $[\ ]_\sim : M \rightarrow M/\sim$  that associates to every element  $x \in M$  its equivalence class  $[x]_\sim$  is called the *canonical morphism*.

A subset  $X \subseteq M$  is *closed under the congruence*  $\sim$  if and only if  $X$  is the union of some equivalence classes of  $\sim$ . Stated differently,  $X \subseteq M$  is *closed under the congruence*  $\sim$  if and only if  $\forall x, y \in M : x \in X$  and  $x \sim y \Rightarrow y \in X$ .

A congruence  $\sim$  is said to have *finite index* if the quotient  $M/\sim$  is a finite set.

### 2.1.3 Formal Languages

We describe the behavior of a sequential system by means of the classical theory of *formal languages*. The ingredients of this formalism are enumerated below:

- First of all, we have a non-empty finite set called the *alphabet* of *actions*, which we will usually denote by  $\Sigma$ .
- Then, we have the notion of a (finite) *word* or *execution*<sup>1</sup> over  $\Sigma$ , which is a finite sequence of actions from the alphabet  $\Sigma$ . A word is represented by the *concatenation* of its actions. E.g.,  $w = a_1 a_2 \dots a_n$  with  $a_i \in \Sigma$  for all  $i \in [1..n]$ . The concatenation of two words will again be a word. Furthermore, we have:
  - We denote the concatenation of  $n$  copies of a word  $w \in \Sigma^*$  as usually by  $w^n$ .
  - The *length* of a word  $w$  is denoted by  $|w|$ , i.e., if  $w = a_1 a_2 \dots a_n$ , then  $|w| = n$ .

<sup>1</sup>In this thesis, we will not consider the case of *infinite* words/executions.

- A special word is the *empty word*, denoted by  $\varepsilon$ , which is the word of length 0.
- For  $w \in \Sigma^*$  and  $a \in \Sigma$ , we denote by  $\#_a w$  the number of occurrences of the action  $a$  in the word  $w$ .

For  $w \in \Sigma^*$ , we denote by  $\Sigma(w)$  the *alphabet* of the word  $w$ , defined as the set of actions appearing in  $w$ , i.e.,

$$\Sigma(w) := \{a \in \Sigma \mid \#_a w > 0\}.$$

Obviously,  $\Sigma(w) \subseteq \Sigma$ .

- For a word  $w \in \Sigma^*$  and a subset  $S \subseteq \Sigma$ , we denote by  $w \upharpoonright_S$  the *projection* of  $w$  onto  $S$ , which is obtained by erasing all actions in  $w$  which do not belong to  $S$ .

Formally, for  $S \subseteq \Sigma$ , we define a function  $\upharpoonright_S: \Sigma \rightarrow \Sigma$  such that  $a \upharpoonright_S := \varepsilon$  if  $a \notin S$  and  $a \upharpoonright_S := a$  if  $a \in S$  (we ‘erase’ the elements not belonging to  $S$  by replacing them with  $\varepsilon$ ’s which are eventually absorbed). This function can be lifted to words  $\upharpoonright_S: \Sigma^* \rightarrow \Sigma^*$  by the natural recursive definition:  $\varepsilon \upharpoonright_S := \varepsilon$  and  $(xa) \upharpoonright_S := x \upharpoonright_S \cdot a \upharpoonright_S$  for all  $x \in \Sigma^*$  and  $a \in \Sigma$ .

- For two words  $t, u \in \Sigma^*$ , we denote by  $\text{Shuffle}(t, u)$  the *shuffle product* of the two words  $t$  and  $u$ , which is the operation that constructs all the interleavings of all possible ‘cuttings’ of each of the two words. Formally,

$$\text{Shuffle}(t, u) := \left\{ t_1 u_1 t_2 u_2 \dots t_n u_n \mid \begin{array}{l} t = t_1 \dots t_n \text{ and } u = u_1 \dots u_n, \\ \text{where } n \geq 1 \text{ and } t_i, u_i \in \Sigma^*, \forall i \in [1..n] \end{array} \right\}.$$

Diagrammatically,

$$\left. \begin{array}{l} t = \boxed{t_1} \cdots \boxed{t_k} \cdots \boxed{t_n} \\ u = \boxed{u_1} \cdots \boxed{u_k} \cdots \boxed{u_n} \end{array} \right\} \Rightarrow \boxed{t_1} \boxed{u_1} \cdots \boxed{t_k} \boxed{u_k} \cdots \boxed{t_n} \boxed{u_n} \in \text{Shuffle}(t, u).$$

For example, we have  $\text{Shuffle}(ab, cd) = \{abcd, acbd, acdb, cabd, cadb, cdab\}$  and  $\text{Shuffle}(ab, a) = \{aba, aab\}$ . Note that in the above definition of  $\text{Shuffle}$ ,  $t_i, u_i \in \Sigma^*$ , which means they might even be  $\varepsilon$ . For instance,  $cdab \in \text{Shuffle}(ab, cd)$  by choosing  $k = 2$ ,  $t_1 = \varepsilon$ ,  $t_2 = ab$ ,  $u_1 = cd$ , and  $u_2 = \varepsilon$ .

- The word  $v \in \Sigma^*$  is called a *prefix* of the word  $w \in \Sigma^*$  if and only if there exists another word  $v' \in \Sigma^*$  such that  $vv' = w$ . Dually,  $v'$  is called a *suffix* of  $w$  if and only if there exists  $v$  such that  $vv' = w$ . (Note that a word  $w$  is always a prefix of itself. Similarly,  $w$  is always the suffix of itself.)

- The set of *all* words over  $\Sigma$  is denoted by  $\Sigma^*$ . In algebraic terms, the set  $\Sigma^*$  together with the operation of word concatenation and the empty word  $\varepsilon$  form a monoid. In fact,  $\Sigma^*$  is the *free monoid* generated by  $\Sigma$ .
- A set of words over  $\Sigma$  (i.e., a subset of  $\Sigma^*$ ) is called a *language* over  $\Sigma$ . If we look at words as executions of a given system able to execute actions from an alphabet  $\Sigma$ , then a language over  $\Sigma$  can describe a behavior of the system.

We can extend some of the operations on words to languages in the following way:

- For two languages  $L_1, L_2 \subseteq \Sigma^*$ , we define their concatenation as follows:

$$L_1 L_2 := \{w_1 w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}.$$

For  $L \subseteq \Sigma^*$  and a natural number  $n \geq 1$ , we denote by  $L^n$  the concatenation of  $n$  copies of the language  $L$ . For  $n = 0$ , by definition we choose  $L^0 := \{\varepsilon\}$ .

Also, the union  $L_1 \cup L_2$  and intersection  $L_1 \cap L_2$  of the two languages  $L_1$  and  $L_2$  are defined as the union, respectively intersection operations on sets (of words).

The complement of a language  $\mathcal{C}L$  is defined as the complement with respect to  $\Sigma^*$ , i.e.,  $\mathcal{C}L := \Sigma^* \setminus L$ .

Finally, we denote by  $L^*$  the (Kleene) *iteration* of  $L$  which is defined as:

$$L^* := \bigcup_{n \geq 0} L^n.$$

- For  $L \subseteq \Sigma^*$ , we denote by  $\Sigma(L)$  the *alphabet* of the language  $L$ , which is defined as the set of actions appearing in the words of  $L$ :

$$\Sigma(L) := \bigcup_{w \in L} \Sigma(w).$$

Obviously,  $\Sigma(L) \subseteq \Sigma$ .

- For  $L \subseteq \Sigma^*$  and a subset  $S \subseteq \Sigma$ , we denote by  $L \upharpoonright_S$  the *projection* of  $L$  onto  $S$ , which is given by:

$$L \upharpoonright_S := \{w \upharpoonright_S \mid w \in L\}.$$

- For  $L_1, L_2 \subseteq \Sigma^*$ , we denote by  $\text{Shuffle}(L_1, L_2)$  the *shuffle product* of the languages  $L_1$  and  $L_2$  which is defined as:

$$\text{Shuffle}(L_1, L_2) := \bigcup_{w_1 \in L_1, w_2 \in L_2} \text{Shuffle}(w_1, w_2).$$

- For  $L \subseteq \Sigma^*$ , we denote by  $\text{Prefix}(L)$  the *prefix-closure* of  $L$ , which is the set of all prefixes of the words of  $L$ :

$$\text{Prefix}(L) := \{v \in \Sigma^* \mid \text{there exists } w \in L \text{ such that } v \text{ is a prefix of } w\}.$$

A language  $L \subseteq \Sigma^*$  is called *prefix-closed* if and only if  $L = \text{Prefix}(L)$ .

It is easy to see that a language is prefix-closed if and only if for any  $v, w \in \Sigma^*$ , if  $w \in L$  and  $v$  is a prefix of  $w$ , then also  $v \in L$ . In particular,  $\varepsilon \in \text{Prefix}(L)$  for any language  $L$ .

Moreover, the language  $\text{Prefix}(L)$  coincides with the smallest (w.r.t. set inclusion) prefix-closed language including  $L$ .

Since we want to observe the behaviors of systems in all the intermediary steps, we will mainly work with prefix-closed languages, because they keep track of all the prefixes of each execution. The following proposition shows that the class of prefix-closed languages behaves well with respect to the operations introduced so far:

**Proposition 2.1** *The class of prefix-closed languages over a given alphabet  $\Sigma$  is closed under the operations of union, intersection, concatenation, iteration, projection, shuffle product, and prefix-closure. The class of prefix-closed languages is not closed under complementation.*

**Proof.** For the union and intersection cases, the proof is obvious: If  $L_1, L_2 \subseteq \Sigma^*$  are two prefix-closed language, then both  $L_1 \cup L_2$  and  $L_1 \cap L_2$  are also prefix-closed. In fact, it is true that the union (respectively intersection) of an infinite number of prefix-closed languages is also prefix-closed.

For the concatenation case, we show that for any two languages  $L_1, L_2 \subseteq \Sigma^*$  that are prefix-closed, we have that  $L_1L_2$  is also prefix-closed. Let  $w \in L_1L_2$  and  $v$  a prefix of  $w$ . From the fact that  $w \in L_1L_2$ , we have that there exist  $w_1 \in L_1$  and  $w_2 \in L_2$  such that  $w = w_1w_2$ . If  $v$  is a prefix of  $w$ , we have two cases: either  $v$  is a prefix of  $w_1$  or  $v = w_1v'$  with  $v'$  a prefix of  $w_2$ . In the first case,  $v \in L_1$  because  $w_1 \in L_1$  and  $L_1$  is prefix-closed. Since  $L_2$  is prefix-closed, we have  $\varepsilon \in L_2$ . From  $v_1 \in L_1$ ,  $\varepsilon \in L_2$ , and the definition of the concatenation operation, we have indeed that  $v = v\varepsilon \in L_1L_2$ . In the second case, since  $v'$  is a prefix of  $w_2 \in L_2$  and  $L_2$  is prefix-closed, we have that  $v' \in L_2$ . From this fact and  $w_1 \in L_1$ , we deduce that  $v = w_1v' \in L_1L_2$ .

For the iteration case, we show that for any prefix-closed language  $L \in \Sigma^*$ , we have that  $L^*$  is also prefix-closed. We can first prove  $L^n$  is a prefix-closed language for any natural number  $n \geq 0$ . The proof is by induction on  $n$ , using the fact that the class of prefix-closed is closed under concatenation. Using this,  $L^*$  will be prefix-closed because the union of all the prefix-closed languages  $L^n$  with  $n$  ranging over  $\mathbb{N}$  is also prefix-closed (the class of prefix-closed languages is closed under (infinite) union).

For the projection case, we show that for any prefix-closed language  $L \subseteq \Sigma^*$  over  $\Sigma$  and any subset  $S$  of  $\Sigma$ , we have that  $L \upharpoonright_S$  is also prefix-closed. Let  $w \in L \upharpoonright_S$  and  $v$  a prefix of  $w$ . From  $w \in L \upharpoonright_S$ , we have that there exists a word  $u \in L$  such that  $w = u \upharpoonright_S$  ( $w$  is obtained from  $u$  by keeping only the action from  $S$  and removing all the others). Let  $a$  be the last action of  $v$  and  $k := \#_a(v)$  the number of the occurrences of  $a$  in  $v$ . We choose  $u'$  to be the *shortest* prefix of  $u$  that contains *exactly*  $k$  occurrences of  $a$ . It is not difficult to show that  $v = u' \upharpoonright_S$  (using also the fact that  $w = u \upharpoonright_S$ ). Since  $L$  is a prefix-closed language,  $u \in L$ , and  $u'$  is a prefix of  $u$ , we have that  $u' \in L$ , which implies that  $v \in L \upharpoonright_S$ .

For the shuffle product case, we show that for any two languages  $L_1, L_2 \subseteq \Sigma^*$  that are prefix-closed, we have that  $\text{Shuffle}(L_1, L_2)$  is also prefix-closed. Let  $w \in \text{Shuffle}(L_1, L_2)$  and  $v$  a prefix of  $w$ . From  $w \in \text{Shuffle}(L_1, L_2)$ , by definition, there exist  $t \in L_1$  and  $u \in L_2$  such that  $w \in \text{Shuffle}(t, u)$ . Further, this means that there exist a natural number  $k \geq 1$  together with the words  $t_i, u_i \in \Sigma^*$  for  $i \in [1..k]$  such that  $t = t_1 \dots t_k$ ,  $u = u_1 \dots u_k$ , and  $w = t_1u_1 \dots t_ku_k$ . By hypothesis,  $v$  is a prefix of  $w = t_1u_1 \dots t_ku_k$ . Let  $j$  be the *smallest* index from  $[1..k]$  such that  $v$  is a prefix of  $t_1u_1 \dots t_ju_j$ . This implies that  $v = t_1u_1 \dots t_{j-1}u_{j-1}v_0$ , where  $v_0$  is a prefix of  $t_ju_j$  (just in case, we choose by definition  $t_0 = u_0 = \varepsilon$ ). Performing an analysis similar to the concatenation case above, we can decompose  $v$  as  $v_0^t v_0^u$  such that  $v_0^t$  is a prefix of  $t_j$  and  $v_0^u$  is a prefix of  $u_j$ . This means that  $v = t_1u_1 \dots t_{j-1}u_{j-1}v_0^t v_0^u$ , which further implies that  $v \in \text{Shuffle}(t_1 \dots t_{j-1}v_0^t, u_1 \dots u_{j-1}v_0^u)$ . Using the fact that  $L_1$  and  $L_2$  are prefix-closed in conjunction with property that  $v_0^t$  is a prefix of  $t_j$  and  $v_0^u$  a prefix of  $u_j$ , we conclude that  $v \in \text{Shuffle}(L_1, L_2)$ .

For the prefix-closure case, we must prove that if  $L \subseteq \Sigma^*$  is a prefix-closed language, then  $\text{Prefix}(L)$  is also prefix-closed. But the very idea of the prefix-closure operation is to construct a prefix-closed language from a given language, in particular also from a prefix-closed one. Formally, it is easy to prove that the prefix-closure operation is idempotent, i.e.,  $\text{Prefix}(\text{Prefix}(L)) = \text{Prefix}(L)$  for any language  $L$ , and, by definition, this means that  $\text{Prefix}(L)$  is prefix-closed for any  $L$ .

For the complementation case, we can prove in fact that for any prefix-closed language  $L \subseteq \Sigma^*$ , the complement of  $L$  is not prefix-closed anymore. More precisely, if  $L$  is a prefix-closed language, then  $\varepsilon \in L$ , which implies that  $\varepsilon \notin \mathcal{C}L$ . But any prefix-closed language contains the empty word  $\varepsilon$ , which necessarily implies that  $\mathcal{C}L$  is not prefix-closed.  $\square$

### Regular Languages

A ‘first-class citizen’ of the formal language theory is the class of *regular languages* (see for instance [HU79, Chapter 2]):

#### Definition 2.2 (Regular language)

The class of *regular languages* over an alphabet  $\Sigma$  is the smallest class containing the empty set  $\emptyset$ , the singleton languages  $\{\varepsilon\}$  and  $\{a\}$  for all  $a \in \Sigma$ , and being closed under the operations of union, concatenation, and Kleene iteration. The class of *regular languages* over  $\Sigma$  will be denoted by  $\text{Reg}(\Sigma)$ .

A language is called *regular* if it belongs to  $\text{Reg}(\Sigma)$ .

The regular languages can be represented by *regular expressions*. The latter show the order in which the three operations union, concatenation, and iteration are applied to the finite languages in order to obtain a given regular language.

The *regular expressions* over an alphabet  $\Sigma$  and the languages they denote are recursively defined as follows:

1.  $\emptyset$  is a regular expression and denotes the empty language.
2.  $\varepsilon$  is a regular expression and denotes the language  $\{\varepsilon\}$ .
3. For each  $a \in \Sigma$ , the symbol  $a$  is a regular expression and denotes the language  $\{a\}$ .
4. If  $r$  and  $s$  are regular expressions denoting the languages  $R$  and  $S$ , respectively, then  $(r + s)$ ,  $(r \cdot s)$ , and  $(r^*)$  are regular expressions that denote the languages  $R \cup S$ ,  $R \cdot L$ , and  $R^*$ , respectively.

In writing regular expression we can omit many parentheses if we assume that  $*$  has the highest precedence and  $+$  the lowest.

If we denote by  $L(r)$  the language of the regular expression  $r$ , then we have

$$\text{Reg}(\Sigma) = \{L(r) \mid r \text{ regular expression over } \Sigma\}.$$

**Example 2.3** In this thesis, we will mainly use regular languages for the specification of the global behavior of a system. For instance, the regular expression

$$(\text{send} \cdot \text{rcv})^*$$

denotes a very simple communication pattern, where a message is *sent* (by a party), and then instantaneously *received* (by a peer party), this play may be repeating at leisure.



Another classic characterization of regular languages is based on the notion of *syntactic congruence*: For a language  $L \subseteq \Sigma^*$ , we denote by  $\sim_L \subseteq \Sigma^* \times \Sigma^*$  the *syntactic congruence* of  $L$  on  $\Sigma^*$ , defined as follows: For  $v, w \in \Sigma^*$ ,

$$v \sim_L w \text{ if and only if } \forall x, y \in \Sigma^* : xvy \in L \Leftrightarrow xwy \in L.$$

The characterization of regular languages based on the syntactic congruence is:

**Theorem 2.4** [Lal79, Chapter 6] *A language  $L \subseteq \Sigma^*$  is regular if and only if the syntactic congruence  $\sim_L$  is of finite index.*

The following proposition recalls the property of the class of regular languages of being closed under all the language operations introduced so far:

**Proposition 2.5** *The class of regular languages over a given alphabet  $\Sigma$  is closed under the operations of union, intersection, concatenation, iteration, complementation, projection, shuffle product, and prefix-closure.*

**Proof.** Proofs for the closure under the operations of union, intersection, concatenation, iteration, and complementation can be found for instance in [HU79, Section 3.2].

The projection  $- \upharpoonright_S$  is a special case of homomorphism under which [HU79, Theorem 3.5] shows that the class of regular languages is closed.

The closure of the class of regular languages under shuffle product is mentioned in [HU79, Exercise 6.6] (with the solution involving closure under homomorphisms and inverse homomorphisms of the class of regular languages).

As far as the prefix-closure operation is concerned, if  $L \in \Sigma^*$  is a regular language, it is easy to show that also  $\text{Prefix}(L)$  is regular. We know that  $L$  is regular if and only if there exists a finite-state automata  $\mathcal{A}$  with a set of accepting states  $F$  such that  $L = L(\mathcal{A}, F)$  (see the notation of Definition 2.13). Then  $\text{Prefix}(L) = L(\mathcal{A}, F')$ , where  $F'$  is the set of all states of  $\mathcal{A}$ . Hence,  $\text{Prefix}(L)$  is also regular.  $\square$

Let us denote by  $\text{PrefReg}(\Sigma)$  the class of *prefix-closed regular languages* over a given alphabet  $\Sigma$ . Then, from Propositions 2.1 and 2.5 we have the following closure result:

**Corollary 2.6** *The class  $\text{PrefReg}(\Sigma)$  is closed under all the operations on languages introduced so far, except for the complementation operation.*

## 2.1.4 Graph Theory

We recall here some concepts of graph theory used throughout the thesis.

- A *graph*  $\mathcal{G}$  is a pair  $(V, E)$ , where  $V$  is a finite set of *vertices* or *nodes* and  $E \subseteq V \times V$  is the set of *edges* or *arcs*.

If  $E$  is symmetric, i.e.,  $(x, y) \in E$  implies  $(y, x) \in E$ , then the graph is said to be *undirected*, otherwise it is *directed*.



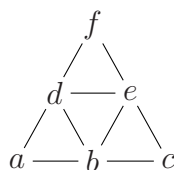


Figure 2.1: Example of a graph [Fig93a]

- A *subgraph*  $\mathcal{G}'$  of a graph  $\mathcal{G} = (V, E)$  is a pair  $(V', E')$  with  $V' \subseteq V$  and  $E' \subseteq E \cap (V' \times V')$ .

The subgraph  $\mathcal{G}'$  of  $\mathcal{G}$  *induced* by a subset of vertices  $V'$  of  $V$  is the graph  $(V', E \cap (V' \times V'))$ .

- A *path* from a node  $x$  to a node  $y$  in a graph  $\mathcal{G}$  is a sequence  $(v_0, v_1, \dots, v_n)$  of vertices such that  $x = v_0$ ,  $y = v_n$ , and  $(v_{i-1}, v_i) \in E$ , for all  $i \in [1..n]$ .

A *cycle* is a path with the initial and final node identical. A graph is called *acyclic* if it contains no cycles.

Two nodes  $x$  and  $y$  of a graph  $\mathcal{G}$  are *connected* if there exists a path between them in  $\mathcal{G}$ . A graph  $\mathcal{G}$  is called *connected* if and only if every pair of its nodes is connected.

A *connected component* of a graph  $\mathcal{G}$  is a maximal connected induced subgraph, i.e., a subgraph  $\mathcal{G}' = (V', E')$  with  $E' = E \cap (V' \times V')$  such that  $\mathcal{G}'$  is connected and for every node  $a \in V \setminus V'$  the subgraph of  $\mathcal{G}$  induced by  $V' \cup \{a\}$  is not connected.

- A *tree* is a graph in which any two nodes are connected by exactly one path. A *spanning tree* of a connected graph is a tree that includes every node of that graph.
- Two graphs  $\mathcal{G}_1 = (V_1, E_1)$  and  $\mathcal{G}_2 = (V_2, E_2)$  are *isomorphic* if there exists a bijection  $f : V_1 \rightarrow V_2$  between the sets of nodes that complies with the edge relations, i.e.,  $(v, v') \in E_1$  if and only if  $(f(v), f(v')) \in E_2$ .

- A *clique* of an undirected graph  $\mathcal{G} = (V, E)$  is a subset  $V'$  of  $V$  that induces a complete graph, i.e.,  $(x, y) \in E$  for all  $x, y \in V'$  with  $x \neq y$ .

A clique  $V'$  of a graph  $\mathcal{G} = (V, E)$  is *maximal* if every set  $V''$  such that  $V' \subsetneq V'' \subseteq V$  is not a clique.

A *clique cover* of  $\mathcal{G}$  is a family  $(V_1, \dots, V_n)$  of cliques of  $\mathcal{G}$  such  $V = \bigcup_{i=1}^n V_i$  and for every edge  $(x, y) \in E$  there exists an index  $i \in [1..n]$  such that  $(x, y) \in V_i$ .

For every undirected graph  $\mathcal{G} = (V, E)$ , there are two distinguished clique covers:

1. The clique cover consisting of cliques with at most two nodes. This cover contains the clique  $\{x, y\}$  for every pair of nodes  $x, y \in V$  such that  $(x, y) \in E$ , and the clique  $\{x\}$  for each *isolated* node  $x \in V$ , i.e.,  $x$  is not connected to any other node from  $V$ .

For the graph in Figure 2.1, this clique cover will consist of the sets:

$$\{a, b\}, \{a, d\}, \{b, c\}, \{b, d\}, \{b, e\}, \{d, e\}, \{d, f\}, \{e, f\}.$$

2. The clique cover consisting of all the maximal cliques of  $\mathcal{G}$ .

For the graph in Figure 2.1, this clique cover will consist of the sets:

$$\{a, b, d\}, \{d, b, e\}, \{e, b, c\}, \{d, e, f\}.$$

Note that if we leave out the clique  $\{d, b, e\}$ , we still have a clique cover. This observation means that we do not necessarily need to consider the set of *all* maximal cliques in order to have a cover.

In fact, it may be dangerous to consider the set of *all* maximal cliques as this set can be exponentially larger than the size of the graph: Take, for instance,  $\mathcal{G} = (V, E)$ , where  $V := \{x_1, \dots, x_n, y_1, \dots, y_n\}$  and  $E := V \times V \setminus \{\{x_i, y_i\} \mid i \in [1..n]\}$ . Then,  $\mathcal{G}$  has exactly  $2^n$  maximal cliques.

On the other hand, the problem of finding a covering with a *minimal* number of cliques is NP-complete [GJ79].

**Remark 2.7** If an undirected graph is *transitive* i.e.,  $(x, y) \in E$  and  $(y, z) \in E$  implies  $(x, z) \in E$ , then the maximal cliques of  $\mathcal{G}$  are disjoint and coincide with the connected components of  $\mathcal{G}$ .

## 2.2 Transition Systems

One of the most general theoretical models able to capture (regular) behaviors of systems is that of *transition system*. A transition system consists of a number of *states* and a number of *transitions* between states which describe in each state which is the next possible state that the system can move to. Moreover, we have one or more initial states. Formally, we have the following definition:

### Definition 2.8 (Labeled transition system)

A *labeled transition system* is a tuple  $TS = (Q, \Sigma, \rightarrow, I)$ , where

- $Q$  is the set of states ( $Q$  is also called the *state space*)
- $\Sigma$  is the alphabet of action labels,
- $\rightarrow$  is the transition relation, where  $\rightarrow \subseteq Q \times \Sigma \times Q$ , and
- $I$  is the (nonempty) set of initial states, where  $I \subseteq Q$ .

Given a transition system  $TS = (Q, \Sigma, \rightarrow, I)$ , we have:

- $TS$  is called *finite*, if the state space  $Q$  is finite.
- The *size* of  $TS$  is denoted by  $|TS|$  and is defined as the size of the state space  $|Q|$ .

- If  $(q, a, q') \in \rightarrow$ , we use the usual notation  $q \xrightarrow{a} q'$ .
- Let  $w = w_1 \dots w_n$  be a word over  $\Sigma$  with  $w_i \in \Sigma$  for all  $i \in [1..n]$ . For two states  $q, q'$ , we write  $q \xrightarrow{w} q'$  if there exist a path  $q_0, q_1, \dots, q_n \in Q$  such that  $q = q_0$ ,  $q' = q_n$ , and  $q_{i-1} \xrightarrow{w_i} q_i$  for all  $i \in [1..n]$ .

For two states  $q, q' \in Q$ , if there exist  $w \in \Sigma^*$  such that  $q \xrightarrow{w} q'$ , we say that  $q'$  is *reachable* from  $q$ , and dually,  $q$  is *co-reachable* from  $q'$ .

A transition system is called *reachable* if all the states are reachable from the set of initial states, that is,

$$\forall q \in Q \exists q_0 \in I, w \in \Sigma^* : q_0 \xrightarrow{w} q.$$

- For an equivalence relation  $\equiv \subseteq Q \times Q$  over the states of  $TS$ , the *quotient* of  $TS$  over  $\equiv$  is defined as  $TS/\equiv := (Q/\equiv, \Sigma, \rightarrow, I/\equiv)$ , where  $Q/\equiv, I/\equiv$  are quotient sets (cf. Section 2.1.1) and  $[q_1]_{\equiv} \xrightarrow{a} [q_2]_{\equiv}$  if and only if there exist  $q'_1, q'_2 \in Q$  such that  $q_1 \equiv q'_1, q_2 \equiv q'_2$ , and  $q'_1 \xrightarrow{a} q'_2$ .
- Two transition systems  $TS_1 = (Q_1, \Sigma, \rightarrow_1, I_1)$  and  $TS_2 = (Q_2, \Sigma, \rightarrow_2, I_2)$  over the same alphabet of actions are *isomorphic* if there exists a bijection  $f : Q_1 \rightarrow Q_2$  between the state spaces that preserves the initial states and the transitions, i.e.,  $f(I_1) = I_2$  and  $(q, a, q') \in \rightarrow_1$  if and only if  $(f(q), a, f(q')) \in \rightarrow_2$ .
- We denote by  $\Sigma(TS)$  the *alphabet* of the transition system  $TS$ , defined as the set of actions labeling transitions of  $TS$  reachable from an initial state, i.e.,

$$\Sigma(TS) := \{a \in \Sigma \mid \exists q_0 \in I, w \in \Sigma^*, q, q' \in Q : q_0 \xrightarrow{w} q \xrightarrow{a} q'\}.$$

Obviously,  $\Sigma(TS) \subseteq \Sigma$ .

- $TS$  is called *deterministic*, if

$$|I| = 1 \text{ and } \forall q, q', q'' \in Q, a \in \Sigma : q \xrightarrow{a} q' \text{ and } q \xrightarrow{a} q'' \text{ implies } q' = q''.$$

- $TS$  is called *acyclic*, if the directed graph generated by the transition relation  $\rightarrow$  is acyclic.

**Convention.** Unless otherwise stated, when we say ‘transition system’, we mean ‘reachable finite labeled transition system’.

We can use transition systems to express in a compact way behaviors of systems.

### Definition 2.9 (Run and language of a transition system)

A *run* of  $TS$  is a word  $w \in \Sigma^*$  that can be executed in  $TS$  starting in an initial state, i.e.,  $\exists q^{in} \in I, q \in Q : q_0 \xrightarrow{w} q$ . The *language* of  $TS$ , denoted by  $L(TS)$ , is the set of all the runs of  $TS$ :

$$L(TS) := \{w \in \Sigma^* \mid \exists q^{in} \in I, q \in Q : q^{in} \xrightarrow{w} q\}.$$

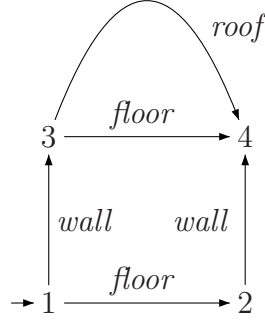


Figure 2.2: A house-like transition system

**Example 2.10** The language of the transition system in Figure 2.2 with 1 as the only initial state (marked by a short incoming arrow) is

$$\{\varepsilon, \text{wall}, \text{floor}, \text{wall.floor}, \text{wall.roof}, \text{floor.wall}\}.$$

**Remark 2.11** For any transition system  $TS$ , we have  $\Sigma(TS) = \Sigma(L(TS))$ , i.e., the alphabet of  $TS$  is equal to the alphabet of its language.

Once we have two transition systems, we can *compare* their state spaces using the graph isomorphism or compare their languages. There are situations where the isomorphism is too strong and language equivalence too weak. An intermediate solution to this issue is given by the notion of *bisimulation*, which proved useful in the study of concurrent systems [Mil89]. Below we give a definition allowing multiple initial states:

**Definition 2.12 (Bisimulation)**

A (strong) *bisimulation* between a pair of transition systems  $TS_1 = (Q_1, \Sigma, \rightarrow_1, I_1)$  and  $TS_2 = (Q_2, \Sigma, \rightarrow_2, I_2)$  is a binary relation  $\sim \subseteq Q_1 \times Q_2$  (for which we use the infix notation) such that:

- For each  $q_1^{in} \in I_1$ , there exists  $q_2^{in} \in I_2$  such that  $q_1^{in} \sim q_2^{in}$ .
- For each  $q_2^{in} \in I_2$ , there exists  $q_1^{in} \in I_1$  such that  $q_1^{in} \sim q_2^{in}$ .
- If  $q_1 \sim q_2$  and  $q_1 \xrightarrow{a}_1 q'_1$ , there exists  $q'_2$  such that  $q_2 \xrightarrow{a}_2 q'_2$  and  $q'_1 \sim q'_2$ .
- If  $q_1 \sim q_2$  and  $q_2 \xrightarrow{a}_2 q'_2$ , there exists  $q'_1$  such that  $q_1 \xrightarrow{a}_1 q'_1$  and  $q'_1 \sim q'_2$ .

Having defined the bisimulation, for a transition system  $TS$  one can define  $\sim_{TS}$  as the *largest* bisimulation between  $TS$  and itself. Since  $\sim_{TS}$  defines an equivalence relation over the states of  $TS$ , we can construct the quotient of  $TS$  over  $\sim_{TS}$ , denoted by  $TS/\sim_{TS}$ .

**Finite Automata**

A finer control over the behavior described by a transition system is obtained by specifying a set of *accepting* states. A run  $w$  will then be accepted only if the state of the transition system after executing  $w$  is accepting. Such an enriched transition system is called a *finite automaton*.

**Definition 2.13 (Finite automaton)**

A *finite automaton* is a tuple  $\mathcal{A} = (Q, \Sigma, \rightarrow, I, F)$ , where  $(Q, \Sigma, \rightarrow, I)$  is a transition system and  $F \subseteq Q$  is a set of *accepting* or *final* states.

The language accepted by the automaton  $\mathcal{A}$  with the set of accepting states  $F$ , denoted by  $L(\mathcal{A}, F)$ <sup>1</sup>, is defined as

$$L(\mathcal{A}, F) := \{w \in \Sigma^* \mid \exists q^{in} \in I, q^{fin} \in F : q^{in} \xrightarrow{w} q^{fin}\}.$$

As intended, finite automata are more expressive than the plain transition systems. The former are able to accept all the regular languages, while the latter only the strictly smaller class of prefix-closed regular languages.

**Theorem 2.14** (Kleene's Theorem) *For a fixed alphabet  $\Sigma$ , the class of languages accepted by finite automata coincides with the class of regular languages.*

**Corollary 2.15** *For a fixed alphabet  $\Sigma$ , the class of languages accepted by transition systems coincides with the class of prefix-closed regular languages.*

**Proof.** For one direction, we can see a transition system  $TS$  as a finite automaton with all states final. Then, by Theorem 2.14,  $L(TS)$  is regular.  $L(TS)$  is prefix-closed because any partial run of a run  $w \in L(TS)$  is also a run of  $TS$ .

For the reverse direction, let  $L \subseteq \Sigma^*$  be a prefix-closed regular language. Since  $L$  is regular, by Theorem 2.14, there exists an automaton  $\mathcal{A} = (Q, \Sigma, \rightarrow, I, F)$  such that  $L(\mathcal{A}, F) = L$ , where  $F \subseteq Q$  is the set of accepting states. Let  $Q' \subseteq Q$  be the subset of states of  $\mathcal{A}$  that are both reachable from an initial state and co-reachable from an accepting state. I.e.,  $Q'$  consists of the states that are on a path from an initial to an accepting state. We choose  $TS := (Q', \Sigma, \rightarrow, I')$  as the transition system induced by  $Q'$  and we show that  $L(TS) = L(\mathcal{A}, F)$  (so  $L(TS) = L$ ).

The inclusion  $L(\mathcal{A}, F) \subseteq L(TS)$  follows easily from the fact that in particular the accepting states of  $Q$  are on a path from an initial to an accepting state in  $\mathcal{A}$  ( $\mathcal{A}$  is reachable). For the reverse inclusion,  $L(TS) \subseteq L(\mathcal{A}, F)$ , we use the hypothesis that  $L = L(\mathcal{A}, F)$  is prefix-closed. Let  $w \in L(TS)$ . Then, there exists a path  $q^{in} \xrightarrow{w} q$  with  $q^{in} \in I'$  and  $q \in Q'$  in  $TS$ . By construction, the state  $q \in Q'$  is on a path from an initial to an accepting state in  $\mathcal{A}$ , so the word  $w$  is the prefix of a word of  $L(\mathcal{A}, F)$  and, since  $L(\mathcal{A}, F)$  is prefix-closed, this implies that  $w \in L(\mathcal{A}, F)$ .  $\square$

Another classic result in automata theory shows that the deterministic restriction does not decrease the expressiveness power of finite automata. The same result will hold for transition systems.

**Theorem 2.16** *For a fixed alphabet  $\Sigma$ , the class of languages accepted by finite automata coincides with the class of languages accepted by deterministic finite automata.*

**Corollary 2.17** *For a fixed alphabet  $\Sigma$ , the class of languages accepted by transition systems coincides with the class of languages accepted by deterministic transition systems.*

<sup>1</sup>We include the set of accepting states into the notation  $L(\mathcal{A}, F)$  for automata to stress the language acceptance condition difference to the transition systems (where all states are seen as accepting). In this thesis, we will mainly work with transition systems.

**Proof.** Let  $L(TS)$  be the language of a (nondeterministic) transition system  $TS$ .  $TS$  can be seen as a finite automaton with all states accepting. Then, we apply Theorem 2.16 (i.e., the usual subset construction) and we obtain a deterministic finite automaton  $\mathcal{A}$  whose language is equal to  $L(TS)$ . Since  $L(TS)$  is prefix-closed, we can apply the construction from the proof of Corollary 2.15 and obtain a deterministic transition system accepting  $L(TS)$ .  $\square$

An important property of deterministic finite automata regards minimization:

**Theorem 2.18** [HU79, Section 3.4] *For each regular language  $L$ , there exists a unique (up to isomorphism) minimal deterministic finite automaton accepting  $L$ .*

**Proof.** (Construction idea – recalled here for further reference) For each regular language  $L$ , there exists a deterministic finite automaton  $\mathcal{A} = (Q, \Sigma, \rightarrow, \{q_0\}, F)$  such that  $L = L(\mathcal{A}, F)$  (Theorems 2.14 and 2.16). Then, the minimal deterministic automaton accepting the same language as  $\mathcal{A}$  is obtained by the classical minimization algorithm:

1. Remove the states unreachable from the (unique) initial state  $q_0$ .
2. Make the transition relation *total*, i.e., for any reachable state  $q \in Q$  and any action  $a \in \Sigma$ , there exists  $q' \in Q$  such that  $q \xrightarrow{a} q'$ . If the original transition relation was not total, introduce a new non-final ‘sink state’  $\perp$  such that:  $\perp \xrightarrow{a} \perp$  for any  $a \in \Sigma$  and  $q \xrightarrow{a} \perp$  for each state  $q$  and action  $a$  such that there is no  $q'$  with  $q \xrightarrow{a} q'$ .

Once we know that  $\rightarrow$  is deterministic and total, for each  $q \in Q$  and  $w \in \Sigma^*$ , there exists a unique state  $q'$  such that  $q \xrightarrow{w} q'$ . We denote  $q'$  by  $\delta(q, w)$ .

3. Compute the following equivalence on the state space depending on the set of final states  $F$ :

$$q \approx_F q' \text{ if and only if } (\forall w \in \Sigma^* : \delta(q, w) \in F \Leftrightarrow \delta(q', w) \in F).$$

4. The minimal deterministic automaton is then obtained by taking the quotient w.r.t.  $\approx_F$ .

The above constructed automaton is unique up to isomorphism w.r.t. the given properties.  $\square$

**Corollary 2.19** *For each prefix-closed regular language  $L$ , there exists a unique (up to isomorphism) minimal deterministic transition system accepting  $L$ .*

**Proof.** (Construction idea) For each prefix-closed regular language  $L$ , there exists a deterministic transition system  $TS = (Q, \Sigma, \rightarrow, \{q_0\})$  accepting  $L$  (Corollaries 2.15 and 2.17). We can see  $TS$  as a finite automaton with all states final, i.e.,  $F := Q$  and apply the minimization construction presented in the proof of Theorem 2.18. Given the special set of final states, computing the equivalence of step 3 in the minimization algorithm becomes:

$$q \approx q' \text{ if and only if } (\forall w \in \Sigma^* : q \xrightarrow{w} \perp \Leftrightarrow q' \xrightarrow{w} \perp).$$

Additionally, at the end of the minimization procedure, we remove the sink state  $\perp$  (which was not final!) in order to obtain the minimal deterministic transition system  $TS'$  such that  $L = L(TS')$ .  $\square$

Finally, it is easy to see that imposing the acyclicity restriction to sequential machines, we obtain finite behaviors. (Note that a finite language is always regular.)

**Theorem 2.20** *For a fixed alphabet  $\Sigma$ , the class of languages accepted by acyclic finite automata coincides with the class of finite languages.*

**Corollary 2.21** *For a fixed alphabet  $\Sigma$ , the class of languages accepted by acyclic transition systems coincides with the class of prefix-closed finite languages.*

## Discussion

In this chapter we provided the basic ingredients needed in this thesis: set and graph theory, formal languages and sequential machines (transition systems and finite automata).

We are now ready to move forward to learn about models of distributed systems which are introduced next.







---

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

*Leslie Lamport*

## CHAPTER 3

# DISTRIBUTED TRANSITION SYSTEMS AND THE SYNTHESIS PROBLEM

---

THE synthesis problem for distributed systems we consider is: Given a global specification and a distribution structure, find if possible a distributed system complying with the specification. In this chapter, we present the models of distributed systems that we are working with together with characterization results that help solving the synthesis problem. The structure of the chapter is sketched below.

We meet the first inhabitants of the theoretical world of concurrency in Section 3.1, where we get to know about independent actions and traces, which are classes of equivalent executions w.r.t. the independence relation. In Section 3.2, we move deeper into the concurrency area learning about distribution of actions over a set of processes and distributed transition systems. We present two related models of local transition systems synchronizing on common actions, namely the class of synchronous products of transition systems [Arn94] and of asynchronous automata [Zie87]. (The index at the end of the thesis will help at a later time to relocate the definitions and notations of this chapter.) Sections 3.3, respectively 3.4, give properties and characterizations of the global state space, respectively the languages, of the chosen models of distributed systems. Finally, Section 3.5 describes the versions of synthesis of distributed systems considered in this thesis.

### 3.1 Trace Theory

The theoretical interest for concurrent systems increased in the 60s with the seminal work of C.A. Petri [Pet62]. The new stream of fundamental research required appropriate tools to study the concurrent behaviors. The formal languages (Section 2.1.3) proved very suitable for sequential machines [HU79], but they showed their limitations when used within the new paradigm<sup>1</sup>. At the end of the 70s, Mazurkiewicz [Maz77] introduced the notion

---

<sup>1</sup>The most popular approach to study concurrent systems using classical formal languages theory was based on *interleaving*. In this case, true concurrency is replaced by the *nondeterministic* choice of the order of execution of a set of concurrent actions. Although useful in many situations, the interleaving unfortunately ‘forgets’ the concurrent structure of a computation which might be essential in problems

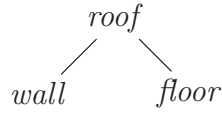


Figure 3.1: The dependence graph of the concurrent alphabet of the house-like transition system

of *trace* as a mathematical object able to maintain the information regarding the independence relation between the actions involved. Since then, the theory of traces continued to develop finding various applications (see surveys in [Maz87] and the monograph *The Book of Traces* [DR95]). In this section, we give the basics of trace theory.

The first ingredient that we need is the notion of *concurrent alphabet*, which is simply an alphabet equipped with an independence relation between its actions:

**Definition 3.1 (Concurrent alphabet)**

A *concurrent alphabet* is a pair  $(\Sigma, \parallel)$ , where:

- $\Sigma$  is a finite alphabet of actions and
- $\parallel \subseteq \Sigma \times \Sigma$  is a symmetric and irreflexive binary relation called the *independence* or *concurrency* relation.

The complementary relation of the independence relation is called the *dependence* relation and it is denoted by  $\bowtie$ . Hence, by definition,  $\bowtie := (\Sigma \times \Sigma) \setminus \parallel$ . Since  $\parallel$  is symmetric and irreflexive, we have that  $\bowtie$  is a symmetric and reflexive relation.

Both the dependence and independence relations can be represented using undirected graphs. More precisely, the dependence graph associated to a concurrent alphabet is the undirected graph with vertex set  $\Sigma$  and with edge set  $\bowtie \setminus \{(a, a) \mid a \in \Sigma\}$ . Dually, the *independence graph* of  $(\Sigma, \parallel)$  is the undirected graph with vertex set  $\Sigma$  and with edge set  $\parallel$ .

**Example 3.2** We can add a concurrency flavor to the house-like transition system of Figure 2.2 by supposing that the actions *wall* and *floor* are independent. So, we have the concurrent alphabet

$$(\{wall, floor, roof\}, \{wall \parallel floor\})$$

The dependence graph of the above concurrent alphabet is depicted in Figure 3.1.

We use the information provided by the independence relation so as to identify runs where adjacent independent actions are permuted. The intuition goes as follows: If two actions  $a$  and  $b$  are supposed to be independent ( $a \parallel b$ ), then if at some point we can execute  $a$  followed by  $b$ , executing  $a$  should not affect in any way the possibility of executing  $b$ , so they may be executed in any order, in particular first  $b$  and then  $a$ . For instance, for  $\Sigma := \{a, b, c\}$  and  $a \parallel b$ , the runs  $caab$ ,  $caba$ , and  $cbaa$  must be all equivalent.

More precisely, we do not want to distinguish two runs that differ only in the order in which two adjacent independent actions are executed. This is theoretically implemented by constructing an equivalence relation in the following way:

---

like *system refinement* or *serializability of transactions* in concurrent databases.

**Definition 3.3 (Trace equivalence)**

Given a concurrent alphabet  $(\Sigma, \parallel)$ , the *trace equivalence*  $\sim$  over  $\Sigma^*$  is defined as follows. For  $w, w' \in \Sigma^*$ , we have  $w \sim w'$  if and only if there exist the words  $v_0, \dots, v_n \in \Sigma^*$  such that  $w = v_0$ ,  $w' = v_n$ , and for each  $i \in [1..n]$ , there exist  $u_i, u'_i \in \Sigma^*$  and  $a_i, b_i \in \Sigma$  such that

$$a_i \parallel b_i, v_{i-1} = u_i a_i b_i u'_i, \text{ and } v_i = u_i b_i a_i u'_i.$$

Equivalently, the trace equivalence is the smallest equivalence  $\sim$  on  $\Sigma^*$  such that

$$\forall u, v \in \Sigma^*, a, b \in \Sigma : a \parallel b \Rightarrow uabv \sim ubav.$$

It is easy to see that the trace equivalence  $\sim$  is in fact a *congruence* on  $\Sigma^*$  with respect to concatenation, i.e., whenever  $w \sim w'$ , then for any words  $u, v \in \Sigma^*$ , we have  $uwv \sim uw'v$ .

**Definition 3.4 (Trace monoid, trace, and trace language)**

Given a concurrent alphabet  $(\Sigma, \parallel)$ , the quotient of  $\Sigma^*$  over the congruence  $\sim$  is called the *trace monoid* and is usually denoted by  $\mathbb{M}(\Sigma, \parallel)$ .<sup>1</sup> If we denote by  $[w]$  the  $\sim$ -equivalence class<sup>2</sup> of the word  $w \in \Sigma^*$ , then

$$\mathbb{M}(\Sigma, \parallel) = (\Sigma^* / \sim, \cdot, [\varepsilon])$$

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

A *trace* is by definition an element of  $\mathbb{M}(\Sigma, \parallel)$ , that is, a  $\sim$ -equivalence class.

A *trace language* is a subset of  $\mathbb{M}(\Sigma, \parallel)$ .

In case the independence (dependence) relation  $\parallel$  has a special form, the trace monoid  $\mathbb{M}(\Sigma, \parallel)$  allows special characterizations. For instance:

- If  $\parallel = \emptyset$ , then the trace monoid  $\mathbb{M}(\Sigma, \parallel)$  is isomorphic to the free monoid  $\Sigma^*$ .
- If  $\parallel$  is full, i.e.,  $\parallel = \Sigma \times \Sigma \setminus \{(a, a) \mid a \in \Sigma\}$ , then  $w \sim w'$  if and only if  $\#_a w = \#_a w'$ , for all  $a \in \Sigma$ . This implies that  $\mathbb{M}(\Sigma, \parallel)$  is isomorphic to  $\mathbb{N}^\Sigma$ .
- If the dependence relation  $\not\parallel$  is transitive, then the maximal cliques  $\Sigma_1, \dots, \Sigma_n$  of the dependence graph are disjoint and  $\mathbb{M}(\Sigma, \parallel)$  is isomorphic to the cartesian product  $\Sigma_1^* \times \dots \times \Sigma_n^*$  of the free monoids generated by these cliques.

To every trace language  $T \subseteq \mathbb{M}(\Sigma, \parallel)$ , we can associate a (word) language of  $\Sigma^*$  in the following way:

$$\text{lin}(T) := \{w \in \Sigma^* \mid \exists t \in T \text{ such that } w \in t\}.$$

For a trace language  $T$ ,  $\text{lin}(T)$  is a *linearization*<sup>3</sup> function which generates the set of all members of each trace (read: equivalence class) of  $T$ .

<sup>1</sup>The trace monoid  $\mathbb{M}(\Sigma, \parallel)$  is also known in the literature as the *free partially commutative monoid*.

<sup>2</sup>We could have written the  $\sim$ -equivalence class as  $[w]_\sim$ , but we did not want to unnecessarily burden the notation with the subscript  $\sim$ , since in most contexts the  $\sim$ -equivalence will be implicit.

<sup>3</sup>The name *linearization* comes from the fact that a trace is compactly described by a poset and the term linearization refers to the set of all linear (i.e., total) orders compatible with the poset.

### Trace-closed Languages

Since we are not mainly interested in studying the algebraic properties of trace languages, we will treat the trace languages as (word) languages of  $\Sigma^*$  satisfying a closure condition rather than as subsets of the trace monoid  $\mathbb{M}(\Sigma, \parallel)$ . The closure property will of course depend on the equivalence relation  $\sim$ :

#### Definition 3.5 (Trace-closed language)

Given a concurrent alphabet  $(\Sigma, \parallel)$ , we say that  $L \subseteq \Sigma^*$  is a *trace-closed language* if it satisfies the following closure property:

$$\forall w \in \Sigma^* : w \in L \Rightarrow [w] \subseteq L.$$

In other words, whenever  $w \in L$  and  $w \sim w'$ , then also  $w' \in L$ .

The following easy proposition shows that the classes of trace languages and trace-closed languages are isomorphic:

**Proposition 3.6** *Given a concurrent alphabet  $(\Sigma, \parallel)$ , there is a 1–1 correspondence between the trace languages of  $\mathbb{M}(\Sigma, \parallel)$  and the trace-closed languages over  $\Sigma^*$ .*

**Proof.** The 1–1 correspondence is given by the linearization function  $lin$ . For that, we show that:

- $lin$  is well-defined: For any  $T \subseteq \mathbb{M}(\Sigma, \parallel)$ ,  $lin(T)$  is a trace-closed language because if  $w \in lin(T)$ , then the trace  $t \in T$  such that  $w \in t$  is exactly  $[w]$ , which will be necessarily included in  $lin(T)$ .
- $lin$  is injective: For  $T_1, T_2 \subseteq \mathbb{M}(\Sigma, \parallel)$  with  $T_1 \neq T_2$ , we show that  $lin(T_1) \neq lin(T_2)$ . Without loss of generality, suppose there exists  $t \in T_1 \setminus T_2$ . For  $w \in t$ , we have that  $w \in lin(T_1)$ . On the other hand,  $w \notin lin(T_2)$ , otherwise  $t \in T_2$ . So,  $w \in lin(T_1) \setminus lin(T_2)$ , which means  $lin(T_1) \neq lin(T_2)$ .
- $lin$  is surjective: Let  $L \subseteq \Sigma^*$  be a trace-closed language. We show that there exists a trace language  $T \subseteq \mathbb{M}(\Sigma, \parallel)$  such that  $L = lin(T)$ . For  $T := \{[w] \mid w \in L\}$ , we have indeed that  $L = lin(T)$ : On one hand,  $L \subseteq lin(T)$  because  $\forall w \in L : w \in [w] \subseteq lin(T)$ . On the other hand,  $lin(T) \subseteq L$  because for any  $w \in lin(T)$ , there exists  $t = [w'] \in T$  with  $w' \in L$  such that  $w \in [w']$ . This implies  $w \sim w'$  and using the fact that  $L$  is trace-closed, we have that  $w \in L$ . □

We mainly use languages to describe behaviors of systems. As we will later see, the behaviors of distributed systems are trace-closed, so it is good to know under which operations is the class of trace-closed languages closed.

**Proposition 3.7** *The class of trace-closed languages over a concurrent alphabet  $(\Sigma, \parallel)$  is closed under the operations of union, intersection, complementation, projection, shuffle product, and prefix-closure, but not closed under concatenation and iteration.*

$$\begin{array}{l}
t = \boxed{t_1} \cdots \boxed{t'.a} \cdots \boxed{t_n} \in L_1 \\
u = \boxed{u_1} \cdots \boxed{b.u'} \cdots \boxed{u_n} \in L_2
\end{array}
\left. \vphantom{\begin{array}{l} t \\ u \end{array}} \right\} \Rightarrow \boxed{t_1} \boxed{u_1} \cdots \boxed{t'.a} \boxed{b.u'} \cdots \boxed{t_n} \boxed{u_n} \in \text{Shuffle}(L_1, L_2)$$

Splitting  $t'.a$  and  $b.u'$ , we manage to permute  $a$  and  $b$  using shuffle

$$\begin{array}{l}
t = \boxed{t_1} \cdots \boxed{t'} \boxed{a} \cdots \boxed{t_n} \in L_1 \\
u = \boxed{u_1} \cdots \boxed{b} \boxed{u'} \cdots \boxed{u_n} \in L_2
\end{array}
\left. \vphantom{\begin{array}{l} t \\ u \end{array}} \right\} \Rightarrow \boxed{t_1} \boxed{u_1} \cdots \boxed{t'} \boxed{b} \boxed{a} \boxed{u'} \cdots \boxed{t_n} \boxed{u_n} \in \text{Shuffle}(L_1, L_2)$$

Figure 3.2: A detail in the proof of Proposition 3.7

**Proof.** The closure under union, intersection, complementation, projection, and prefix-closure can be easily proved.

For the closure under shuffle product, the proof is a bit more elaborated. For every trace-closed languages  $L_1, L_2 \subseteq \Sigma^*$ , we must show that  $\text{Shuffle}(L_1, L_2)$  is also trace-closed.

Let  $w \in \text{Shuffle}(L_1, L_2)$  and  $w' \sim w$ . By the definition of the trace equivalence,  $w'$  is obtained from  $w$  by a number of permutations of adjacent independent actions. We prove that  $w' \in \text{Shuffle}(L_1, L_2)$  by induction on the number of permutations needed to obtain  $w'$  from  $w$ .

*Base case :* We assume that  $w'$  is obtained from  $w$  by *one* permutation of the independent actions  $a||b$  presumably appearing in a sequence  $ab$  in  $w$ . We show that  $w' \in \text{Shuffle}(L_1, L_2)$ .

By the definition of shuffle, there exist  $n \geq 1$  and  $t_i, u_i \in \Sigma^*$  for  $i \in [1..n]$  such that  $t = t_1 \dots t_n \in L_1$ ,  $u = u_1 \dots u_n \in L_2$ , and  $w = t_1 u_1 \dots t_n u_n \in \text{Shuffle}(t, u)$ . We have to consider two cases depending on the position of the sequence  $ab$  in  $w = t_1 u_1 \dots t_n u_n$ :

Case 1 :  $ab$  appears inside a string  $t_k$ , for a certain  $k \in [1..n]$ .

Then,  $w' = t_1 u_1 \dots t'_k u_k \dots t_n u_n$ , where  $t'_k$  is obtained from  $t_k$  by replacing the chosen occurrence of  $ab$  by  $ba$ . Since  $t_1 \dots t_k \dots t_n \in L_1$  and  $L_1$  is trace-closed, we have that  $t_1 \dots t'_k \dots t_n \in L_1$ . So,

$$w' \in \text{Shuffle}(t_1 \dots t'_k \dots t_n, u_1 \dots u_k \dots u_n) \subseteq \text{Shuffle}(L_1, L_2).$$

Case 1' :  $ab$  appears inside a string  $u_k$ , for a certain  $k \in [1..n]$ . Similar to Case 1.

Case 2 :  $a$  appears as the last action of  $t_k$  and  $b$  as the first action of  $u_k$ , for a certain  $k \in [1..n]$ .

Formally, there exist  $t', u' \in \Sigma^*$  such that  $t_k = t'a$  and  $u_k = bu'$ . In this case,  $w = t_1 u_1 \dots t' a b u' \dots t_n u_n$  and  $w' = t_1 u_1 \dots t' b a u' \dots t_n u_n$  ( $a$  and  $b$  where permuted). To prove that  $w' \in \text{Shuffle}(L_1, L_2)$  we apply the trick depicted in Figure 3.2. More precisely, we refine the decomposition  $t_1 \dots t'a \dots t_n$  of  $t$ , by considering  $t'$  and  $a$  as separate substrings, i.e., we choose  $(t'_i)_{i \in [1..n+1]}$  such that  $t'_i := t_i$  for  $i \in [1..k-1]$ ,  $t'_k = t'$ ,  $t'_{k+1} = a$ , and  $t'_i := t_{i-1}$  for  $i \in [k+2..n]$ . Similarly, we split  $bu'$  by choosing  $(u'_i)_{i \in [1..n+1]}$  such that  $u'_i := u_i$  for  $i \in [1..k-1]$ ,  $u'_k = b$ ,  $u'_{k+1} = u'$ , and  $u'_i := u_{i-1}$  for  $i \in [k+2..n]$ .

Using the above decompositions of  $t$  and  $u$ , it is easy to see that

$$w' \in \text{Shuffle}(t, u) \subseteq \text{Shuffle}(L_1, L_2)$$

Case 2' :  $a$  appears as the last action of  $u_k$  and  $b$  as the first action of  $t_{k+1}$ , for a certain  $k \in [1 .. n - 1]$ . Similar to Case 2.

*Induction step* : If we suppose  $w' \in \text{Shuffle}(L_1, L_2)$  for all  $w'$  obtained from  $w$  by  $N$  permutations of adjacent independent actions, then it is immediate using the base case to show that the same property holds for  $N + 1$ .

For the last part of the proposition, we show that the class of trace languages is not closed under concatenation and iteration: Take for instance,  $\Sigma = \{a, b\}$  with  $a||b$  and the trace-closed languages  $L_1 := \{a\}$ ,  $L_2 := \{b\}$ , and  $L := \{ab, ba\}$ . Then,  $L_1L_2 = \{ab\}$  is not trace-closed, because  $ba \notin L_1L_2$ . Also,  $L^*$  is not trace-closed, because  $abab \in L^*$ , but  $aabb \notin L^*$ .  $\square$

For a run  $w$ , the trace  $[w]$  consists of all words  $\sim$ -equivalent to  $w$ , which means that  $[w]$  is the closure of  $w$  with respect to the trace equivalence  $\sim$ . In a similar fashion, we can lift the trace-closure operation from words to languages in the natural way:

For  $L \subseteq \Sigma^*$ , we denote by  $[L]$  the *trace-closure* of the language  $L$ , which is defined as:

$$[L] := \bigcup_{w \in L} [w].$$

Equivalently,  $[L]$  is the smallest (w.r.t. set inclusion) trace-closed language including  $L$ .

Regarding the classes of languages introduced as far, the trace-closure operator behaves as follows:

**Proposition 3.8** *For a concurrent alphabet  $(\Sigma, ||)$ , neither the class of prefix-closed languages nor the class of regular languages is closed under the trace-closure operation.*

**Proof.** Counterexamples can be quickly given. Let the concurrent alphabet be

$$(\{a, b\}, a||b).$$

On one hand, take the prefix-closed language  $L := \{\epsilon, a, ab\}$ . Then,  $[L]$  is not trace-closed because  $ba \in [L]$ , but  $b \notin [L]$ . On the other hand, take the regular language  $L := (ab)^*$ . Then,  $[L] = \{w \in \Sigma^* \mid \#_a w = \#_b w\}$ , which is a classical example of a non-regular language.  $\square$

### Regular Trace Languages

Maybe the most studied class of trace languages is that of *regular trace languages*. To introduce them, we follow the presentation and terminology of [Zie87]. There are other equivalent definitions and characterizations for the class of regular trace languages (which are sometimes known as *recognizable trace languages*), but since they do not play a rôle in this thesis we do not present them here (a curious reader is referred to [DR95]).

Similar to the word languages, for a trace language  $T \subseteq \mathbb{M}(\Sigma, \parallel)$ , we define the *syntactic congruence*  $\sim_T$  of  $T$  as follows: For  $t, t' \in \mathbb{M}(\Sigma, \parallel)$ ,

$$t \sim_T t' \text{ if and only if } \forall x, y \in \mathbb{M}(\Sigma, \parallel) : xty \in T \Leftrightarrow xt'y \in T.$$

Following [Zie87], we have the following definition (cf. Theorem 2.4):

**Definition 3.9 (Regular trace language)**

A trace language  $T \subseteq \mathbb{M}(\Sigma, \parallel)$  is *regular*, if its associated syntactic congruence  $\sim_T$  is of finite index.

We denote the class of regular trace languages by  $\mathbb{R}\text{eg}(\Sigma, \parallel)$ .

The following two results relate the trace languages with their word counterparts:

**Lemma 3.10** [Zie87] *Let  $T \subseteq \mathbb{M}(\Sigma, \parallel)$  and  $v, w \in \Sigma^*$ . Then,  $[v] \sim_T [w]$  if and only if  $v \sim_{\text{lin}(T)} w$ .*

**Proposition 3.11** [Zie87] *For  $T \subseteq \mathbb{M}(\Sigma, \parallel)$ , we have  $T \in \mathbb{R}\text{eg}(\Sigma, \parallel)$  if and only if  $\text{lin}(T) \in \text{Reg}(\Sigma)$ .*

From the above result and Proposition 3.6 we easily obtain:

**Corollary 3.12** *Given a concurrent alphabet  $(\Sigma, \parallel)$ , there is a 1–1 correspondence between the regular trace languages of  $\mathbb{M}(\Sigma, \parallel)$  and the regular trace-closed languages over  $\Sigma^*$ .*

For convenience, we will work more with trace-closed languages rather than with trace languages (we find it easier to work with classic languages of words than with languages of equivalence classes of words).

## 3.2 Distributed Transition Systems

In this section we generalize the notion of transition system to a distributed setting.

A *sequential system* consists of one agent that executes a number of tasks in a certain order. Such a system can be theoretically modeled, for instance, by a transition system (see Definition 2.8). A *distributed system* consists of a number of agents that cooperate to execute a number of tasks. In real life, given the complexity of possible interactions, it is harder to find examples of sequential systems than of distributed ones. For instance, reading this piece of text by a human implies a mechanical synchronization of the two eyes (running over the lines) done in parallel with the synchronization of the hands (handling the cup of coffee/browsing the pages).

A distributed system which has no interaction with the environment is called *closed* (e.g., a distributed algorithm processing a given input), otherwise it is called *open* (e.g., an operating system interacting with several users). In this thesis, we will only consider the case of closed distributed systems, so, unless otherwise specified, when we talk about a ‘distributed system’, we mean ‘closed distributed system’.

Modeling a distributed system by an ordinary transition system is usually inefficient and ontologically inaccurate since the very paradigm of distributed activity is flattened



and thus structural information is lost. There is a number of already established theoretical models for distributed computation (Petri nets occupying a prominent position). In this thesis we will study two of them, both based on *synchronization on common actions*: the synchronous products of transition systems [Arn94] and the asynchronous automata [Zie87]. Loosely speaking, they consist of a number of local processes (or agents) that are able to execute a set of actions and which must synchronize on their common actions in order to change their local states. The latter model will take in consideration also the local states of the agents synchronizing to execute an action (so the latter model is more general than the former). Moreover, the synchronization on common actions requires a *distribution* pattern telling which agents are able to execute what actions.

To exemplify the concepts, think of a building site with a team of diligent workers. We suppose that each worker has several specializations (e.g., masonry, carpentry, plumbing, plastering, painting). Due to their complexity, the tasks require the full manpower with a given specialization. Then, we can see the multitasking activity of building as a distributed transition system, where: the specializations are the possible actions, each task is the synchronization of workers, and each worker is a local process that takes part only in tasks involving his specializations.

The next sections will introduce and discuss the notion of distribution of actions over a set of processes followed by the two notions of distributed transition systems which will be studied in this thesis.

### 3.2.1 Distributions

In this section we define the notion of *distributed alphabet* or (shorter) *distribution*. This will provide the ‘infrastructure’ of the distributed transition systems.

Informally, a distributed alphabet consists of an alphabet of actions together with a set of processes and the information whether an action may be executed by a process or not. Formally, we have:

**Definition 3.13 (Distribution)**

A *distribution* is a tuple  $(\Sigma, Proc, \Delta)$ , where

- $\Sigma$  is a nonempty, finite alphabet of *actions*,
- $Proc$  is a nonempty, finite set of *process labels*, and
- $\Delta \subseteq \Sigma \times Proc$  is a binary relation between actions and processes such that each action is in relation with at least one process and vice versa, each process is in relation with at least one action<sup>1</sup>.

Once we have a distribution  $(\Sigma, Proc, \Delta)$ , we can extract action-oriented, respectively process-oriented information in the following way:

- For each action we can derive the (nonempty) set of processes that are able to execute that action. Namely, we have the function  $dom : \Sigma \rightarrow \mathcal{P}(Proc) \setminus \emptyset$  defined for each  $a \in \Sigma$  as:

$$dom(a) := \{p \in Proc \mid (a, p) \in \Delta\}.$$

---

<sup>1</sup>The two last conditions keep out abnormal specifications with ‘phantom’ actions or processes.



The function  $dom$  is called the *domain function*, as it provides the ‘domain’ where the action is active.

One can give directly a domain function  $dom$  and from it to uniquely determine the distribution  $\Delta := \{(a, p) \mid p \in dom(a)\}$  that generates  $dom$ .<sup>1</sup>

- Dually, for each process we can derive the (nonempty) set of actions that may be executed by that process. Namely, we have the function  $\Sigma_{loc} : Proc \rightarrow \mathcal{P}(\Sigma) \setminus \emptyset$  defined for each  $p \in Proc$  as:

$$\Sigma_{loc}(p) := \{a \in \Sigma \mid (a, p) \in \Delta\}.$$

The function  $\Sigma_{loc}$  provides the *local alphabet* of each process, i.e., the set of actions that the given process is able to execute.

One can give directly the local alphabets by  $\Sigma_{loc}$  and from them to uniquely determine the distribution  $\Delta := \{(a, p) \mid a \in \Sigma_{loc}(p)\}$  that can generate them.<sup>2</sup>

**Convention.** In unequivocal contexts we will simply use  $\Delta$  to denote  $(\Sigma, Proc, \Delta)$ . Also, when it is more convenient, instead of the binary relation  $\Delta$ , we provide the domain function or the local alphabets. According to the observations above, the distribution  $\Delta$  can be immediately recovered.

**Example 3.14** For the alphabet of the house-like transition system of Figure 2.2,  $\Sigma = \{wall, floor, roof\}$ , we can give a distribution over two processes/agents  $Proc = \{1, 2\}$  in the following way:

$$\Delta := \{(wall, 1), (roof, 1), (floor, 2), (roof, 2)\}.$$

If we interpret the set of processes as a team of two workers having various specializations, then the above distribution says that worker 1 can raise walls and roofs, while worker 2 is skilled to build floors and roofs.

We can further generate the domains of the actions and the local alphabets of the processes:

	<i>dom</i>		$\Sigma_{loc}$
<i>wall</i>	1	1	<i>wall, roof</i>
<i>floor</i>	2	2	<i>floor, roof</i>
<i>roof</i>	1, 2		

### The Relation between Distributed and Concurrent Alphabets

We have now two generalizations of the notion of alphabet of actions: the concurrent alphabet (Definition 3.1) and the distributed alphabet (Definition 3.13). The names themselves already hint at the difference between the two notions. The term ‘concurrent’ suggests that a concurrent alphabet  $(\Sigma, \parallel)$  tells which actions can be executed in parallel, or concurrently. The term ‘distributed’ suggests that a distributed alphabet  $(\Sigma, Proc, \Delta)$

<sup>1</sup>To comply with Definition 3.13, one has to check whether each process occurs in at least one domain.

<sup>2</sup>To comply with Definition 3.13, one has to check whether each action occurs in at least one local alphabet.

provides more information, namely, how the actions are dispersed in a network of local processes.

Every distribution  $(\Sigma, Proc, \Delta)$  gives rise to a natural independence relation  $\parallel$  between the actions of  $\Sigma$ . For each pair  $a, b \in \Sigma$  we set:

$$a \parallel b \Leftrightarrow \text{dom}(a) \cap \text{dom}(b) = \emptyset.$$

The intuition is that two actions are independent if and only if they can be executed by disjoint sets of processes. With this definition, two independent actions cannot interfere in any way and thus they may be executed in parallel. Clearly, the relation defined above is irreflexive and symmetric.

The other way around, for every concurrent alphabet  $(\Sigma, \parallel)$ , we can construct a distribution  $(\Sigma, Proc, \Delta)$  such that the independence relation induced by  $\Delta$  is the same as  $\parallel$  in the following way:

1. start with the dependence graph of  $(\Sigma, \parallel)$
2. choose a clique cover of it
3. create a process  $p_C$  for each clique  $C$  of the cover (so the size of  $Proc$  is equal to the size of the cover)
4. set  $(a, p_C) \in \Delta$  if and only if  $a$  belongs to the clique  $C$ .

It is easy to show that indeed, the independence relation generated by the above distribution is equal to  $\parallel$ .

According to the observations in Section 2.1.4, there are several possibilities to choose a covering by cliques of a graph, and thus several distributions inducing the same independence relation. Special cases are the distributions generated by the covers of all maximal cliques and of cliques of at most two nodes (see page 17).

Let us consider the concurrent alphabet of Example 3.2. The associated dependence graph (depicted in Figure 3.1) accepts the clique cover consisting of the two cliques  $\{wall, roof\}$  and  $\{floor, roof\}$ . According to the above algorithm, we create a process for each clique, say process 1, respectively process 2. Then the distribution obtained is exactly the one given in Example 3.14.

### 3.2.2 Synchronous Products of Transition Systems

We introduce now the first of our two models of distributed transition systems that we use in this thesis: the *synchronous product of transition systems*.

Let us give the general idea on the structure and dynamics of a synchronous product of transition systems. As the name suggests, we have a number of local transition systems that synchronize to execute shared actions. The relation between the actions and the local transition systems is given by a distribution: We identify a *process* with each local transition system and the binary relation between actions and processes tells which actions can be executed by what local transition system. From this relation we extract the domains of the actions and the local alphabets of the local transition systems. Then, an

action  $a$  can be executed at a certain point by the synchronous product of the transition systems only if all the local transition systems of  $\text{dom}(a)$  are able to execute  $a$  at that point. The execution of  $a$  triggers the changing of the local states of the transition systems of  $\text{dom}(a)$  and leaves the rest of the local transition systems unaffected.

The global state space of a synchronous product of transition systems is included in the cartesian product of the state spaces of the local transition systems. More precisely, the global state space consists of the global states that are reachable from a set of initial global states. Formally, we have the following definition:

**Definition 3.15 (Synchronous product of transition systems)**

A *synchronous product of transition systems*  $\mathcal{SP}$  over a distribution  $(\Sigma, \text{Proc}, \Delta)$  is a transition system  $(Q, \Sigma, \rightarrow, I)$  for which there exist

- a family of local state sets  $(Q_p)_{p \in \text{Proc}}$  and
- a family of local transition relations  $(\rightarrow_p)_{p \in \text{Proc}}$  with  $\rightarrow_p \subseteq Q_p \times \Sigma_{\text{loc}}(p) \times Q_p$  for each  $p \in \text{Proc}$ ,

such that:

- $I \subseteq \prod_{p \in \text{Proc}} Q_p$  is the set of global initial states, and
- $Q \subseteq \prod_{p \in \text{Proc}} Q_p$  consists of all the global states reachable from  $I$  by the following transition relation:

$$(q_p)_{p \in \text{Proc}} \xrightarrow{a} (q'_p)_{p \in \text{Proc}} \Leftrightarrow \begin{cases} q_p \xrightarrow{a}_p q'_p & \text{for all } p \in \text{dom}(a) \text{ and} \\ q_p = q'_p & \text{for all } p \notin \text{dom}(a). \end{cases}$$

The synchronous product of transition systems  $\mathcal{SP}$  is *deterministic* if the transition system  $(Q, \Sigma, \rightarrow, I)$  is deterministic.

The synchronous product of transition systems  $\mathcal{SP}$  is *acyclic* if the transition system  $(Q, \Sigma, \rightarrow, I)$  is acyclic.

Sometimes, we simply use ‘synchronous product’ as a shorter version of the longer name ‘synchronous product of transition systems’. The synchronous product of transition systems is a popular model of distributed systems and can be found in the literature under different names like: mixed product [Dub86], product transition systems [TH98], loosely cooperating systems [Zie87, Muk02].

We see that the distribution  $(\Sigma, \text{Proc}, \Delta)$  is deeply embedded in the above definition. We have one local state set together with a local transition relation for *each* process label. The local transitions are labeled with actions taken from the local alphabet of the process. Then, the synchronization on a common action affects only the processes of the domain of that action.

Note that the transition systems are particular cases of synchronous products over distributions with  $|\text{Proc}| = 1$ .

Definition 3.15 identifies a synchronous product of transition systems with the global transition system obtained from synchronizations on common actions (starting with a set of initial states). Nevertheless, this transition system can be generated using the local

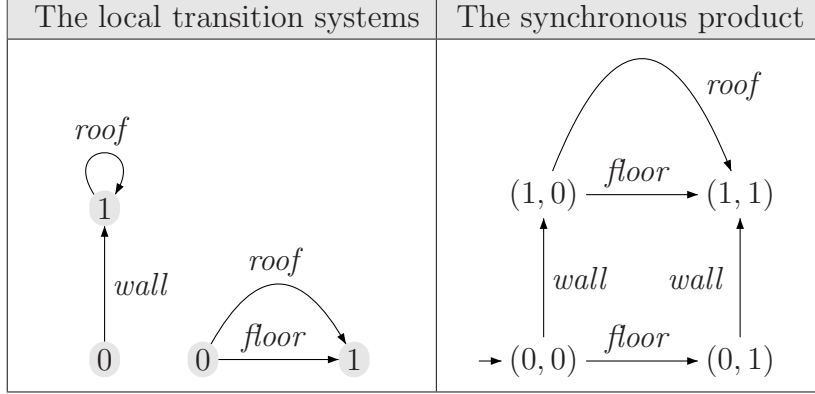


Figure 3.3: The synchronization of two workers

state spaces  $(Q_p)_{p \in Proc}$  and  $(\rightarrow_p)_{p \in Proc}$ , so an efficient data structure for synchronous products keeps only this local information and generates the needed part of the global transition system on demand. In this case, the size of a synchronous product is merely the sum of the sizes of the local transition relations. We use the global transition system in the first place of our definition just for the convenience of presentation.

**Example 3.16** Let us return to the paradigm of specialized workers given at the beginning of Section 3.2. We consider the distribution of Example 3.14 where we have two processes, alias two workers, that have as specializations the local alphabets  $\{wall, roof\}$  and  $\{floor, roof\}$ , respectively. To each worker we associate a behavior by two local states and two local transitions for each worker as shown on the left of Figure 3.3. More precisely, for the first worker we have

$$Q_1 := \{0, 1\} \text{ and } \{0 \xrightarrow{wall} 1, 1 \xrightarrow{roof} 1\},$$

while for the second we have

$$Q_2 := \{0, 1\} \text{ and } \{0 \xrightarrow{floor} 1, 0 \xrightarrow{roof} 1\}.$$

The synchronous product of the two transition systems starting in the global initial state  $(0, 0)$  (so,  $I = \{(0, 0)\}$ ) is given in the right of Figure 3.3.

In the global state  $(0, 0)$  there are two actions executable: *wall* and *floor*. The action *wall* can be executed in  $(0, 0)$  because  $dom(wall) = \{1\}$  and *wall* can be executed in local state 0 of the first component. In this case, only the first component will update to the local state 1. Thus,  $(0, 0) \xrightarrow{wall} (1, 0)$ . A similar reasoning can be performed for the global transition  $(0, 0) \xrightarrow{floor} (0, 1)$ .

The action *roof* has domain  $\{1, 2\}$ , so both workers must synchronize in order to execute it. That is, both of them must be in a local state where *roof* is enabled and they simultaneously change their local states by executing *roof*. In our case, *roof* can be executed only in the global state  $(1, 0)$  by the transition  $(1, 0) \xrightarrow{roof} (1, 1)$ .

We see that the resulted global transition system is isomorphic to the transition system of Example 2.10 (see Figure 2.2).

According to the Definition 3.15,  $Q \subseteq \prod_{p \in Proc} Q_p$ , i.e., the global state space of a synchronous product is embedded in the cartesian product of its local state spaces (note that we consider in  $Q$  only the global states that are reachable from the initial states  $I$ ). In the worst case, we have that  $Q = \prod_{p \in Proc} Q_p$ , which implies that  $|Q| = \times_{p \in Proc} |Q_p|$ . This means that the size of the global state space may be exponentially (in the number of processes) larger than the size of local state spaces. This phenomenon is called the ‘state space explosion problem’ and is responsible for most difficulties in reasoning about distributed systems in practice. For instance, in the Example 3.16 above, we see that the global state space has size 4 which is the product of the sizes of the local state spaces.

As a final notion of this section, by definition, the behavior of a synchronous product is given by the language that its global transition system accepts:

**Definition 3.17 (The language of a synchronous product of transition systems)**

The *language accepted* by the synchronous product of transition systems  $\mathcal{SP} = (Q, \Sigma, \rightarrow, I)$  is simply the language of its transition system:

$$L(\mathcal{SP}) := \{w \in \Sigma^* \mid \exists q^{in} \in I, q \in Q : q^{in} \xrightarrow{w} q\}.$$

For the synchronous product of Example 3.16 (see Figure 3.3), the accepted language is:

$$\{\varepsilon, wall, floor, wall.floor, wall.roof, floor.wall\}.$$

A finer control over the accepted language is obtained adding a set of global final states  $F \subseteq Q$  and modifying the definition accordingly:

$$L(\mathcal{SP}, F) := \{w \in \Sigma^* \mid \exists q^{in} \in I, q^{fin} \in F : q^{in} \xrightarrow{w} q^{fin}\}.$$

Further details and discussions about the two versions above are postponed to Section 3.4.

### 3.2.3 Asynchronous Automata

We introduce now the second of our two models of distributed transition systems that we use in this thesis: the *asynchronous automata* [Zie87].

Compared to the synchronous products, the asynchronous automata are more complex and also more powerful: Any synchronous product can be presented as an asynchronous automaton and there are asynchronous automata that cannot be modeled by synchronous products. Based also on synchronization on common actions, the particularity of asynchronous automata is that the execution of an action depends on the local states of the processes taking part in the synchronization (i.e., the processes of the domain of the action). More precisely, we associate a transition relation with *each action*  $a$  which tells for which tuples of local states in  $dom(a)$ , the action  $a$  can be executed and how the local states are modified. The local processes outside the domain are untouched. Formally, we have:

**Definition 3.18 (Asynchronous automaton)**

An *asynchronous automaton*  $\mathcal{AA}$  over a distribution  $(\Sigma, Proc, \Delta)$  is a transition system  $(Q, \Sigma, \rightarrow, I)$  for which there exist

- a family of local state sets  $(Q_p)_{p \in Proc}$  and
- a family of transition relations  $(\rightarrow_a)_{a \in \Sigma}$  with  $\rightarrow_a \subseteq \prod_{p \in dom(a)} Q_p \times \prod_{p \in dom(a)} Q_p$  for each  $a \in \Sigma$ ,

such that:

- $I \subseteq \prod_{p \in Proc} Q_p$  is the set of global initial states, and
- $Q \subseteq \prod_{p \in Proc} Q_p$  consists of all the global states reachable from  $I$  by the following transition relation:

$$(q_p)_{p \in Proc} \xrightarrow{a} (q'_p)_{p \in Proc} \Leftrightarrow \begin{cases} (q_p)_{p \in dom(a)} \rightarrow_a (q'_p)_{p \in dom(a)} \text{ and} \\ q_p = q'_p \text{ for all } p \notin dom(a). \end{cases}$$

The asynchronous automaton  $\mathcal{AA}$  is *deterministic* if the transition system  $(Q, \Sigma, \rightarrow, I)$  is deterministic.

The asynchronous automaton  $\mathcal{AA}$  is *acyclic* if the transition system  $(Q, \Sigma, \rightarrow, I)$  is acyclic.

The synchronization mechanism of asynchronous automata is more powerful than in the case of synchronous products. When synchronizing on a common action  $a$ , the processes of  $dom(a)$  update their local state *taking into account* the local states of their peers in  $dom(a)$ . So, more information is exchanged via the relations  $\rightarrow_a$  (in the case of synchronous products, the processes agree on synchronization, but update their local states independently).

The global transition system on the right of Figure 3.3 can be realized by an asynchronous automaton in the following way: The local state spaces associated with the two processes  $\{1, 2\}$  are the same as before

$$Q_1 = \{0, 1\} \text{ and } Q_2 = \{0, 1\},$$

and the local transition relations associated with the actions are:

$$\begin{array}{ll} 0 \xrightarrow{wall} 1 & \text{in state space } Q_1 \text{ (because } dom(wall) = \{1\}\text{),} \\ 0 \xrightarrow{floor} 1 & \text{in state space } Q_2 \text{ (because } dom(floor) = \{2\}\text{), and} \\ (1, 0) \xrightarrow{roof} (1, 1) & \text{in state space } Q_1 \times Q_2 \text{ (because } dom(roof) = \{1, 2\}\text{).} \end{array}$$

In the global initial state  $(0, 0)$ , we can execute *wall* and *floor* in parallel (since they affect disjoint sets of processes, i.e., the first, respectively second component). For instance,  $(0, 0) \xrightarrow{wall} (1, 0)$  because  $0 \xrightarrow{wall} 1$  and the first component of  $(0, 0)$  is 0. Then,  $(1, 0) \xrightarrow{roof} (1, 1)$  because  $(1, 0) \xrightarrow{roof} (1, 1)$ . In the global state  $(1, 0)$  also *floor* can be executed because  $dom(floor) = \{2\}$ , the second component of  $(1, 0)$  is 0, and  $0 \xrightarrow{floor} 1$ , so we have  $(1, 0) \xrightarrow{floor} (1, 1)$ . On the other hand, *wall* cannot be executed in  $(1, 0)$  because the first component is 1 and there is no transition starting with 1 in  $\rightarrow_{wall}$ .

In fact, it is not difficult to see that the synchronous products are weaker than the asynchronous automata:

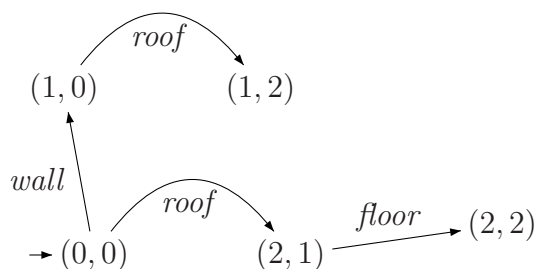


Figure 3.4: A ‘futuristic ambient’ asynchronous automaton that is not a synchronous product

**Remark 3.19** Every synchronous product over a distribution  $\Delta$  accepts a description as an asynchronous automaton over the same distribution  $\Delta$ , but not the other way around.

**Proof.** Let  $\mathcal{SP} = (Q, \Sigma, \rightarrow, I)$  be a synchronous product over distribution  $(\Sigma, Proc, \Delta)$ . Then, there exist a family of local state sets  $(Q_p)_{p \in Proc}$  and a family of local transition relations  $(\rightarrow_p)_{p \in Proc}$  with  $\rightarrow_p \subseteq Q_p \times \Sigma_{loc}(p) \times Q_p$  for each  $p \in Proc$ . We construct a family of transition relations  $(\rightarrow_a)_{a \in \Sigma}$  with  $\rightarrow_a \subseteq \prod_{p \in dom(a)} Q_p \times \prod_{p \in dom(a)} Q_p$  for each  $a \in \Sigma$  in the following way:

$$(q_p)_{p \in dom(a)} \rightarrow_a (q'_p)_{p \in dom(a)} \Leftrightarrow \forall p \in dom(a) : q_p \xrightarrow{a}_p q'_p.$$

It is clear then that global transition system  $(Q, \Sigma, \rightarrow, I)$  can be simulated by the synchronization specific to asynchronous automata:

$$(q_p)_{p \in Proc} \xrightarrow{a} (q'_p)_{p \in Proc} \Leftrightarrow \begin{cases} (q_p)_{p \in dom(a)} \rightarrow_a (q'_p)_{p \in dom(a)} \text{ and} \\ q_p = q'_p \text{ for all } p \notin dom(a). \end{cases}$$

On the other hand, consider again the distribution of Example 3.14 and the asynchronous automaton represented in Figure 3.4 obtained from the local state spaces

$$Q_1 = \{0, 1, 2\} \text{ and } Q_2 = \{0, 1, 2\},$$

and the local transition relations associated with the actions as:

$$\begin{array}{ll} 0 \xrightarrow{wall} 1 & \text{in state space } Q_1 \text{ (} dom(wall) = \{1\}\text{),} \\ 1 \xrightarrow{floor} 2 & \text{in state space } Q_2 \text{ (} dom(floor) = \{2\}\text{), and} \\ (0,0) \xrightarrow{roof} (2,1) \text{ and } (1,0) \xrightarrow{roof} (1,2) & \text{in state space } Q_1 \times Q_2 \text{ (} dom(roof) = \{1,2\}\text{).} \end{array}$$

It is easy to see that the ‘modern architecture’ of Figure 3.4 is indeed the global transition system of the synchronization of the above transition relations. For instance, the only transition labeled with *floor* is  $(2,1) \xrightarrow{floor} (2,2)$  because  $(2,1)$  is the only global reachable state (from  $(0,0)$ ) that has the second component 1 as required by  $1 \xrightarrow{floor} 2$ .

We prove now that the above transition system cannot be simulated by any synchronous product.<sup>1</sup> By contradiction, suppose that such synchronous product exists.

<sup>1</sup>An alternative proof can be given once we introduce characterizations of the global transition systems of synchronous products (see Theorem 3.26).



Then, there exist two workers (i.e., local processes) that synchronize to execute the given ‘architecture’ (i.e., global transition system). Since  $(0, 0) \xrightarrow{wall} (1, 0) \xrightarrow{roof} (1, 2)$  and  $\Sigma_{loc}(1) = \{wall, roof\}$ , the schedule of the first worker (i.e., the first local transition relation) must include the sequence  $wall.roof$ . Since  $(0, 0) \xrightarrow{roof} (2, 1) \xrightarrow{floor} (2, 2)$  and  $\Sigma_{loc}(2) = \{floor, roof\}$ , the schedule of the second worker (i.e., the second local transition relation) must include the sequence  $roof.floor$ . Since the synchronization model is weaker than the one of asynchronous automata, the two workers only agree that they synchronize to construct a roof ( $dom(roof) = \{1, 2\}$ ) without exchanging any other information, so the following scenario is possible:

The first worker rises a wall, then both workers team up (i.e., synchronize) to construct a roof, and at that point the second worker continues his schedule and builds a floor.

The above will generate the sequence

$wall.roof.floor,$

which contradicts the original plan of the ‘building’ in Figure 3.4. □

As a final notion of this section, by definition, the behavior of an asynchronous automaton is given by the language that its global transition system accepts:

**Definition 3.20 (The language of an asynchronous automaton)**

The *language accepted* by the asynchronous automaton  $\mathcal{AA} = (Q, \Sigma, \rightarrow, I)$  is simply the language of its transition system:

$$L(\mathcal{AA}) := \{w \in \Sigma^* \mid \exists q^{in} \in I, q \in Q : q^{in} \xrightarrow{w} q\}.$$

A finer control over the accepted language is obtained adding a set of global final set  $F \subseteq Q$  and modifying the definition accordingly:

$$L(\mathcal{AA}, F) := \{w \in \Sigma^* \mid \exists q^{in} \in I, q^{fin} \in F : q^{in} \xrightarrow{w} q^{fin}\}.$$

In fact, Zielonka included a set of final states in the original definition of asynchronous automata [Zie87]. This refinement makes the model expressive enough that the class of languages accepted by asynchronous automata *with final states* coincides with the class of regular trace-closed languages. This solved the long-standing problem of finding a natural distributed model to exactly capture the regular trace-closed behaviors. Nonetheless, we will try to avoid a set of global final states for reasons given in Section 3.4.1.

### 3.3 Shapes

We know now that the distributed transition systems consist of local processes collaborating to execute actions. The first natural questions are: ‘What are the characteristics of the resulted global state space?’ and ‘Can we find a general pattern that allows us



to recognize that a certain transition system originated from a distributed activity?' In other words: What is the global 'shape' of a distributed transition system?

In this section we examine structural properties of the global transition system of a distributed system and present characterizations for them from the literature. These characterizations provide theoretical bases for the synthesis of distributed transition systems (Section 3.5).

### 3.3.1 Diamonds *et al.*

A distributed system follows a distribution pattern. In turn, the distribution hides a natural independence relation between the actions (see end of Section 3.2.1), which can be uncovered considering the domains of the actions:

$$a\parallel b \Leftrightarrow \text{dom}(a) \cap \text{dom}(b) = \emptyset.$$

Thus we obtain a concurrent alphabet  $(\Sigma, \parallel)$  (Definition 3.1) telling which pairs of actions can be executed in parallel. We will see how this parallelism is reflected in our models of distributed transition systems.

Looking at the Definitions 3.15 and 3.18, we notice that the synchronizations on common actions depend on the domain of the actions. Two independent actions have disjoint domains, which implies that the synchronizations on the two actions do not interfere. This means that in a certain global state the two actions may be executed in any order (cf. the trace equivalence in Definition 3.3). Moreover, if in a certain global state both actions are enabled, the execution of one of them does not hinder the execution of the other one. These two properties are formally captured by:

#### Definition 3.21 (Independent and forward diamond properties)

Let  $(\Sigma, \parallel)$  be a concurrent alphabet and  $TS = (Q, \Sigma, \rightarrow, I)$  a transition system. We say that  $TS$  satisfies the *independent diamond* (ID) and *forward diamond* (FD) properties if for any  $q_1, q_2, q_3 \in Q$  and  $a, b \in \Sigma$  we have:

- ID :  $q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_3 \wedge a\parallel b \Rightarrow \exists q_4 \in Q : q_1 \xrightarrow{b} q_4 \xrightarrow{a} q_3$
- FD :  $q_1 \xrightarrow{a} q_2 \wedge q_1 \xrightarrow{b} q_3 \wedge a\parallel b \Rightarrow \exists q_4 \in Q : q_2 \xrightarrow{b} q_4 \wedge q_3 \xrightarrow{a} q_4.$

(Figure 3.5 presents the above properties graphically.)

It is easy to see that every distributed transition system satisfies the ID and FD rules:

**Proposition 3.22** *Let  $(\Sigma, Proc, \Delta)$  be a distribution and  $\parallel$  its corresponding independence relation. Then, every synchronous product  $\mathcal{SP} = (Q, \Sigma, \rightarrow, I)$  over  $\Delta$  satisfies the ID and FD properties. Similarly, every asynchronous automaton  $\mathcal{AA} = (Q, \Sigma, \rightarrow, I)$  over  $\Delta$  satisfies the ID and FD properties.*

**Proof.** It is enough to show that ID and FD hold for asynchronous automata because the same will hold for synchronous products using Remark 3.19.

Let  $\mathcal{AA} = (Q, \Sigma, \rightarrow, I)$  be an asynchronous automaton over distribution  $(\Sigma, Proc, \Delta)$  and  $q_1 = (q_{1p})_{p \in Proc}$ ,  $q_2 = (q_{2p})_{p \in Proc}$ , and  $q_3 = (q_{3p})_{p \in Proc}$  be three global states of  $Q$ . Then, we have:

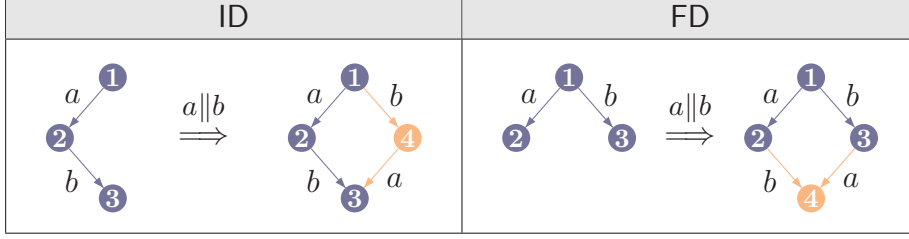


Figure 3.5: The independent and forward diamond properties

Formal description	Graphical description
$\Sigma := \{a, b\}, \quad a  b$ $Q := \{0, 1\}, \quad I := \{0\}$ $\rightarrow := \{0 \xrightarrow{a,b} 1 \xrightarrow{a,b} 0\}$	

Figure 3.6: A transition system satisfying ID and FD, but not isomorphic to an asynchronous automaton

- If  $q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_3$  and  $a||b$ , it is easy to see that the global state  $q_4 = (q_{4p})_{p \in Proc}$  given by:  $q_{4p} := q_{3p}$  for  $p \in dom(b)$  and  $q_{4p} := q_{1p}$  for  $p \notin dom(b)$ , satisfies ID.
- If  $q_1 \xrightarrow{a} q_2, q_1 \xrightarrow{b} q_3$ , and  $a||b$ , it is easy to see that the global state  $q_4 = (q_{4p})_{p \in Proc}$  given by:  $q_{4p} := q_{2p}$  for  $p \in dom(a)$  and  $q_{4p} := q_{3p}$  for  $p \notin dom(a)$ , satisfies FD.  $\square$

Nevertheless, the ID and FD properties are not sufficient to characterize the shape of (the global transition systems of) distributed transition systems as the simple example below witnesses:

**Example 3.23** [Mor98] Let  $TS = (Q, \Sigma, \rightarrow, I)$  be the transition system of Figure 3.6. It is easy to show that  $TS$  is the ‘smallest’ transition system such that  $TS$  satisfies ID and FD, but there is no asynchronous automaton isomorphic to it (and by Remark 3.19, also no synchronous product).

**Proof.** Since it is obvious that  $TS$  satisfies ID and FD, we only have to prove that there is no asynchronous automaton isomorphic to it. By contradiction, suppose there exists such an asynchronous automaton whose distribution pattern is compliant with the given independence relation  $a||b$ , i.e.,  $dom(a) \cap dom(b) = \emptyset$ .

From  $0 \xrightarrow{a} 1$  and the definition of synchronization for asynchronous automata (Definition 3.18) we have that the local states of (the global states) 0 and 1 must be equal on all processes of  $Proc \setminus dom(a)$ . Similarly, from  $0 \xrightarrow{b} 1$ , the local states of 0 and 1 must be equal on all processes of  $Proc \setminus dom(b)$ . Summing up, the local states of 0 and 1 coincide on all processes of  $(Proc \setminus dom(a)) \cup (Proc \setminus dom(b)) = Proc \setminus (dom(a) \cap dom(b)) = Proc$ , because  $dom(a) \cap dom(b) = \emptyset$ . At this point we reached the contradiction that the *distinct* global states 0 and 1 have all the local states equal.  $\square$

Sometimes we can infer structural properties of the global transition relation from the local ones, as shown below.

For synchronous products, we can lift the properties of being deterministic or acyclic from the local transition relations to the global generated one:

**Proposition 3.24** *Let  $\mathcal{SP} = (Q, \Sigma, \rightarrow, I)$  be a synchronous product over a distribution  $(\Sigma, Proc, \Delta)$  and let  $(\rightarrow_p)_{p \in Proc}$  be its local transition relations (according to Definition 3.15). Then we have:*

1. *If all  $(\rightarrow_p)_{p \in Proc}$  are deterministic<sup>1</sup>, then their global synchronization  $\rightarrow$  is also deterministic.*
2. *If all  $(\rightarrow_p)_{p \in Proc}$  are acyclic, then their global synchronization  $\rightarrow$  is also acyclic.*
3. *The converse implications of 1. and 2. above are not true in general.*

**Proof.** Easy:

1. Let  $q = (q_p)_{p \in Proc} \xrightarrow{a} q' = (q'_p)_{p \in Proc}$  and  $q = (q_p)_{p \in Proc} \xrightarrow{a} q'' = (q''_p)_{p \in Proc}$  in  $\rightarrow$ . Then, according to Definition 3.15, we have that  $q_p \xrightarrow{a} q'_p$  and  $q_p \xrightarrow{a} q''_p$  for all  $p \in dom(a)$  (\*) and  $q'_p = q_p = q''_p$  for all  $p \notin dom(a)$  (\*\*). From (\*) and the deterministic property of the local transition relations, we necessarily have that  $q'_p = q''_p$  for all  $p \in dom(a)$ , which together with (\*\*) implies  $q' = q''$ .
2. By contradiction, suppose that  $\rightarrow$  contains the cycle  $q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \dots \xrightarrow{a_{n-1}} q_n \xrightarrow{a_n} q_1$  (with  $n \geq 1$ ). Then, it not difficult to show that  $\rightarrow_p$ , where  $p \in dom(a_1)$ , contains a cycle (the projection on the component  $p$  of the global cycle). But this is a contradiction.
3. We choose the distribution of  $\Sigma := \{a\}$  over two processes  $Proc := \{1, 2\}$ , with  $dom(a) := \{1, 2\}$ . Also, for each process we choose the local state spaces  $Q_1 := \{q_1, q'_1\}$  with  $\rightarrow_1 := \{(q_1, a, q_1), (q_1, a, q'_1)\}$  and  $Q_2 := \{q_2\}$  with  $\rightarrow_2 := \emptyset$ .  
If we take  $I := \{(q_1, q_2)\}$ , the global transition relation  $\rightarrow$  is empty (because  $\rightarrow_2 = \emptyset$ ), so  $\rightarrow$  is both deterministic and acyclic. On the other hand, the first local transition  $\rightarrow_1$  is neither deterministic, nor acyclic.  $\square$

For asynchronous automata (which have a finer synchronization) the situation is slightly different than for synchronous products:

**Proposition 3.25** *Let  $\mathcal{AA} = (Q, \Sigma, \rightarrow, I)$  be an asynchronous automaton over a distribution  $(\Sigma, Proc, \Delta)$  and let  $(\rightarrow_a)_{a \in \Sigma}$  be its local transition relations (according to Definition 3.18). Then we have:*

1. *If all  $(\rightarrow_a)_{a \in \Sigma}$  are deterministic<sup>2</sup>, then their global synchronization  $\rightarrow$  is also deterministic<sup>1</sup>.*

<sup>1</sup>A labeled transition relation  $\rightarrow \subseteq Q \times \Sigma \times Q$  is *deterministic* if and only if for all  $q, q', q'' \in Q$  and  $a \in \Sigma$  such that  $q \xrightarrow{a} q'$  and  $q \xrightarrow{a} q''$ , we necessarily have  $q' = q''$ .

<sup>2</sup>An unlabeled transition relation  $\rightarrow \subseteq S \times S$  is *deterministic* if and only if for all  $s, s', s'' \in S$  such that  $s \rightarrow s'$  and  $s \rightarrow s''$ , we necessarily have  $s' = s''$ .

2. If all  $(\rightarrow_a)_{a \in \Sigma}$  are acyclic, then their global synchronization  $\rightarrow$  is not necessarily acyclic.
3. If  $\rightarrow$  is deterministic (respectively, acyclic), then not necessarily all  $(\rightarrow_a)_{a \in \Sigma}$  are also deterministic (respectively, acyclic).

**Proof.** Easy:

1. Let  $q = (q_p)_{p \in Proc} \xrightarrow{a} q' = (q'_p)_{p \in Proc}$  and  $q = (q_p)_{p \in Proc} \xrightarrow{a} (q''_p)_{p \in Proc}$  in  $\rightarrow$ . Then, according to Definition 3.18, we have that  $(q_p)_{p \in dom(a)} \rightarrow_a (q'_p)_{p \in dom(a)}$  and  $(q_p)_{p \in dom(a)} \rightarrow_a (q''_p)_{p \in dom(a)}$  (\*), and  $q'_p = q_p = q''_p$  for all  $p \notin dom(a)$  (\*\*). From (\*) and the deterministic property of the local transition relations, we necessarily have that  $(q'_p)_{p \in dom(a)} = (q''_p)_{p \in dom(a)}$ , which together with (\*\*) implies  $q' = q''$ .
2. Take for instance  $\Sigma = \{a, b\}$ ,  $Proc = \{1\}$ , and  $dom(a) = dom(b) = 1$ . Also, take  $Q_1 := \{0, 1\}$  with  $\rightarrow_a := \{(0, 1)\}$  and  $\rightarrow_b := \{(1, 0)\}$ . For  $I := \{0\}$ , we have  $Q = Q_1$  and  $\rightarrow = \{(0, a, 1), (1, b, 0)\}$ . Then,  $\rightarrow$  is cyclic (because  $0 \xrightarrow{a} 1 \xrightarrow{b} 0$ ), while the local transitions relations  $\rightarrow_a$  and  $\rightarrow_b$  are both acyclic.
3. In this case, the proof uses the fact in Definition 3.18 that the global state space is *reachable* and  $\rightarrow$  is restricted to this part only:

We choose  $\Sigma := \{a\}$ ,  $Proc := \{1\}$ , with  $dom(a) = \{1\}$ , and  $Q_1 := \{0, 1, 2\}$  with  $\rightarrow_a := \{(1, 1), (1, 2)\}$ . If we take  $I := \{0\}$ , then  $Q := \{0\}$  and the global transition relation  $\rightarrow$  is empty (because there is no  $s$  such that  $0 \rightarrow_a s$ ), so  $\rightarrow$  is both deterministic and acyclic. On the other hand, the local transition relation  $\rightarrow_a$  is neither deterministic, nor acyclic.  $\square$

Example 3.23 testifies that the ID and FD properties are not sufficient to characterize the shape of (the global transition systems of) distributed transition systems. Nevertheless, precise characterizations can be found in the literature [Mor98, CMT99, Mor99b, Muk02] and are presented in the following two sections (3.3.2 and 3.3.3).

### 3.3.2 Synchronous Products of Transition Systems

We present a characterization theorem for the global transition system of synchronous products. More precisely, [CMT99, Muk02] give a set of three necessary and sufficient conditions that a transition system must satisfy in order to be isomorphic to a synchronous product over a given distribution. If such isomorphic synchronous product exists, then each of its local state spaces  $Q_p$  (cf. Definition 3.15) is constructed as the quotient of the input state space under a local equivalence relation  $\equiv_p$ . These equivalences must be chosen such that the following hold:

1. an  $a$ -labeled transition does not affect the local states of the processes *not contained* in  $dom(a)$ ;
2. the global state space is no more than the cartesian product of the  $Q_p$ 's, and

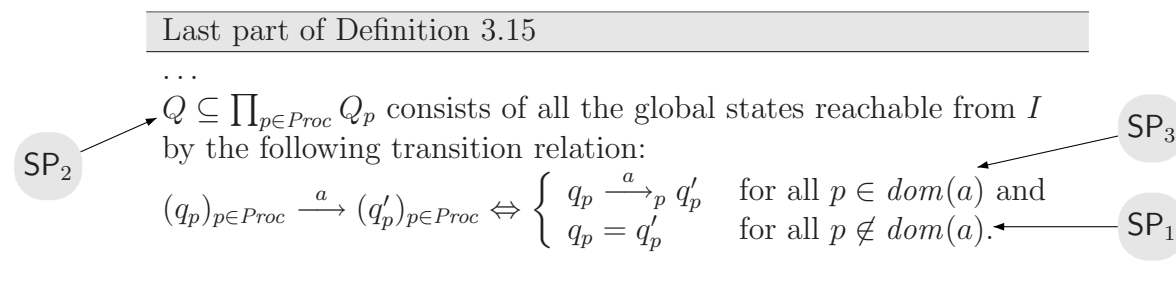


Figure 3.7: The relation between conditions SP<sub>1</sub>–SP<sub>3</sub> and the definition of synchronous products

3. for an action  $a \in \Sigma$ , if the local states of the processes in  $dom(a)$  are able to perform an  $a$ -labeled transition, then a global synchronization must also be possible.

Before we give the theorem, we need to introduce an abbreviation:

**Notation.** For any two sets of indexes  $I$  and  $J$  such that  $J \subseteq I$  and an indexed family of binary relations  $(\equiv_i)_{i \in I}$ , the expression  $(q_1 \equiv_J q_2)$  abbreviates  $(\forall j \in J : q_1 \equiv_j q_2)$ .

**Theorem 3.26** [CMT99, Muk02] *Let  $(\Sigma, Proc, \Delta)$  be a distribution and  $TS = (Q, \Sigma, \rightarrow, I)^1$  be a transition system. Then, the following are equivalent:*

- (i) *TS is isomorphic to a synchronous product of transition systems over  $\Delta$*
- (ii) *For each  $p \in Proc$ , there exists an equivalence relation  $\equiv_p \subseteq Q \times Q$  such that the following conditions hold (for any  $q_1, q_2 \in Q$  and  $a \in \Sigma$ ):*

SP<sub>1</sub> : *If  $q_1 \xrightarrow{a} q_2$ , then  $q_1 \equiv_{Proc \setminus dom(a)} q_2$ .*

SP<sub>2</sub> : *If  $q_1 \equiv_{Proc} q_2$ , then  $q_1 = q_2$ .*

SP<sub>3</sub> : *Let  $a \in \Sigma$  and  $q \in Q$ . If for each  $p \in dom(a)$ , there exist  $q_p, q'_p \in Q$  such that  $q_p \xrightarrow{a} q'_p$  and  $q \equiv_p q_p$ , then for each choice of such  $q_p$ 's and  $q'_p$ 's, there exists  $q' \in Q$  such that  $q \xrightarrow{a} q'$  and  $q' \equiv_p q'_p$  for each  $p \in dom(a)$ .*

Since the above theorem is not very easy to digest, we link in Figure 3.7 the definition of a synchronized product with the three conditions of Theorem 3.26 (see also the informal discussion preceding the theorem). Below we give an example that shows Theorem 3.26 ‘at work’:

**Example 3.27** Consider again the transition system of Figure 3.4 together with the distribution from Example 3.14, now with the austere look of Figure 3.8 (obtained by

<sup>1</sup>In [CMT99, Muk02], Theorem 3.26 is restricted to the case when  $|I| = 1$  (i.e., all transition systems have only one initial state). By inspecting the proof of [CMT99], it is easy to see that the theorem holds in fact for an arbitrary  $I$ .

We also mention that in [CMT99, Muk02], the transition system  $TS$  is required to be *reachable*, but this is indeed the case in our framework: We assumed in Section 2.2 that all transition systems we work with are finite and reachable.

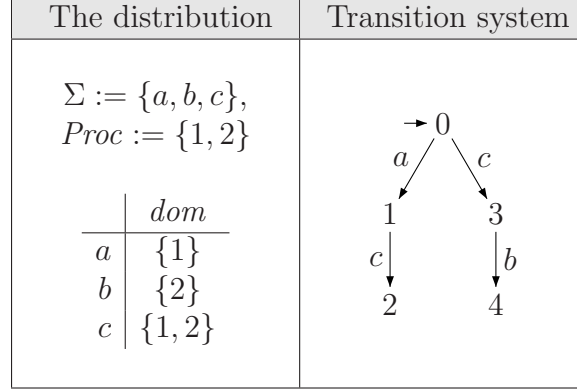


Figure 3.8: An asynchronous automaton that is not a synchronous product

relabeling *wall* by *a*, *floor* by *b*, and *roof* by *c*). In the proof of Remark 3.19, we gave intuitive arguments why the transition system is not (isomorphic to) a synchronous product. We can now provide an alternative formal proof using Theorem 3.26.

By contradiction, suppose that the transition system of Figure 3.8 is isomorphic to a synchronous product over the distribution of Figure 3.8. Then, there exists a set of equivalences  $(\equiv_p)_{p \in Proc}$  such that  $SP_1$ – $SP_3$  are satisfied and therefore we have:

- From  $SP_1$  applied to  $0 \xrightarrow{a} 1$ , we have that  $0 \equiv_2 1$  (recall that  $dom(a) = \{1\}$ , so  $Proc \setminus dom(a) = \{2\}$ ).
- Next, we apply  $SP_3$  to the action *c* and the state  $q := 0$  in the following way:

For each process  $p$  of  $dom(c) = \{1, 2\}$ , we choose

- for  $p = 1$ :  $0 \xrightarrow{c} 3$  with  $0 \equiv_1 0$  (reflexivity of  $\equiv_1$ ) and
- for  $p = 2$ :  $1 \xrightarrow{c} 2$  with  $0 \equiv_2 1$  (the application of  $SP_1$  above).

Then, there exists necessarily  $q'$  such that  $0 \xrightarrow{c} q'$  and  $q' \equiv_1 3$  and  $q' \equiv_2 2$ . Looking at the transition relation, we see that the only  $q'$  such that  $0 \xrightarrow{c} q'$  is the state 3. So, we have  $3 \equiv_1 3$  and  $3 \equiv_2 2$  (\*).

- Finally, we apply  $SP_3$  to the action *b* and the state 2. Since  $dom(b) = \{2\}$ , we choose  $3 \xrightarrow{b} 4$  for which we have  $2 \equiv_2 3$  (from (\*) above). Then, there must exist a  $q''$  such that  $2 \xrightarrow{b} q''$  and  $q'' \equiv_2 4$ . But there is *no transition* going out of state 2, so we reached a contradiction.  $\square$

**Remark 3.28** At this point, we note that the characterization given by Theorem 3.26 is effective and constructive:

- *effective*, because we can always decide whether a transition system is isomorphic to a synchronous product or not (the number of all possible local equivalences is finite and we can test for each combination the  $SP_1$ – $SP_3$  conditions) and

- *constructive*, because if there exists an isomorphic synchronous product, then we can derive its local transition systems by taking the quotients of the original transition system w.r.t. the local equivalences (see [CMT99, Muk02] for more details).

The above remark solves what is called the *synthesis problem* for synchronous products modulo isomorphism (see Section 3.5) and whose computational complexity will be discussed in Chapter 4.

In the special case of *deterministic* transition systems, Morin gave a similar characterization in [Mor98]. (In fact, the result for the deterministic case was published before the general case.)

**Theorem 3.29** [Mor98] *Let  $(\Sigma, Proc, \Delta)$  be a distribution and  $TS = (Q, \Sigma, \rightarrow, \{q^{in}\})$  be a deterministic transition system. Then, we have:*

1. *There exists the least family of equivalences  $(\equiv_p)_{p \in Proc}$  over the states of  $Q$  such that the following conditions hold (for any  $q_1, q'_1, q_2, q'_2 \in Q$ ,  $a \in \Sigma$ , and  $p \in Proc$ ):*

DSP<sub>1</sub> : *If  $q_1 \xrightarrow{a} q_2$ , then  $q_1 \equiv_{Proc \setminus dom(a)} q_2$ .*

DSP<sub>2</sub> : *If  $q_1 \xrightarrow{a} q'_1$ ,  $q_2 \xrightarrow{a} q'_2$ , and  $q_1 \equiv_p q_2$ , then  $q'_1 \equiv_p q'_2$ .*

2. *The following are equivalent:*

(i)  *$TS$  is isomorphic to a synchronous product of transition systems over  $\Delta$ .*

(ii) *For the equivalences  $(\equiv_k)_{k \in Proc}$  from 1. the following conditions hold:*

DSP<sub>3</sub> : *If  $q_1 \equiv_{Proc} q_2$ , then  $q_1 = q_2$ .*

DSP<sub>4</sub> : *Let  $a \in \Sigma$  and  $q \in Q$ . If for each  $p \in dom(a)$ , there exist  $q_p, q'_p \in Q$  such that  $q_p \xrightarrow{a} q'_p$  and  $q \equiv_p q_p$ , then there exists  $q' \in Q$  such that  $q \xrightarrow{a} q'$ .*

We note that SP<sub>1</sub>, respectively SP<sub>2</sub> are the same as DSP<sub>1</sub>, respectively DSP<sub>3</sub>. Moreover, for a *deterministic* transition system, SP<sub>3</sub> is equivalent to DSP<sub>2</sub> and DSP<sub>4</sub>. Even if the conditions seem to be equivalent, the advantage of Theorem 3.29 over Theorem 3.26 is given by the elimination of the existential quantifier on the class of local equivalences. The first part of Theorem 3.29 constructs a set of equivalences (computed as a fixed point) which are further tested in the second part. This should give a more efficient implementation compared to the ‘guessing’ of local equivalences suggested by Theorem 3.26. The advantage of Theorem 3.29 over Theorem 3.26 is confirmed theoretically in Chapter 4 and practically in Chapter 6.

### 3.3.3 Asynchronous Automata

A characterization similar to Theorem 3.26 can be given for asynchronous automata:

**Theorem 3.30** [Mor99b, Muk02] *Let  $(\Sigma, Proc, \Delta)$  be a distribution and  $TS = (Q, \Sigma, \rightarrow, I)^1$  be a transition system. Then, the following are equivalent:*

<sup>1</sup>We mention that in [Mor99b, Muk02], the transition system  $TS$  is required to be *reachable*, but this is indeed the case in our framework: We assumed in Section 2.2 that all transition systems we work with are finite and reachable.



- (i)  $TS$  is isomorphic to an asynchronous automaton over  $\Delta$
- (ii) For each  $p \in Proc$ , there exists an equivalence relation  $\equiv_p \subseteq Q \times Q$  such that the following conditions hold (for any  $q_1, q_2 \in Q$  and  $a \in \Sigma$ ):

$AA_1$  : If  $q_1 \xrightarrow{a} q_2$ , then  $q_1 \equiv_{Proc \setminus dom(a)} q_2$ .

$AA_2$  : If  $q_1 \equiv_{Proc} q_2$ , then  $q_1 = q_2$ .

$AA_3$  : If  $q_1 \xrightarrow{a} q'_1$  and  $q_1 \equiv_{dom(a)} q_2$ , then there exists  $q'_2 \in Q$  such that  $q_2 \xrightarrow{a} q'_2$  and  $q'_2 \equiv_{dom(a)} q'_1$ .

Since  $SP_1$ , respectively  $SP_2$ , is the same as  $AA_1$ , respectively  $AA_2$ , the difference between Theorem 3.26 and Theorem 3.30 is given by  $SP_3$  vs.  $AA_3$ . More precisely, it is easy to see that  $SP_3$  implies  $AA_3$ . This observation gives an alternative proof to Remark 3.19 saying that any synchronous product accepts a description as an asynchronous automaton.

As a simple (pedagogical) exercise, we show how Theorem 3.30 can be used to prove structural properties of distributed transition systems:

**Remark 3.31** Let  $(\Sigma, Proc, \Delta)$  be a distribution and  $TS = (Q, \Sigma, \rightarrow, I)$  be (the global transition system of) an asynchronous automata. Then:

1.  $TS$  satisfies ID and FD (Proposition 3.22).
2. There exist no  $q, q' \in Q$ ,  $a, b \in \Sigma$  such that  $q \xrightarrow{a,b} q'$ ,  $q \neq q'$ , and  $a \parallel b$ .

As a consequence, Example 3.23 cannot be an asynchronous automata (because  $0 \xrightarrow{a,b} 1$  and  $a \parallel b$ ).

**Proof.** Easy:

1. We use Theorem 3.30 to give an alternative proof to Proposition 3.22. We consider only the FD case (ID being similar).

For  $q_1 \xrightarrow{a} q_2$ ,  $q_1 \xrightarrow{b} q_3$ , and  $a \parallel b$ , we show that there exists  $q_4$  such that  $q_2 \xrightarrow{b} q_4$  and  $q_3 \xrightarrow{a} q_4$  (cf. Figure 3.5). Since  $TS$  is an asynchronous automaton over  $\Delta$ , according to Theorem 3.30, there must exist a family of local equivalences  $(\equiv_p)_{p \in Proc}$  satisfying  $AA_1$ – $AA_3$ . We apply the three conditions in turn as follows:

- From  $AA_1$  applied to  $q_1 \xrightarrow{a} q_2$ , we have  $q_1 \equiv_{Proc \setminus dom(a)} q_2$  and from  $a \parallel b$ , by definition, we have  $dom(a) \cap dom(b) = \emptyset$ , which is equivalent to  $dom(b) \subseteq (Proc \setminus dom(a))$ . Thus, we have  $q_1 \equiv_{dom(b)} q_2$ .

From  $AA_1$  applied to  $q_1 \xrightarrow{b} q_3$ , we similarly obtain  $q_1 \equiv_{dom(a)} q_3$ .

- From  $AA_3$  applied to  $q_1 \xrightarrow{b} q_3$  and  $q_1 \equiv_{dom(b)} q_2$ , there exists  $q'_4$  such that  $q_2 \xrightarrow{b} q'_4$  and  $q'_4 \equiv_{dom(b)} q_3$ . Moreover, we have that  $q'_4 \equiv_{Proc \setminus dom(a)} q_3$  (\*): If  $p \in (Proc \setminus dom(a)) \setminus dom(b)$ , then  $q'_4 \equiv_p q_3$  from transitivity applied to  $q'_4 \equiv_p q_2$ ,  $q_2 \equiv_p q_1$ , and  $q_1 \equiv_p q_3$  (the last three equivalences are obtained applying  $AA_1$  to  $q_2 \xrightarrow{b} q'_4$ ,  $q_1 \xrightarrow{a} q_2$ , respectively  $q_1 \xrightarrow{b} q_3$ ).



From  $\text{AA}_3$  applied to  $q_1 \xrightarrow{a} q_2$  and  $q_1 \equiv_{\text{dom}(a)} q_3$ , there exists  $q_4''$  such that  $q_3 \xrightarrow{a} q_4''$  and  $q_4'' \equiv_{\text{dom}(a)} q_2$ . From  $\text{AA}_1$  applied to  $q_3 \xrightarrow{a} q_4''$ , we have that  $q_3 \equiv_{\text{Proc} \setminus \text{dom}(a)} q_4''$ , which together with (\*) gives (by transitivity) that  $q_4' \equiv_{\text{Proc} \setminus \text{dom}(a)} q_4''$ . Similarly, we have also  $q_4' \equiv_{\text{Proc} \setminus \text{dom}(b)} q_4''$ . From the last two equivalences and  $\text{dom}(a) \cap \text{dom}(b) = \emptyset$ , we have that  $q_4' \equiv_{\text{Proc}} q_4''$ .

- Finally, from  $\text{AA}_2$  applied to  $q_4' \equiv_{\text{Proc}} q_4''$ , we have that  $q_4' = q_4''$  and we denote this single state by  $q_4$ . Then, this state  $q_4$  satisfies condition  $\text{FD}$  for  $q_1 \xrightarrow{a} q_2$ ,  $q_1 \xrightarrow{b} q_3$ , and  $a \parallel b$  [q.e.d.].

2. By contradiction, assume there exist  $q \neq q'$  such that  $q \xrightarrow{a,b} q'$  and  $a \parallel b$ . Applying  $\text{AA}_1$  to  $q_1 \xrightarrow{a} q_2$ , respectively  $q_1 \xrightarrow{b} q_2$  we easily obtain that  $q_1 \equiv_{\text{Proc}} q_2$ , which by  $\text{AA}_2$  implies that  $q_1 = q_2$ , which is a contradiction.  $\square$

For *deterministic* transition systems, a characterization similar to Theorem 3.29 is available also for asynchronous automata:

**Theorem 3.32** [Mor98] *Let  $(\Sigma, \text{Proc}, \Delta)$  be a distribution and  $TS = (Q, \Sigma, \rightarrow, \{q^{in}\})$  be a deterministic transition system. Then, we have:*

1. *There exists the least family of equivalences  $(\equiv_p)_{p \in \text{Proc}}$  over the states of  $Q$  such that the following conditions hold (for any  $q_1, q_1', q_2, q_2' \in Q$ ,  $a \in \Sigma$ , and  $p \in \text{Proc}$ ):*

$\text{DAA}_1$  : *If  $q_1 \xrightarrow{a} q_2$ , then  $q_1 \equiv_{\text{Proc} \setminus \text{dom}(a)} q_2$ .*

$\text{DAA}_2$  : *If  $q_1 \xrightarrow{a} q_1'$ ,  $q_2 \xrightarrow{a} q_2'$ , and  $q_1 \equiv_{\text{dom}(a)} q_2$ , then  $q_1' \equiv_{\text{dom}(a)} q_2'$ .*

2. *The following are equivalent:*

(i)  *$TS$  is isomorphic to an asynchronous automaton over  $\Delta$ .*

(ii) *For the equivalences  $(\equiv_k)_{k \in \text{Proc}}$  from 1. the following conditions hold:*

$\text{DAA}_3$  : *If  $q_1 \equiv_{\text{Proc}} q_2$ , then  $q_1 = q_2$ .*

$\text{DAA}_4$  : *If  $q_1 \xrightarrow{a} q_1'$  and  $q_1 \equiv_{\text{dom}(a)} q_2$ , then there exists  $q_2' \in Q$  such that  $q_2 \xrightarrow{a} q_2'$ .*

We end with a ‘genealogical’ remark. The results of Sections 3.3.2 and 3.3.3 have their roots in the *theory of regions* [ER90], proposed by Ehrenfeucht and Rozenberg in the early 90s to characterize (modulo isomorphism) the reachability graphs of Petri nets<sup>1</sup>. The connection between the Theorems 3.26, 3.30 and the theory of regions is given by the fact that the synchronous products and the asynchronous automata can be seen as particular cases of (1-safe) Petri nets.

<sup>1</sup>Loosely speaking, a *region* is a subset of states that is consistently entered, respectively exited by transitions having the same label.

### 3.4 Languages

In this section we study the languages accepted by our two models of distributed systems. We start with a short discussion regarding the final states of a distributed system. Then, we show the relation between the structural diamond properties of transition systems and closure properties of their accepted languages. We continue the presentation with characterizations of the languages of distributed transition systems with remarks about special cases. We finish with an expressiveness comparison of various classes of languages.

Before we start, we introduce notations for the classes of languages accepted by distributed transition systems with different constraints.

**Notation.** Let us fix a distribution  $(\Sigma, Proc, \Delta)$ . Then, for each of the theoretical models  $M$  below:

- NSP   nondeterministic synchronous products,
- DSP   deterministic synchronous products,
- NAA   nondeterministic asynchronous automata, and
- DAA   deterministic asynchronous automata,

we denote by

$$\mathcal{L}(M), \text{ respectively } \mathcal{L}(M, F)$$

the class of languages accepted by the model  $M$  over  $\Delta$  without, respectively with a set of final states.

Moreover, we consider also *acyclic* versions of the above four classes of systems, and the notation is obtained by adding the prefix ‘A’. So, we have

- ANSP   acyclic nondeterministic synchronous products,
- ADSP   acyclic deterministic synchronous products,
- ANAA   acyclic nondeterministic asynchronous automata, and
- ADAA   acyclic deterministic asynchronous automata,

For instance,  $\mathcal{L}(DAA, F)$  denotes the class of all languages  $L = L(\mathcal{AA}, F)$  where  $\mathcal{AA}$  is a deterministic asynchronous automaton over  $\Delta$  with a set of final states  $F$ , while  $\mathcal{L}(ANSP)$  denotes the class of all languages  $L = L(\mathcal{SP})$  where  $\mathcal{SP}$  is an acyclic (nondeterministic) synchronous product over  $\Delta$ .

A small clarification: As in classic automata theory, when we use the term ‘nondeterministic’ for a (distributed) transition system, we do not mean a system that is ‘not deterministic’ (thus excluding the deterministic systems from the class), but we mean a general transition system that may as well be deterministic or not. According to Definitions 3.15 and 3.18, the usage of the descriptive adjective ‘nondeterministic’ is redundant, but may help to better differentiate various classes of models.

### 3.4.1 Final States

In this subsection we shortly discuss the appropriateness of final/accepting states to distributed systems.

The behavior or the language of a system is given by the set of all its possible runs. A run has a starting point and an ending point. Therefore, we can ‘tune’ the language of a system by appropriately modifying the set of initial and final states. This works per se for sequential machines (Definition 2.13) and we could naturally extend the idea to distributed transition systems and choose a set of initial and final *global* states. This is a seamless theoretical approach, as one could simply observe the global transition system of a distributed system as a sequential system and apply the classic reasoning. Yet, there is a pragmatic objection to the above choice of *global* final states: An implementation of the mechanism of telling when we can accept a run implies the existence of a global controller that has access to *all* the local states and decides whether their tuple is accepting (i.e., final) or not. However, this is against the guiding principle that a distributed system involves only local synchronization. Also, it is unnatural to have a set of final global states that declare a posteriori that a run already executed is not valid because the reached global state is not final (the processes have only partial views and therefore cannot prevent possible executions that do not satisfy a global condition). Therefore:

In this thesis we will not to deal with final states and will observe *all* the possible executions of the distributed transition systems (see  $L(\mathcal{SP})$ , respectively  $L(\mathcal{AA})$  in Definitions 3.17, respectively 3.20). In other words, we consider by default all the reachable global states as accepting.

We mention two other related models that address the above mentioned problems regarding a set of global final states for asynchronous automata. In his paper [Zie89], Zielonka restricts the model of asynchronous automata with final states to those having the following *safety* property: from any global state reachable from an initial state, we can always reach a final state. The new class, called *safe* asynchronous automata, was shown to have the same expressive power as the asynchronous automata with final states, but only in case we allow *nondeterminism*. Another possibility, mentioned by Pighizzini in [Pig93a, Section 3.3], is to have a set of *local* final states for each process of the distribution and to consider the cartesian product of the sets of local final states as the set of global final states. We will further discuss these options at a later time.

### 3.4.2 Traces of Diamonds

In Section 3.3.1, we have seen how the independence relation between actions ‘watermarks’ the structures of distributed systems. In this section, we study the ‘reflexions’ of the independence relation in the languages of distributed transition systems and show the relation of these languages to the trace languages from Section 3.1.

In Section 3.3.1, the ID and FD diamond properties were shown to hold for the class of distributed systems over a concurrent alphabet  $(\Sigma, \parallel)$ . Since the behavior of a system depends of course on its shape, it is natural to investigate the properties of the languages of systems satisfying the diamond properties. Thus, it is easy to see that to ID it corresponds

the *trace-closure* property (Definition 3.5): For all  $w, w' \in \Sigma^*$ , and  $a, b \in \Sigma$ , it holds

$$wabw' \in L \wedge a\|b \Rightarrow wbaw' \in L.$$

On the other hand, to FD we can associate a notion of *forward closure*<sup>1</sup>:

**Definition 3.33 (Forward-closed language)**

Let  $(\Sigma, \parallel)$  be a concurrent alphabet. A language  $L \subseteq \Sigma^*$  is called *forward-closed* if for all  $w \in \Sigma^*$ , and  $a, b \in \Sigma$  it holds:

$$wa \in L \wedge wb \in L \wedge a\|b \Rightarrow wab \in L.$$

The following proposition confirms the relation between the above structural and language properties:

**Proposition 3.34** *For any concurrent alphabet  $(\Sigma, \parallel)$  and any transition system  $TS = (Q, \Sigma, \rightarrow, I)$ , we have:*

1. *If  $TS$  satisfies ID, then  $L(TS)$  is trace-closed.*
2. *If  $TS$  is deterministic and satisfies FD, then  $L(TS)$  is forward-closed.*
3. *If  $TS$  is nondeterministic and satisfies FD, then it is not always the case that  $L(TS)$  is forward-closed.*

**Proof.** 1. Let  $w, w' \in \Sigma^*$  and  $a, b \in \Sigma$  such that  $wabw' \in L(TS)$  and  $a\|b$ . Then, there exist  $q^{in} \in I$ ,  $q_1, q_2, q_3 \in Q$  such that  $q^{in} \xrightarrow{w} q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_3 \xrightarrow{w'} q_4$ . From  $q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_3$  and  $a\|b$ , by ID we have that there exists  $q'_2$  such that  $q_1 \xrightarrow{b} q'_2 \xrightarrow{a} q_3$ . So  $q^{in} \xrightarrow{w} q_1 \xrightarrow{b} q'_2 \xrightarrow{a} q_3 \xrightarrow{w'} q_4$ , which implies that  $wbaw' \in L(TS)$ .

2. Let  $w \in \Sigma^*$  and  $a, b \in \Sigma$  such that  $wa, wb \in L(TS)$  and  $a\|b$ . Since  $TS$  is deterministic, there exists a unique initial state  $q^{in}$  and *unique* states  $q_1, q_2, q_3 \in Q$  such that  $q^{in} \xrightarrow{w} q_1$ ,  $q_1 \xrightarrow{a} q_2$ , and  $q_1 \xrightarrow{b} q_3$ . From  $q_1 \xrightarrow{a} q_2$ ,  $q_1 \xrightarrow{b} q_3$ , and  $a\|b$ , by FD we have that there exists  $q_4 \in Q$  such that  $q_2 \xrightarrow{b} q_4$  and  $q_3 \xrightarrow{a} q_4$ . So  $q^{in} \xrightarrow{w} q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_4$  in  $TS$ , which implies that  $wab \in L(TS)$ .

3. For the concurrent alphabet  $\Sigma := \{a, b, c\}$  with  $a\|b$ , the nondeterministic transition systems of Figure 3.9 satisfy FD (and ID). On the other hand, the language accepted by the left, respectively right, transition system, are

$$\{\varepsilon, a, b\}, \text{ respectively } \{\varepsilon, c, ca, cb\}.$$

We see that none of the two languages above is forward-closed (E.g., for the second language we have that  $a\|b$  and  $ca, cb$  belong to the language, but  $cab$  does not belong to it.)

The left transition system has multiple initial states, while the second one has only one initial state (but nondeterministic choice for  $c$ ). □

<sup>1</sup>This property bears other different names like: *properness* in [Maz87] (see also [DR95, Chapter 9]), *ff-closure* in [Muk02], *coherence* in [BM03], or *safe-branching* in [SEM03].

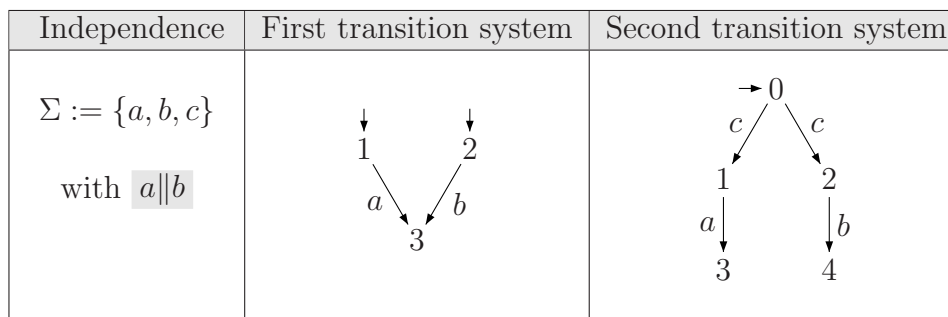


Figure 3.9: Two transition systems satisfying FD whose languages are not forward-closed

Since the distributed transition systems satisfy the ID and FD properties (cf. Proposition 3.22), we further obtain:

**Corollary 3.35** *Let  $(\Sigma, Proc, \Delta)$  be a distribution and  $\parallel$  its corresponding independence relation. Then:*

1.  $\mathcal{L}(DSP)$ ,  $\mathcal{L}(DAA)$ ,  $\mathcal{L}(ADSP)$ , and  $\mathcal{L}(ADAA)$  are included in the class of forward-closed trace-closed languages.
2.  $\mathcal{L}(NSP)$ ,  $\mathcal{L}(NAA)$ ,  $\mathcal{L}(ANSP)$ , and  $\mathcal{L}(ANAA)$  are included in the class of trace-closed languages.

We can investigate also the structural properties of regular languages satisfying the forward and trace closure properties. Interestingly enough, we recover the ID and FD properties, but only for the *minimal* transition system accepting the given language<sup>1</sup>. The following result will later be useful in the distributed implementability test.

**Proposition 3.36** *Let  $(\Sigma, \parallel)$  be a concurrent alphabet and  $L \subseteq \Sigma^*$  a prefix-closed regular language. Then, the following are equivalent:*

1.  $L$  is forward- and trace-closed.
2. The minimal deterministic transition system accepting  $L$  satisfies ID and FD.

**Proof.** First of all, according to Corollary 2.19, for the prefix-closed regular language  $L$ , there exists a unique (up to isomorphism) *minimal* deterministic transition system, say  $TS = (Q, \Sigma, \rightarrow, \{q^{in}\})$ , such that  $L = L(TS)$ .

1  $\Rightarrow$  2 : Let us first prove that  $TS$  satisfies ID. Let  $q_1, q_2, q_3 \in Q$  and  $a \parallel b$  such that  $q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_3$ . Since  $TS$  is reachable, there exists  $w_0 \in \Sigma^*$  such that  $q^{in} \xrightarrow{w_0} q_1$ . So,  $w_0 ab \in L(TS)$ . Since  $L(TS)$  is trace-closed, we have that also  $w_0 ba \in L(TS)$ , which implies that there exists two states  $q_4, q'_4 \in Q$  such that  $q^{in} \xrightarrow{w_0} q_1 \xrightarrow{b} q_4 \xrightarrow{a}$

<sup>1</sup>According to Proposition 3.34, if a deterministic transition system satisfies FD, then its language is forward-closed, but *not* the other way around: E.g., the language  $L := \{\varepsilon, a, b, ab, ba\}$  is forward-closed for  $a \parallel b$ , but the (non-minimal) deterministic transition system  $2 \xleftarrow{b} 1 \xleftarrow{a} 0 \xrightarrow{b} 3 \xrightarrow{a} 4$  (with 0 initial state) accepts  $L$  and does not satisfy FD.

$q'_4$  (after executing  $w_0$  from  $q^{in}$ , we always reach  $q_1$ , because  $TS$  is deterministic). If we prove that  $q_3$  is equal to  $q'_4$ , we can conclude that the ‘diamond’ property ID holds in  $q_1$  for  $a\|b$ .

By contradiction, assume  $q_3 \neq q'_4$ . Recall now the construction of the minimal deterministic transition system from the proof of Corollary 2.19. There, we made the transition relation total by adding a ‘sink state’  $\perp$ . From  $q_3 \neq q'_4$ , we have that  $q_3 \not\approx q'_4$ , which means that there exists an execution  $w \in \Sigma^*$  such that  $q_3 \xrightarrow{w} q'$  and  $q'_4 \xrightarrow{w} q''$  with either  $q' = \perp$  or  $q'' = \perp$ , but not both equal to  $\perp$ . Without loss of generality, suppose  $q' \neq \perp$  and  $q'' = \perp$ . Then, on one hand, we have that  $q^{in} \xrightarrow{w_0} q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_3 \xrightarrow{w} q'$ , so  $w_0abw \in L(TS)$ . On the other hand,  $q^{in} \xrightarrow{w_0} q_1 \xrightarrow{b} q_4 \xrightarrow{a} q'_4 \xrightarrow{w} \perp$ , so  $w_0baw \notin L(TS)$ . Thus, we have  $w_0abw \in L(TS)$ ,  $a\|b$ , and  $w_0baw \notin L(TS)$ , which implies that  $L(TS)$  is not trace-closed, but this contradicts the hypothesis that  $L(TS)$  is trace-closed.

Let us prove now that  $TS$  satisfies FD. Let  $q_1, q_2, q_3 \in Q$  and  $a\|b$  such that  $q_1 \xrightarrow{a} q_2$  and  $q_1 \xrightarrow{b} q_3$ . Since  $TS$  is reachable, there exists  $w_0 \in \Sigma^*$  such that  $q^{in} \xrightarrow{w_0} q_1$ . So,  $w_0a \in L(TS)$  and  $w_0b \in L(TS)$ . Since  $L(TS)$  is forward-closed, we have that also  $w_0ab \in L(TS)$ , which implies that there exists a state  $q_4 \in Q$  such that  $q^{in} \xrightarrow{w_0} q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_4$ . Symmetrically, from  $wb \in L(TS)$ ,  $wa \in L(TS)$ , and  $b\|a$  (the independence relation is symmetric), there exists  $q'_4 \in Q$  such that  $q^{in} \xrightarrow{w_0} q_1 \xrightarrow{b} q_3 \xrightarrow{a} q'_4$ . Finally, FD holds in  $q_1$  for  $a$  and  $b$ , because  $q_4 = q'_4$  for the same reasons as in the ID case above (i.e., we use the minimality of  $TS$ ).

2  $\Rightarrow$  1 : Follows directly from 1. and 2. of Proposition 3.34.  $\square$

**Final states** We redo the analysis of the relation between diamond and closure properties taking a set of *final states* into account. The difference to the above results lies in the forward diamond case. More precisely, the following hold:

**Proposition 3.37** *For any concurrent alphabet  $(\Sigma, \parallel)$  and any finite automaton  $\mathcal{A} = (Q, \Sigma, \rightarrow, I, F)$ , we have:*

1. *If  $\mathcal{A}$  satisfies ID, then  $L(\mathcal{A}, F)$  is trace-closed.*
2. *If  $\mathcal{A}$  satisfies FD, then it is not always the case that  $L(\mathcal{A}, F)$  is forward-closed, even for  $\mathcal{A}$  deterministic.*

**Proof.** 1. Same proof as the one for 1. of Proposition 3.37.

2. For the concurrent alphabet  $\Sigma := \{a, b\}$  with  $a\|b$ , the finite automaton  $\mathcal{A}$  of Figure 3.10 with the set of final states  $F := \{0, 1, 2\}$  is deterministic and satisfies FD (and ID). On the other hand, his language  $\{\varepsilon, a, b\}$  is not forward-closed.  $\square$

Last part of Proposition 3.37 suggests that taking final states into account is not suitable to a distributed setting, since the structural information given by FD is not reflected in the accepted language of the distributed system. Therefore, we can derive only the following corollary from the first part of Proposition 3.37:

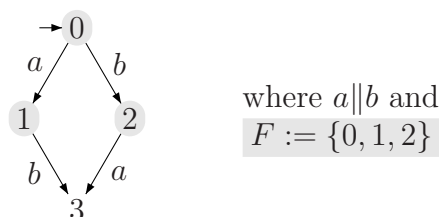


Figure 3.10: Deterministic finite automaton satisfying FD whose language is not forward-closed

**Corollary 3.38** *Let  $(\Sigma, Proc, \Delta)$  be a distribution and  $\parallel$  its corresponding independence relation. Then, all  $\mathcal{L}(DSP, F)$ ,  $\mathcal{L}(DAA, F)$ ,  $\mathcal{L}(NSP, F)$ ,  $\mathcal{L}(NAA, F)$ ,  $\mathcal{L}(ADSP, F)$ ,  $\mathcal{L}(ADAA, F)$ ,  $\mathcal{L}(ANSP, F)$ , and  $\mathcal{L}(ANAA, F)$  are included in the class of trace-closed languages.*

We also have a similar of Proposition 3.36 for ID only.

**Proposition 3.39** *Let  $(\Sigma, \parallel)$  be a concurrent alphabet and  $L \subseteq \Sigma^*$  a regular language. Then, the following are equivalent:*

1.  $L$  is trace-closed.
2. The minimal deterministic finite automaton accepting  $L$  satisfies ID.

**Proof.**  $1 \Rightarrow 2$  : Similar to the proof of  $(1 \Rightarrow 2)$  from Proposition 3.36.

$2 \Rightarrow 1$  : Follows directly from 1. of Proposition 3.37. □

### 3.4.3 Synchronous Products of Transition Systems

In this section, we give characterizations for the class of languages accepted by synchronous products of transition systems. Many of the results are picked up from the literature [Dub86, Zie87, Thi95, CMT99, Muk02].

In Section 3.4.2 we have seen that the structural properties ID and FD induce the trace and forward closure properties on the corresponding languages. Similarly, the synchronization on common actions of a synchronous product will be present in the characterization of the languages of synchronous products. We will see that the languages accepted by synchronized products are synchronizations on common actions of the local languages corresponding to the components. Thus the first step is to define a synchronization on common actions for languages. The result of the synchronization will be called a *product language*<sup>1</sup>:

<sup>1</sup>Maybe a more suitable name would have been ‘synchronous language’, but we stick to the nomenclature used in [Thi95, CMT99, Muk02] in order to avoid confusion with the established name of ‘synchronous language’ used inside the synchronous programming languages community [BCE<sup>+</sup>03].



**Definition 3.40 (Product language)**

Let  $(\Sigma, Proc, \Delta)$  be a distribution together with its local alphabets<sup>1</sup>  $(\Sigma_{loc}(p))_{p \in Proc}$ . We say that a language  $L \subseteq \Sigma^*$  is a *product language* over  $\Delta$ , if for each  $p \in Proc$  there exists a language  $L_p \subseteq (\Sigma_{loc}(p))^*$  such that

$$L = \{w \in \Sigma^* \mid w \upharpoonright_{\Sigma_{loc}(p)} \in L_p \text{ for all } p \in Proc\}.$$

In other words,  $L$  consists of all possible synchronizations on common actions of words of the ‘local’ languages  $L_p$ . (Note that in the particular case where the actions of the local alphabets are pairwise independent, the synchronization on common actions becomes the shuffle operation. At the opposite pole, if the actions of the local alphabets are pairwise dependent, we recover the intersection operation.)

Using the definition, it is not difficult to prove that the product languages are both forward- and trace-closed:

**Proposition 3.41** [Thi95] *Assume  $L \subseteq \Sigma^*$  is a product language over  $(\Sigma, Proc, \Delta)$ . Then,  $L$  is forward- and trace-closed with respect to the independence generated by  $\Delta$ .*

The reverse of the above proposition is not true: Not all forward- and trace-closed languages are necessarily product languages as the following example shows.

**Example 3.42** We choose the language of the transition system of Figure 3.8, i.e.,

$$L := \{\varepsilon, a, c, ac, cb\}$$

over the distribution  $(\{a, b, c\}, \{1, 2\}, \{(a, 1), (b, 2), (c, 1), (c, 2)\})$ . On the one hand,  $L$  is forward- and trace-closed over the independence relation  $\parallel = \{(a, b)\}$ . On the other hand,  $L$  is not a product language: By contradiction, assume  $L$  is a product language. Then, there exist  $L_1 \subseteq (\Sigma_{loc}(1))^* = \{a, c\}^*$  and  $L_2 \subseteq (\Sigma_{loc}(2))^* = \{b, c\}^*$  as in Definition 3.40. From  $ac \in L$ , we have  $ac \in L_1$ . Similarly, from  $cb \in L$ , we have  $cb \in L_2$ . Then, we must also have  $acb \in L$  (because  $acb \upharpoonright_{\{a,c\}} = ac \in L_1$  and  $acb \upharpoonright_{\{b,c\}} = cb \in L_2$ ), but this is a contradiction.

An important characterization of the product languages says that they are exactly those languages that are equal to the synchronization of their own projections on the local alphabets:

**Proposition 3.43** [Thi95, CMT99] *Let  $(\Sigma, Proc, \Delta)$  be a distribution and  $L \subseteq \Sigma^*$  a language. Then,  $L$  is a product language over  $\Delta$  if and only if*

$$L = \{w \in \Sigma^* \mid w \upharpoonright_{\Sigma_{loc}(p)} \in L \upharpoonright_{\Sigma_{loc}(p)} \text{ for all } p \in Proc\}.$$

We relate now the (regular) product languages to the languages of synchronous products of transition systems. Starting with the synchronous products having *only one global initial state* (i.e.,  $I = \{q^{in}\}$ ), one can show that their languages are product languages. The proof relies on Proposition 3.43 and with the observation that the projections  $L \upharpoonright_{\Sigma_{loc}(p)}$  of the language  $L$  of a synchronous product are exactly the languages of the local transition systems  $(Q_p, \Sigma_{loc}(p), \rightarrow_p, \{q_p^{in}\})$ , where the initial state is  $q^{in} = (q_p^{in})_{p \in Proc}$ .

<sup>1</sup>Recall Definition 3.13.



**Proposition 3.44** [Dub86, Thi95] *The language of a synchronous product with only one global initial state (i.e.,  $|I| = 1$ ) over a distribution  $\Delta$  is a product language (over  $\Delta$ ).*

**Corollary 3.45** *The language of a deterministic synchronous product (over a distribution  $\Delta$ ) is a product language (over  $\Delta$ ).*

**Proof.** A deterministic synchronous product has by definition only one initial state.

In the general case, when there may exist multiple initial states, the language of a synchronous product of transition systems is a finite union of product languages (each initial state will generate a product language by Proposition 3.44, so the language of the synchronous product is the union over the initial states of these product languages):

**Proposition 3.46** [Dub86, Zie87, Thi95] *The language of a synchronous product over a distribution  $\Delta$  is a finite union of product languages (over  $\Delta$ ).*

Note that the class of product languages over a (non-trivial) distribution may *not* be closed under finite union as the following trivial example shows:

**Example 3.47** For a distribution containing a pair of independent actions  $a||b$ , the languages  $L_1 := \{a\}$  and  $L_2 := \{b\}$  are obviously product languages, but their union  $L_1 \cup L_2 = \{a, b\}$  is not (because  $\{a, b\}$  is not forward-closed, whereas the forward-closure is a property of product languages according to Proposition 3.41).

Moreover, since the language of a synchronous product is the language of its global transition system (Definition 3.17) and the language of a transition system is prefix-closed and regular (Corollary 2.15), we have:

**Lemma 3.48** *The language of a synchronous product is a prefix-closed regular language.*

Until now we have seen that the class of languages of synchronous products is included in the intersection of the class of (finite unions of) product languages (Propositions 3.44,3.46) and of prefix-closed regular languages (Lemma 3.48). The next proposition says that the reverse inclusion also holds:

**Proposition 3.49** [Thi95, CMT99] *Let  $(\Sigma, Proc, \Delta)$  be a distribution and  $L \in \Sigma^*$  a language. Then:*

1. *If  $L$  is a prefix-closed regular product language over  $\Delta$ , then there exists a deterministic synchronous product  $\mathcal{SP}$  over  $\Delta$  such that  $L = L(\mathcal{SP})$ .*
2. *If  $L$  is a finite union of prefix-closed regular product languages over  $\Delta$ , then there exists a synchronous product  $\mathcal{SP}$  over  $\Delta$  such that  $L = L(\mathcal{SP})$ .*

**Proof.** 1. Let  $L$  be a prefix-closed regular product language over  $\Delta$ . According to Proposition 3.43,  $L$  is the synchronization on common actions of its projections  $L \upharpoonright_{\Sigma_{loc}(p)}$ . Since the class of prefix-closed regular languages is closed under projection (Corollary 2.6), all  $L \upharpoonright_{\Sigma_{loc}(p)}$  are prefix-closed regular languages. Then, according

to Corollaries 2.15 and 2.17, there exists a deterministic local transition system  $TS_p = (Q_p, \Sigma_{loc}(p), \rightarrow_p, \{q_p^{in}\})$  for each  $p \in Proc$  such that  $L(TS_p) = L \upharpoonright_{\Sigma_{loc}(p)}$ .

We consider now  $\mathcal{SP}$  to be the synchronous product (over  $\Delta$ ) of the transition systems  $(TS_p)_{p \in Proc}$ . Since all  $TS_p$ s are deterministic, using Proposition 3.24,  $\mathcal{SP}$  is also deterministic. Using the definition of synchronization on common actions of product languages, respectively of local transition systems, it is not hard to see that indeed  $L(\mathcal{SP}) = L$ .

2. Let  $L = \bigcup_{j \in J} L_j$  be a finite union of prefix-closed regular product languages  $(L_j)_{j \in J}$  over  $\Delta$ . According to the previous step, for each  $L_j$  there exists a deterministic synchronous product  $\mathcal{SP}_j$  over  $\Delta$  such that  $L(\mathcal{SP}_j) = L_j$ . We can construct now a synchronous product  $\mathcal{SP}$  to accept the union of  $\bigcup_{j \in J} L_j = L$  from the  $(\mathcal{SP}_j)_{j \in J}$  in the following way: For each process  $p \in Proc$ , the  $p$ -local transition system of  $\mathcal{SP}$  is constructed as the *disjoint* union of the  $p$ -local transition systems of the  $\mathcal{SP}_j$ s and the set of global initial states of  $\mathcal{SP}$  consists of the initial states of the  $\mathcal{SP}_j$ s.

It is not difficult to see that indeed  $L(\mathcal{SP}) = \bigcup_{j \in J} L(\mathcal{SP}_j)$ .

The constructions sketched above play an important rôle in the synthesis of synchronous products and we will revisit them at a later point.  $\square$

From all the above results we can characterize the class of synchronous products by means of product languages in the following way:

**Theorem 3.50** *For a fixed distribution  $(\Sigma, Proc, \Delta)$  we have:*

1.  $\mathcal{L}(DSP)$  is equal to the class of prefix-closed regular product languages.
2.  $\mathcal{L}(NSP)$  is equal to the class of finite unions of prefix-closed regular product languages.

The above result can be adapted further to *acyclic* systems, by restricting the characteristic ‘regular’ to ‘finite’ (the acyclic transition systems accept finite languages, i.e., languages with a finite number of words – cf. Theorem 2.20).

**Corollary 3.51** *For a fixed distribution  $(\Sigma, Proc, \Delta)$  we have:*

1.  $\mathcal{L}(ADSP)$  is equal to the class of prefix-closed finite product languages.
2.  $\mathcal{L}(ANSP)$  is equal to the class of finite unions of prefix-closed finite product languages.<sup>1</sup>

The above characterizations will prove helpful in the study of the computational complexity of the synthesis problem for synchronous products.

As a last remark, from Example 3.47 and Theorem 3.50, we see that the nondeterministic synchronous products are more expressive than their deterministic counterparts. Nevertheless, this is due only to the *multiple global initial states*: Once we force the (nondeterministic) synchronous products to have the set of global initial states as a *cartesian product of local initial states*, we obtain the same expressiveness power as for the deterministic ones, as the following proposition shows:

<sup>1</sup>Note that according to Example 3.47 the class of prefix-closed finite product languages may not be closed under finite union, so  $\mathcal{L}(ADSP) \subsetneq \mathcal{L}(ANSP)$ .

**Proposition 3.52** *Let  $\mathcal{SP} = (Q, \Sigma, \rightarrow, I)$  be a nondeterministic synchronous product over a distribution  $(\Sigma, Proc, \Delta)$  such that*

$$I = \prod_{p \in Proc} I_p$$

*with  $I_p \subseteq Q_p$  (the  $Q_p$  are the local state spaces of Definition 3.15). Then, there exists a deterministic synchronous product  $\mathcal{SP}'$  over the same  $\Delta$  such that  $L(\mathcal{SP}) = L(\mathcal{SP}')$ .*

**Proof.** The deterministic synchronous product  $\mathcal{SP}'$  is obtained by *determinizing* the local components of  $\mathcal{SP}$  taking as local sets of initial states the local sets  $I_p$  (cf. Corollary 2.17). The synchronous product of the determinized components preserves the original language and is deterministic according to Proposition 3.24.  $\square$

**Corollary 3.53** *For any nondeterministic synchronous product  $\mathcal{SP} = (Q, \Sigma, \rightarrow, I)$  over a distribution  $\Delta$  with only one initial state (i.e.,  $|I| = 1$ ), there exists a deterministic synchronous product  $\mathcal{SP}'$  over the same  $\Delta$  such that  $L(\mathcal{SP}) = L(\mathcal{SP}')$ .*

**Final states** Once we take final states into account, the accepted languages are in general not prefix-closed anymore. The first thought might be that characterization results similar to Theorem 3.50 (and Corollary 3.51) for final states can be obtained simply by removing the ‘prefix-closure’ constraint. As we will see in the following, this is only part of the story. If we want a result similar to Theorem 3.50, we need a set of final states of a special form, and namely, we must require that the set of global final states is a cartesian product of *local* final states.

Let us start the study with the class of regular product languages and let  $L$  be a regular product language over a distribution  $\Delta$ . Then, according to Proposition 3.43,  $L$  is the synchronization of its projections  $L \upharpoonright_{\Sigma_{loc}(p)}$ , while according to Proposition 2.5, these projections are regular (because  $L$  is regular). Next, for each process  $p \in Proc$ , we can construct a local deterministic finite automaton  $\mathcal{A}_p = (Q_p, \Sigma_{loc}(p), \rightarrow_p, \{q_p^{in}\}, F_p)$  accepting the  $p$ -local projection, i.e.,  $L(\mathcal{A}_p, F_p) = L \upharpoonright_{\Sigma_{loc}(p)}$ . Then, the synchronous product, denoted by  $\mathcal{SP}$ , of the local transitions given by the  $\mathcal{A}_p$ s with the unique global initial state  $q^{in} = (q_p^{in})_{p \in Proc}$  and the set of global final states

$$F := \prod_{p \in Proc} F_p,$$

accepts the original language, i.e.,  $L(\mathcal{SP}, F) = L$ . Moreover,  $\mathcal{SP}$  is deterministic (because the  $\mathcal{A}_p$ s are deterministic). Thus, the class of regular product languages is included in the class of the languages accepted by deterministic synchronous products with local sets of final states. It is easy to see that the reverse inclusion also holds. To ease the presentation, we will introduce a new notation for the above-emerging class of languages.

**Notation.** We denote by

$$\begin{aligned} &\mathcal{L}(DSP, locF), \text{ respectively} \\ &\mathcal{L}(NSP, locF), \end{aligned}$$

the class of languages of deterministic, respectively nondeterministic, synchronous products with a set of global final states given by a *cartesian product of sets of local final states*.

Now we are ready to give a theorem similar to Theorem 3.50 for local final states.

**Theorem 3.54** *For a fixed distribution  $(\Sigma, Proc, \Delta)$  we have that:*

1.  $\mathcal{L}(DSP, locF)$  is equal to the class of regular product languages.
2.  $\mathcal{L}(NSP, locF)$  is equal to the class of finite unions of regular product languages.

**Proof.** 1. The proof was already given above.

2. For the direct inclusion, let  $\mathcal{SP}$  be a nondeterministic synchronous product together with a local sets of final states  $(F_p)_{p \in Proc}$  and  $F := \prod_{p \in Proc} F_p$ . Then the accepted language  $L(\mathcal{SP}, F)$  is the finite union of the languages generated by each initial state of  $\mathcal{SP}$ . Using an argument similar to the proof of Proposition 3.52, each of the languages generated by one initial state is a regular product language. Therefore,  $L(\mathcal{SP}, F)$  is a finite union of regular product languages.

For the inverse inclusion, let  $L$  be a finite union of regular product languages. Then, there exists a family of, say  $k$ , product languages  $(L_i)_{i \in [1..k]}$  such that  $L = \bigcup_{i=1}^k L_i$ . According to the first part of this theorem, for each regular product language  $L_i$  (with  $i \in [1..k]$ ) there exists a deterministic synchronous product  $\mathcal{SP}_i$  over  $\Delta$  with local final states accepting  $L_i$ .

Suppose for convenience (and without loss of generality) that  $Proc = \{1, \dots, m\}$  (i.e.,  $|Proc| = m$ ). For each  $i \in [1..k]$ ,  $\mathcal{SP}_i$  is the synchronous product of a set of local automata  $(\mathcal{A}_{ip})_{p \in [1..m]}$  where  $\mathcal{A}_{ip} := (Q_{ip}, \Sigma_{loc}(p), \rightarrow_{ip}, \{q_{ip}^{in}\}, F_{ip})$ . Then, by hypothesis,  $L_i = L(\mathcal{SP}_i, \prod_{p=1}^m F_{ip})$  where the unique global initial state of  $\mathcal{SP}_i$  is  $q_i^{in} = (q_{i1}^{in}, \dots, q_{im}^{in})$ .

We construct now a (nondeterministic) synchronous product  $\mathcal{SP}$  over  $\Delta$  with local final states accepting the union  $\bigcup_{i=1}^k L_i$  in the following natural way (see Figure 3.11):

- the local components consist of the *disjoint* union of the local automata of the  $\mathcal{SP}_i$ s, i.e.,

$$\mathcal{A}_p := \bigsqcup_{i=1}^k \mathcal{A}_{ip},$$

for each  $p \in [1..m]$ .

- the set of initial states of  $\mathcal{SP}$  is the union of the initial states of the  $\mathcal{SP}_i$ s, i.e.,

$$I := \{q_i^{in} \mid i \in [1..k]\}.$$

- the set of the global final states of  $\mathcal{SP}$  is the cartesian product of the *disjoint* union of the local final states of the  $\mathcal{SP}_i$ s, i.e.,

$$F := \prod_{p=1}^m \left( \bigsqcup_{i=1}^k F_{ip} \right).$$

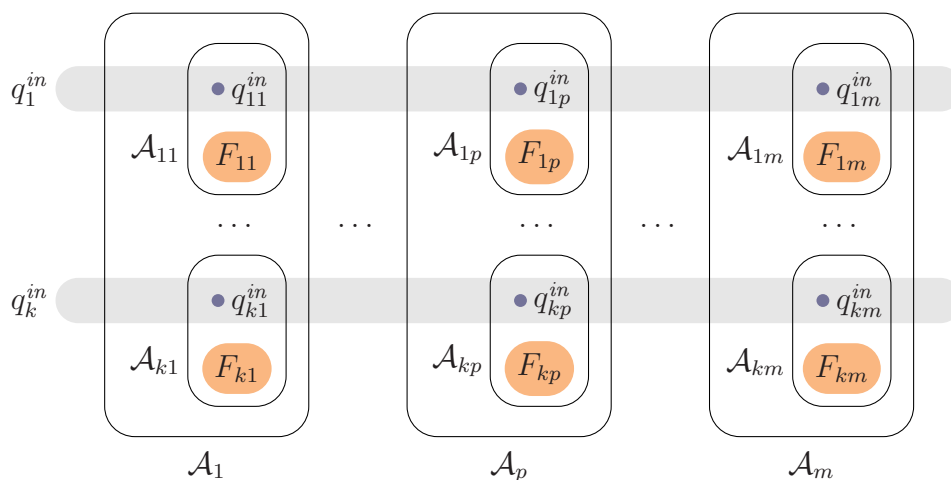


Figure 3.11: Nondeterministic synchronous product with local final states accepting a finite union of regular product languages

We prove now that for the  $\mathcal{SP}$  constructed above indeed holds that

$$L = L(\mathcal{SP}, F).$$

For the direct inclusion, if  $w \in L = \bigcup_{i=1}^k L_i$ , there exists  $i \in [1..k]$  such that  $w \in L_i$ , which means that  $w \in L(\mathcal{SP}_i, \prod_{p=1}^m F_{ip})$ , so there exists a global final state  $q_i^{fin} \in \prod_{p=1}^m F_{ip}$  such that  $q_i^{in} \xrightarrow{w} q_i^{fin}$ . Looking at Figure 3.11, it is not difficult to see that  $q_i^{in} \xrightarrow{w} q_i^{fin}$  is also possible in  $\mathcal{SP}$  and  $q_i^{fin} \in F$ , so  $w \in L(\mathcal{SP}, F)$ .

For the reverse inclusion, let  $w \in L(\mathcal{SP}, F)$ . Then, there exists  $q^{in} \in I$  and  $q^{fin} \in F$  such that  $q^{in} \xrightarrow{w} q^{fin}$ . Since  $I = \{q_i^{in} \mid i \in [1..k]\}$ , there exists  $i \in [1..k]$  such that  $q^{in} = q_i^{in}$ . According to the definition of a synchronous product (Definition 3.15), the global state space consists only of states reachable from the initial ones. Moreover, the local transition systems  $(\mathcal{A}_p)_{p \in [1..m]}$  are constructed as *disjoint* unions of  $\mathcal{A}_{ip}$ s. From the last two facts, we deduce that from  $q^{in}$  by  $w$  we cannot ‘touch’ any of the local states of  $\mathcal{A}_{jp}$ , with  $j \neq i$ . This implies that the reachable global state  $q^{fin}$  is necessarily a reachable global state of  $\mathcal{SP}_i$ . Finally, since  $q^{fin} \in F$ , we have that  $q^{fin} \in \prod_{p=1}^m F_{ip}$ , which implies  $w \in L(\mathcal{SP}_i, \prod_{p=1}^m F_{ip}) = L_i \subseteq L$ .  $\square$

After the smooth characterization of Theorem 3.54 for languages of synchronous product with *local* final states, we can further consider *unrestricted* sets of global final states. The class of accepted languages in this case will not become richer than in the case of local final states for nondeterministic systems. However, there will be a difference in the deterministic case:

**Corollary 3.55** *In general, we have that*

$$\mathcal{L}(DSP, locF) \subsetneq \mathcal{L}(DSP, F) \subseteq \mathcal{L}(NSP, F) = \mathcal{L}(NSP, locF).$$

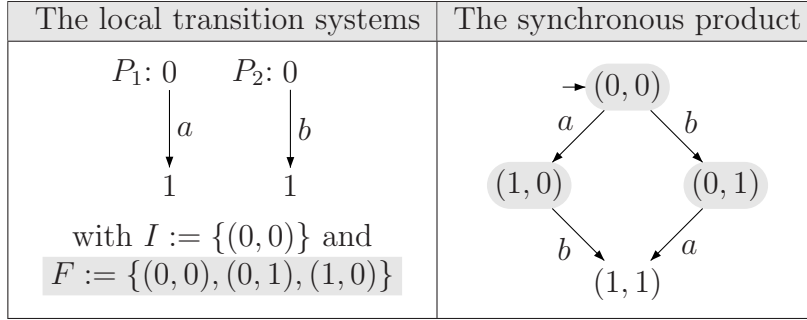


Figure 3.12: Deterministic synchronous product with final states accepting  $\{\varepsilon, a, b\}$  for  $a \parallel b$

**Proof.** First, the following inclusions are obvious:

$$\begin{aligned}
 \mathcal{L}(DSP, locF) &\subseteq \mathcal{L}(DSP, F), & \mathcal{L}(NSP, locF) &\subseteq \mathcal{L}(NSP, F), \\
 \mathcal{L}(DSP, locF) &\subseteq \mathcal{L}(NSP, locF), & \mathcal{L}(DSP, F) &\subseteq \mathcal{L}(NSP, F).
 \end{aligned}$$

Then, the following are enough to complete the proof:

$\mathcal{L}(NSP, F) \subseteq \mathcal{L}(NSP, locF)$  : Similar to Proposition 3.46, the class  $\mathcal{L}(NSP, F)$  of languages of synchronous products with global final states is included in the class of finite unions of regular product languages [Dub86, Zie87]. Since the latter class is equal to  $\mathcal{L}(NSP, locF)$ , we have the inclusion  $\mathcal{L}(NSP, F) \subseteq \mathcal{L}(NSP, locF)$ .

$\mathcal{L}(DSP, F) \subseteq \mathcal{L}(NSP, locF)$  : Because  $\mathcal{L}(DSP, F) \subseteq \mathcal{L}(NSP, F) = \mathcal{L}(NSP, locF)$  (see previous inclusions).

$\mathcal{L}(DSP, F) \setminus \mathcal{L}(DSP, locF) \neq \emptyset$  : Recall Example 3.47. Let  $L := \{\varepsilon, a, b\}$  and a distribution of  $\Sigma := \{a, b\}$  over two processes  $Proc := \{1, 2\}$  with  $\Delta = \{(a, 1), (b, 2)\}$  (so,  $a \parallel b$ ). On one hand, the deterministic synchronous product with final states from Figure 3.12 accepts  $L$ , so  $L \in \mathcal{L}(DSP, F)$ . On the other hand,  $L$  is not a product language (because  $L$  is not forward-closed – cf. Proposition 3.41). From the first part of Theorem 3.54, the languages of  $\mathcal{L}(DSP, locF)$  are product languages, so  $L \notin \mathcal{L}(DSP, locF)$ .  $\square$

In the last corollary, we have seen that  $\mathcal{L}(DSP, F) \subseteq \mathcal{L}(NSP, F)$ . Nevertheless, it is not clear yet whether  $\mathcal{L}(DSP, F)$  is *properly* included in  $\mathcal{L}(NSP, F)$  or not.

For the acyclic case, similar to the Corollary 3.51, we have:

**Corollary 3.56** *For a fixed distribution  $(\Sigma, Proc, \Delta)$  we have:*

1.  $\mathcal{L}(ADSP, locF)$  is equal to the class of finite product languages.
2.  $\mathcal{L}(ANSP, locF)$  is equal to the class of finite unions of finite product languages.<sup>1</sup>

<sup>1</sup>Note that according to Example 3.47 the class of finite product languages may not be closed under finite union, so  $\mathcal{L}(ADSP, locF) \subsetneq \mathcal{L}(ANSP, locF)$ .

Moreover, taking global final states into account we have:

**Corollary 3.57** *In general, we have that*

$$\mathcal{L}(ADSP, locF) \subsetneq \mathcal{L}(ADSP, F) = \mathcal{L}(ANSP, F) = \mathcal{L}(ANSP, locF).$$

**Proof.** See last section of [Zie87].

Furthermore, the unions of finite product languages admit the following simpler characterization:

**Proposition 3.58** [Zie87]  $\mathcal{L}(ANSP, F)$  is equal to the class of trace-closed finite languages.<sup>1</sup>

Thus, from Corollary 3.57 and Proposition 3.58 we get:

**Corollary 3.59**  $\mathcal{L}(ADSP, F)$ ,  $\mathcal{L}(ANSP, F)$ ,  $\mathcal{L}(ANSP, locF)$  are all equal to the class of trace-closed finite languages.

We end with the remark that if we want to consider final states for synchronous products, the right choice is to have *local* final states rather than global ones (the intuition lies in the loose synchronization on common actions implemented by the synchronous products). As we will see in the next section, this is not necessarily the case for asynchronous automata (for which there is more information exchanged between the processes during a synchronization).

### 3.4.4 Asynchronous Automata

In this section, we give characterizations for the class of languages accepted by asynchronous automata which are based on a fundamental result by Zielonka [Zie87]. This deep result says that the class of languages of (deterministic) asynchronous automata with *global final states* is equal to the class of regular trace-closed languages.

**Theorem 3.60** [Zie87] *For a fixed distribution  $(\Sigma, Proc, \Delta)$  and a language  $T \subseteq \Sigma^*$ , the following are equivalent:*

1.  $T$  is a regular trace-closed language.
2. There exists a (deterministic) asynchronous automaton  $\mathcal{AA}$  with a set of global final states  $F$  such that  $L(\mathcal{AA}, F) = T$ .

The difficult direction is from 1 to 2 (the other direction simply follows from Corollary 3.38). The construction is indeed involved and produces very large asynchronous automata (we will see ways to find smaller solutions in Chapter 5).

Motivated by problems regarding the global final states similar to those mentioned in Section 3.4.1, Zielonka devoted a subsequent paper [Zie89] to obtain the same result (i.e., Theorem 3.60) for the restricted class of so-called *safe* asynchronous automata, which are asynchronous automata with the additional property that any run from an initial state can be extended to an accepted run.

<sup>1</sup>It is easy to see that a similar result holds when final states are not taken into account, and namely,  $\mathcal{L}(ANSP)$  (i.e., the class of finite unions of prefix-closed finite product languages) is equal to the class of prefix-closed trace-closed finite languages.



**Theorem 3.61** [Zie89] *For a fixed distribution  $(\Sigma, Proc, \Delta)$  and a language  $T \subseteq \Sigma^*$ , the following are equivalent:*

1.  $T$  is a regular trace-closed language.
2. There exists a safe asynchronous automaton  $\mathcal{AA}$  with a set of global final states  $F$  such that  $L(\mathcal{AA}, F) = T$ .

Nevertheless, Theorem 3.61 is not true in general for *deterministic* safe asynchronous automata (Theorem 3.60 holds also for deterministic asynchronous automata): Recall for instance the language of Example 3.47. Then, there exists a deterministic asynchronous automaton with final states accepting the language (see Figure 3.12), but it is not difficult to see that there is no *safe deterministic* asynchronous automaton accepting it.

Faithful to the conclusions of Section 3.4.1, we are interested in similar results, but without considering a set of global final states. Using the Theorems 3.60 and 3.61, we are able to obtain the following characterizations:

**Theorem 3.62** *For a fixed distribution  $(\Sigma, Proc, \Delta)$  we have that:*

1.  $\mathcal{L}(DAA)$  is equal to the class of prefix-closed regular forward-closed trace-closed languages.
2.  $\mathcal{L}(NAA)$  is equal to the class of prefix-closed regular trace-closed languages.

**Proof.** 1. By construction (Definition 3.20), the language of an asynchronous automaton is a prefix-closed regular language. Moreover, using Corollary 3.35, the language of deterministic asynchronous automaton is a forward-closed trace-closed language. Hence,  $\mathcal{L}(DAA)$  is included in the class of prefix-closed regular forward-closed trace-closed languages. For the reverse inclusion, the proof builds upon the details of Zielonka's construction (Theorem 3.60). Since it would be distracting at this point to introduce the ingredients of Theorem 3.60, we shift the solution to Chapter 5 (Section 5.2.2).

2. The inclusion of  $\mathcal{L}(NAA)$  into the class of prefix-closed regular trace-closed languages is immediate.

For the reverse inclusion, let  $T$  be a prefix-closed regular trace-closed language. According to Theorem 3.61, there exists a safe asynchronous automaton  $\mathcal{AA}$  with a set of global final states  $F$  such that  $L(\mathcal{AA}, F) = T$ .

We show that  $L(\mathcal{AA}, F) = L(\mathcal{AA})$ , where  $L(\mathcal{AA})$  is our definition of language containing all the runs starting in an initial state of  $\mathcal{AA}$ . Since the inclusion  $L(\mathcal{AA}, F) \subseteq L(\mathcal{AA})$  is obvious, we only prove  $L(\mathcal{AA}) \subseteq L(\mathcal{AA}, F)$  using the hypothesis that  $T = L(\mathcal{AA}, F)$  is prefix-closed.

Let  $w \in L(\mathcal{AA})$ . Then, there exists the run  $q^{in} \xrightarrow{w} q$  with  $q^{in} \in I$  in  $\mathcal{AA}$ . By construction,  $\mathcal{AA}$  is a *safe* asynchronous automaton, so any run of  $\mathcal{AA}$  from an initial state can be extended to an accepting run. In particular,  $w$  can be extended to a run  $q^{in} \xrightarrow{w'} q^{fin}$  with  $q^{fin} \in F$ . Hence  $w$  is a prefix of  $w' \in L(\mathcal{AA}, F)$  and since  $L(\mathcal{AA}, F)$  is prefix-closed, we conclude that  $w \in L(\mathcal{AA}, F)$ .  $\square$



**Remark 3.63** At this point we mention a small mistake in the literature. Theorem 3.62 is presented in [Muk02, Theorem 8] and [SEM03, Theorem 2] as  $\mathcal{L}(NAA)$  being equal to  $\mathcal{L}(DAA)$ , and further equal to the class of prefix-closed regular forward-closed trace-closed languages. In reality,  $\mathcal{L}(NAA)$  properly includes  $\mathcal{L}(DAA)$  as the following example shows.

Take, for instance, the transition systems of Figure 3.9. For each of them there exists an isomorphic *nondeterministic* asynchronous automaton, this meaning that their languages belong to  $\mathcal{L}(NAA)$ . On the other hand, since the languages are not forward-closed, they cannot belong to  $\mathcal{L}(DAA)$ .

Adapting Theorem 3.62 to the *acyclic* case, we have

**Corollary 3.64** For a fixed distribution  $(\Sigma, Proc, \Delta)$  we have that:

1.  $\mathcal{L}(ADAA)$  is equal to the class of prefix-closed forward-closed trace-closed finite languages.
2.  $\mathcal{L}(ANAA)$  is equal to the class of prefix-closed trace-closed finite languages.

**Final states** Taking the global final states into account we have the following result as a direct consequence of Theorem 3.60:

**Corollary 3.65** For a fixed distribution  $(\Sigma, Proc, \Delta)$  we have that:

- $\mathcal{L}(DAA, F)$  and  $\mathcal{L}(NAA, F)$  are both equal to the class of regular trace-closed languages<sup>1</sup>.
- $\mathcal{L}(ADAA, F)$  and  $\mathcal{L}(ANAA, F)$  are both equal to the class of trace-closed finite languages.

At the moment, there are no characterizations known for asynchronous automata with *local final states*. This suggests that, in contrast to the model of synchronous products, local sets of final states are not suitable for asynchronous automata.

### 3.4.5 Comparative Expressiveness

In this section, we collect the results from the previous subsections and give an overview on the expressiveness power of various models. More precisely, we show the relation between the behaviors of our distributed transition systems with or without final states, imposing determinism and/or acyclicity or not. More precisely, the Figures 3.13, 3.14, 3.15, and 3.16 give the *inclusion* relations between the aforementioned classes of languages (represented by arrows, i.e., if  $\mathcal{L}_1 \rightarrow \mathcal{L}_2$  in the diagram, then  $\mathcal{L}_1 \subseteq \mathcal{L}_2$ ). For non-trivial distributions (i.e., distributions involving concurrency), some of the inclusions are in fact *proper* inclusions<sup>2</sup>, i.e.,  $\mathcal{L}_1 \subsetneq \mathcal{L}_2$ , and the labels on the arrows give an example of a language belonging to  $\mathcal{L}_2 \subseteq \mathcal{L}_1$ . The correctness of the diagrams is discussed next.

<sup>1</sup>Direct determinization constructions for asynchronous automata are provided in the papers [KMS94, Mus94].

<sup>2</sup>For trivial distributions involving for instance only one process (i.e.,  $|Proc| = 1$ ), the synchronous products and asynchronous automata both reduce to the class of classical transition systems, so nothing interesting from concurrency point of view.

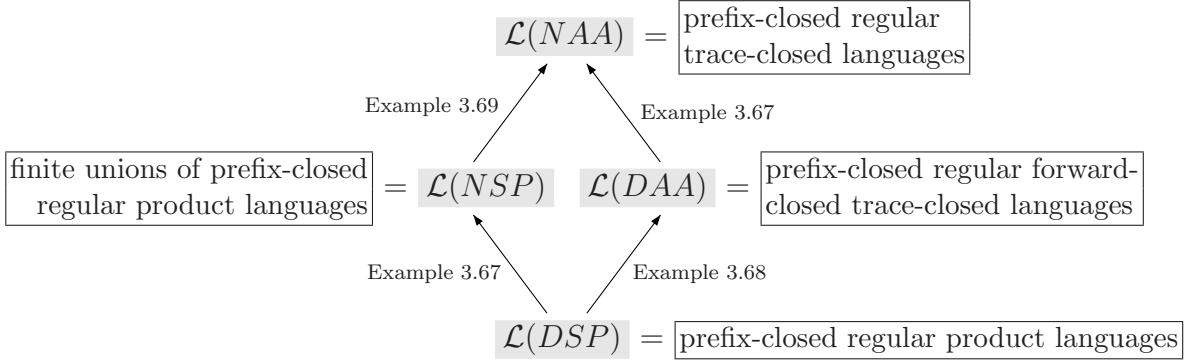


Figure 3.13: Comparison between the classes of languages of distributed transition systems

**Proposition 3.66** *For each pair of nodes  $\mathcal{L}_1, \mathcal{L}_2$  in the graphs of Figures 3.13, 3.14, 3.15, and 3.16, if  $\mathcal{L}_2$  is reachable from  $\mathcal{L}_1$ , then  $\mathcal{L}_1 \subseteq \mathcal{L}_2$ .*

*Moreover, the equality signs in the graphs correspond to equality of classes.*

**Proof.** First, it is enough to show that for each arrow  $\mathcal{L}_1 \rightarrow \mathcal{L}_2$ , we have  $\mathcal{L}_1 \subseteq \mathcal{L}_2$ . This is easy, since: (1) the deterministic models are particular cases of nondeterministic ones, (2) the synchronous products particular cases of asynchronous automata (Remark 3.19), and (3) cartesian products of sets of local final states are particular cases of sets of global final states.

The correctness of equalities is given by the results in Sections 3.4.3 and 3.4.4.  $\square$

Note that there exist also obvious inclusion relations between corresponding class across the given figures: The models with final states are more expressive than the ones without (the languages in the latter category are always prefix-closed); e.g.,  $\mathcal{L}(NSP) \subsetneq \mathcal{L}(NSP, F)$  with the non-prefix-closed language  $\{a\}$  distinguishing them. Also, the acyclic models are less expressive than the corresponding general ones (the languages in the former category are always finite); e.g.,  $\mathcal{L}(ANSP) \subsetneq \mathcal{L}(NSP)$  with the infinite language  $\{a\}^*$  distinguishing them.

Next, we give examples that prove the proper inclusions for non-trivial distributions. Let us choose a generic distribution  $(\Sigma, Proc, \Delta)$  such that  $\{a, b, c\} \subseteq \Sigma$  with  $a \parallel b$ , but  $a \not\parallel c \parallel b$ .

**Example 3.67** We choose the following prefix-closed language:

$$L := \{\varepsilon, a, b\}.$$

On one hand, since  $L$  is a prefix-closed trace-closed finite language, we have

$$L \in \mathcal{L}(ANAA) = \mathcal{L}(ANSP) \subsetneq \mathcal{L}(NSP) \subseteq \mathcal{L}(NAA)$$

(for the equality  $\mathcal{L}(ANAA) = \mathcal{L}(ANSP)$  see the footnote of Proposition 3.58). On the other hand, since  $L$  is not forward-closed,  $L \notin \mathcal{L}(DAA)$  (and thus  $L \notin \mathcal{L}(ADAA)$  and  $L \notin \mathcal{L}(DSP)$  – the two classes are included in  $\mathcal{L}(DAA)$ ). Thus, for  $a \parallel b$ , the arrows labeled by ‘Example 3.67’ in Figures 3.13 and 3.14 depict indeed *proper* inclusions.

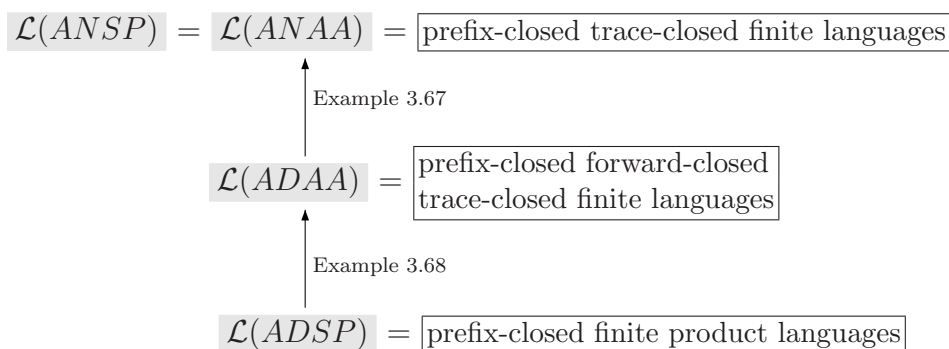


Figure 3.14: Comparison between the classes of languages of *acyclic* distributed transition systems

Taking final states into account, on one hand, we have  $L \in \mathcal{L}(DSP, F)$  (see Figure 3.12) and also  $L \in \mathcal{L}(ANAA, F)$  ( $L$  is a trace-closed finite language). On the other hand,  $L \notin \mathcal{L}(DSP, locF)$  and  $L \notin \mathcal{L}(ADSP, locF)$  (because  $L$  is not forward-closed, and consequently, not a product language – cf. Proposition 3.41). Thus, for  $a\|b$ , the arrows labeled by Example 3.67 in Figures 3.15 and 3.16 depict indeed *proper* inclusions.

**Example 3.68** We choose the following prefix-closed language:

$$L = \{\varepsilon, a, c, ac, cb\}$$

(see also Figure 3.8). On one hand, since  $L$  is a prefix-closed finite forward-closed trace-closed language,  $L \in \mathcal{L}(ADAA) \subsetneq \mathcal{L}(DAA)$ . On the other hand, since  $L$  is not a product language (arguments similar to those given in Example 3.42),  $L \notin \mathcal{L}(ADSP)$  and  $L \notin \mathcal{L}(DSP)$ . Thus, for  $c\|a\|b\|c$ , the arrows labeled by ‘Example 3.68’ in Figures 3.13 and 3.14 depict indeed *proper* inclusions.

**Example 3.69** [Zie87] We choose the following prefix-closed language:

$$L := \text{Prefix}(\left(\left(\left[ab\right] + \left[abb\right]\right)c\right)^*),$$

where  $[u]$  denotes the trace of the word  $u$  (see Section 3.1 for the definition). Figure 3.17 gives a transition system accepting  $L$ .

In [Zie87] it is proved that  $L \in \mathcal{L}(NAA) \subsetneq \mathcal{L}(NAA, F)$  and that  $L$  is *not* a finite union of (prefix-closed) regular product languages, so  $L \notin \mathcal{L}(NSP, F)$  and  $L \notin \mathcal{L}(NSP)$ . Thus, for  $c\|a\|b\|c$ , the arrows labeled by ‘Example 3.69’ in Figures 3.13 and 3.15 depict indeed *proper* inclusions.

Obviously, we have  $\mathcal{L}(DSP, F) \subseteq \mathcal{L}(NSP, F)$ . Nevertheless, since characterizations for the class  $\mathcal{L}(DSP, F)$  are missing, it seems difficult to decide whether the inclusion  $\mathcal{L}(DSP, F) \subseteq \mathcal{L}(NSP, F)$  is *proper* or not. We depict this open question by a dotted line in Figure 3.15.

As a final remark, we can conclude that the synchronous products are in general less expressive than the asynchronous automata.

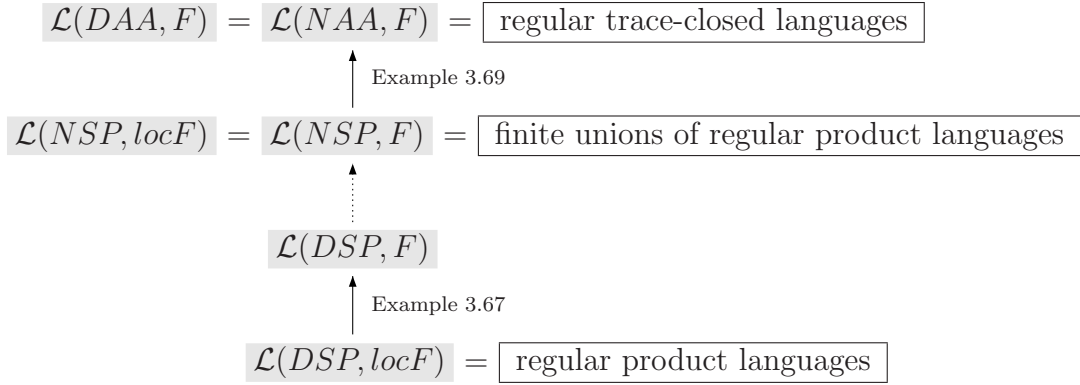


Figure 3.15: Comparison between the classes of languages of distributed transition systems with *final states*

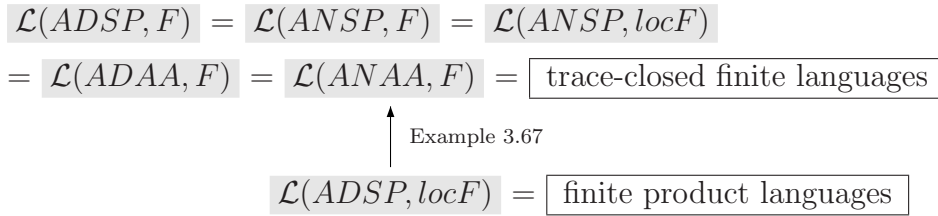


Figure 3.16: Comparison between the classes of languages of acyclic distributed transition systems with *final states*

### 3.5 The Synthesis Problem

Now we are able to state the main problem studied in this thesis, i.e., the synthesis of distributed transition systems from global specifications:

**Problem 3.70 (Synthesis of distributed transition systems)**

<p>INPUT: <i>Given a distribution <math>(\Sigma, Proc, \Delta)</math> and a transition system <math>TS</math> over <math>\Sigma</math>,</i></p> <p>OUTPUT: <i>construct, if possible, a distributed transition system over <math>\Delta</math> whose global state space is equivalent to <math>TS</math>.</i></p>
---

As ‘distributed transition systems’ we consider the two introduced models:

- *synchronous products of transition systems, respectively*
- *asynchronous automata,*

whereas, as ‘equivalence’ we consider respectively:

- *isomorphism,*

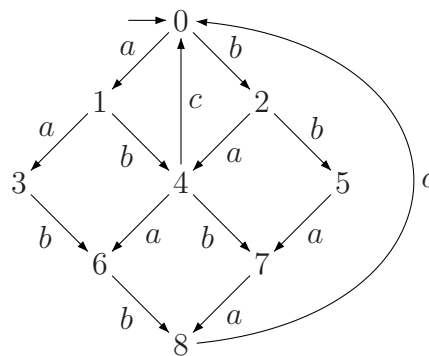


Figure 3.17: Transition system for the language of Example 3.69 [Zie87]

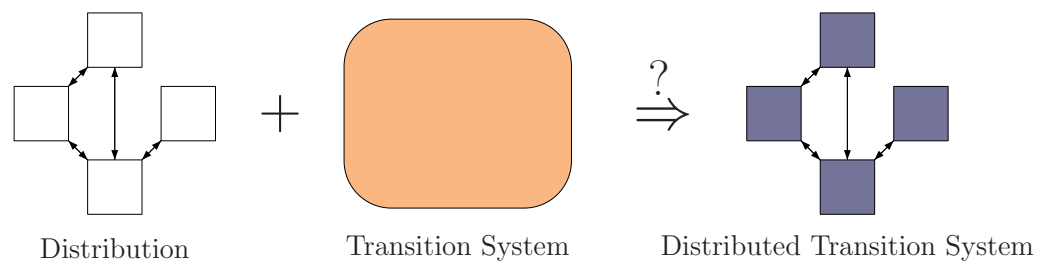


Figure 3.18: The synthesis problem for distributed transition systems

- *language-equivalence*<sup>1</sup>, and
- *bisimulation*.

A schematic view is given in Figure 3.18: We have a number of agents with a given communication pattern (the distribution) and a global specification (the transition system) and we want to know whether there exists a local behavior for each agent (the distributed transition system) such that their synchronization complies with the specification.

A first observation is that Problem 3.70 is parametrized by the possible interpretations for ‘distributed transition system’ (two choices) and for ‘equivalence’ (three choices), giving thus altogether *six* versions of the problem.

A second observation is that we can decompose the synthesis problem in two parts:

1. **test** whether for a given distribution and transition system, there exists an equivalent distribution transition system, and
2. if the previous test is positive, **construct** the equivalent distribution transition system.

The structure of the following chapters is suggested by the above decomposition: Chapter 4 studies the computational effort of the test of distributed implementability, whereas Chapter 5 provide efficient constructions for implementable specifications.

<sup>1</sup>Following the literature (e.g., [BPS01]), one would use the term ‘trace equivalence’ instead of our usage of ‘language equivalence’. We stick though to ‘language equivalence’, because the name ‘trace equivalence’ might clash with the notion of ‘trace’ introduced in Section 3.1.

The characterizations given in the previous sections provide already some answers to the first part of the problem and partially also to the second part (recall for instance Remark 3.28). Since the presentation of solutions to various versions of Problem 3.70 needs further elaboration, we move it over to the next chapter.

## Discussion

In this chapter, we have introduced the synchronous products of transition systems and the asynchronous automata as theoretical models of distributed systems and presented characterizations of their global state spaces, respectively their behaviors. A detailed study for systems with and without final state and various particular cases was conducted.

All these results will be used to provide, when possible, decision procedures for the *implementability test* which will be presented in the next chapter. The core of the next chapter consists of upper and lower complexity bounds for the problems known until now to be decidable (there are still open problems for instance in the case of bisimulation for nondeterministic distributed systems). We will consider all six versions of Problem 3.70 together with special cases (deterministic and/or acyclic specifications).

The *effective construction* of distributed transition systems is discussed in Chapter 5, where we focus on the challenging problem of synthesis of asynchronous automata from trace-closed regular languages. Since the classical result of Zielonka produces very large asynchronous automata, in Chapter 6 we provide heuristics in an effort to find smaller asynchronous automata accepting the same language. Moreover, we give heuristics to find under-approximated solutions in case the original synthesis problem has no solution.



---

I was working on the proof of one of my poems all the morning, and took out a comma. In the afternoon I put it back again.

*Oscar Wilde*

## CHAPTER 4

# THE COMPLEXITY OF THE DISTRIBUTED IMPLEMENTABILITY TEST

---

THE distributed implementability problem is defined as follows: ‘Given a specification as a distribution of actions on a set of processes and a transition system, does there exist a distributed transition system over the given distribution that is equivalent to the transition system?’ As mentioned in the previous chapters (cf. Section 3.5 and Figure 1.2), this is the first question to ask when solving the synthesis problem (Problem 3.70). If the answer to this question is positive, we construct a solution to the problem. If the answer to the question is negative, we do not surrender but try to relax the specification so as to obtain an approximated solution to the problem (half a loaf is better than none).

In this chapter we investigate the computational complexity of the distributed implementability question together with some relaxations of it<sup>1</sup>. The chapter is organized as follows. After a short introduction to the problem (Section 4.1), we present decision procedures together with upper and lower complexity bounds for the distributed implementability modulo isomorphism (Section 4.2), modulo language equivalence (Section 4.3), respectively modulo bisimulation (Section 4.4). We consider in turn the synchronous products and then the asynchronous automata as models for distributed systems. In Section 4.5 we study a relaxed implementability problem where the distributed transition system is required to be ‘embedded into’ (rather than ‘equivalent to’) the transition system given as specification. We end the chapter with a short discussion.

For definitions and representative problems for complexity classes like P (deterministic polynomial time<sup>2</sup>), NP (nondeterministic polynomial time), PSPACE (deterministic polynomial space) *et al.*, the reader is referred to classical textbooks on computational complexity theory [GJ79, Pap94, HMU01]. For the lower bound proofs in this chapter, we use polynomial-time reductions<sup>3</sup>. (A hard-nosed reader can safely skip this chapter and go for the practical approach of next chapter.)

---

<sup>1</sup>Most of the results in the first part of his chapter were published in [HŞ04, HŞ05], while the relaxed implementability appeared in [SEM03].

<sup>2</sup>From now on, when we say ‘polynomial time’, we mean ‘deterministic polynomial time’.

<sup>3</sup>Without conducting a very careful analysis, we think that the reductions we use are in fact log-space reductions.

Table 4.1: Complexity results for the implementability of synchronous products of transition systems with one initial state ( $|I| = 1$ )

Specification ( $TS$ )	Isomorphism	Language Equivalence	Bisim. (determ. impl.)
Nondeterministic Deterministic	NP-complete P [Mor98]	PSPACE-complete	PSPACE-complete
Acyclic & Nondet. Acyclic & Determ.	NP-complete P [Mor98]	coNP-complete	coNP-complete

## 4.1 The Distributed Implementability Problem

The problem whose computational complexity we will mainly study in this chapter is:

### Problem 4.1 (Distributed implementability)

INSTANCE: *Given a distribution  $(\Sigma, Proc, \Delta)$  and a transition system  $TS$  over  $\Sigma$ ,*

QUESTION: *does there exist a distributed transition system over  $\Delta$  whose global state space is equivalent to  $TS$ ?*

*Similar to Problem 3.70, as ‘distributed transition systems’ we consider the models of synchronous products of transition systems, respectively asynchronous automata, whereas as ‘equivalence’ we consider the (transition system) isomorphism, the language equivalence, respectively the bisimulation.*

**Overview** Before going into details, we provide a short overview of the main complexity results of the first part of this chapter with some references to related work.

The distributed implementability problem has been studied for a number of abstract models of distributed systems (elementary net systems, place/transition Petri nets, synchronous products of transition systems [Arn94], and Zielonka’s asynchronous automata [Zie87]) using various behavioral equivalences between the implementation and the specification (isomorphism, language equivalence, and bisimulation). For nearly all these variants, axiomatic or language theoretic characterizations of the transition systems that can be distributed have been provided [ER90, NRT92, CKLY98, Mor98, CMT99, Vog99, BCD02, Muk02] (see also Sections 3.3 and 3.4). Moreover, the computational complexity of the variants concerning elementary net systems and place/transition Petri nets is rather well understood [BBD95, BBD97]. However, the complexity of many variants concerning synchronous products of transition systems and asynchronous automata were still open. In this chapter we fill many of these gaps, and in particular solve some problems left open in [CMT99, Mor99b].

Mukund [Muk02] surveys (structural, behavioural) characterizations for synchronous products of transition systems and asynchronous automata. He presents the results without a computational complexity analysis viewpoint. Since we are in the end interested to know which cases are tractable in practice, in this chapter we provide (the missing) lower and upper bounds for all the implementability tests presented in [Muk02].



Table 4.2: Complexity results for the implementability of asynchronous automata (with multiple initial states)

Specification ( <i>TS</i> )	Isomorphism	Language Equivalence	Bisim. (determ. impl.)
Nondeterministic	NP-complete	PSPACE-complete	P
Deterministic	P [Mor98]	P	
Acyclic & Nondet.	NP-complete	coNP-complete	P
Acyclic & Determ.	P [Mor98]	P	

Tables 4.1 and 4.2 present a summary of most of new results together with some known ones. A couple of corollaries not appearing at this point will be given in the following chapters. Note that, due to slightly different existing characterizations, the two models consider one, respectively multiple initial states (we touch upon the case of synchronous products with multiple initial states in Section 4.3.2). As already discussed in Section 3.4, we take into account special cases where the input transition system is supposed to be *deterministic* and/or *acyclic* (see column 1 of Tables 4.1 and 4.2). As expected, the complexity results for the special cases turn out to be usually more favorable than the results for the general cases. In the following we detail a little.

In [Mor98], Morin proved that the distributed implementability problem modulo isomorphism for both synchronous products and asynchronous automata can be solved in polynomial time when the input transition system is deterministic (see column 2 of Tables 4.1 and 4.2). In the nondeterministic case, results from [CMT99, Mor99b] show that the problem is in NP, but precise lower bounds were explicitly left open: In [CMT99, Section 5], the authors conjecture that ‘the synthesis problem for deterministic systems is much less expensive computationally than the synthesis problem in the general case’, while in [Mor98, Conclusion], the author mentions that ‘till now precise complexity results for the underlying synthesis problem are still missing’. We show that the implementability problem for nondeterministic distributed transition systems is NP-complete, even for acyclic specifications.

In [CMT99, Muk02], Mukund *et al.* characterized the transition systems that can be implemented as synchronous products modulo language equivalence. It is not difficult to see that the characterization leads to a PSPACE algorithm. We show that the problem is PSPACE-complete, even if the input transition system is deterministic, and coNP-complete if it is acyclic (see column 3 of Table 4.1). From these results we then easily obtain the same results for the implementability problem modulo bisimulation *when the implementation is required to be deterministic*<sup>1</sup> (see column 4 of Table 4.1). The proof is based on a characterization from [CMT99] for the implementability modulo bisimulation for deterministic implementations. Up to date there are no characterizations published for the implementability modulo bisimulation for *nondeterministic implementations* and also no known complexity bounds for it.

In [Zie87, Zie89], Zielonka characterized the transition systems that can be implemented as asynchronous automata modulo language equivalence. Combining his results

<sup>1</sup>Note that this is a natural constraint in many areas (e.g., hardware design).

with several others from the literature, we show that the implementability problem has the same complexity as for synchronous products in the nondeterministic case, but can be solved in polynomial time in the deterministic case (see column 3 of Table 4.2). Maybe surprisingly, a simple trick allows us to extend this result to the implementability problem modulo bisimulation, again when the implementation is required to be deterministic (see column 4 of Table 4.2). Up to date there are no characterizations published for the implementability modulo bisimulation for nondeterministic implementations and also no known complexity bounds for it.

## 4.2 Implementability modulo Isomorphism

This section presents complexity results for checking whether an input transition system is *isomorphic* to the global state space of a distributed transition system.

We mention that, although in practice the initial specification is usually not isomorphic to a distributed transition system, the implementability problem is still of relevance because it can be used to guide heuristics for solving the synthesis problem modulo language equivalence (see Chapter 5 and [SEM03] for more details).

### 4.2.1 Synchronous Products of Transition Systems

In Section 3.3.2 we presented a characterization result for the global transition systems of synchronous products (Theorem 3.26). For convenience, we recall below the text of the theorem and refer the reader to Section 3.3.2 for additional explanations.

**Theorem 4.2** [CMT99, Muk02] *Let  $(\Sigma, Proc, \Delta)$  be a distribution and  $TS = (Q, \Sigma, \rightarrow, I)$  be a transition system. Then, the following are equivalent:*

- (i) *TS is isomorphic to a synchronous product of transition systems over  $\Delta$*
- (ii) *For each  $p \in Proc$ , there exists an equivalence relation  $\equiv_p \subseteq Q \times Q$  such that the following conditions hold (for any  $q_1, q_2 \in Q$  and  $a \in \Sigma$ ):*

SP<sub>1</sub> : *If  $q_1 \xrightarrow{a} q_2$ , then  $q_1 \equiv_{Proc \setminus dom(a)} q_2$ .*

SP<sub>2</sub> : *If  $q_1 \equiv_{Proc} q_2$ , then  $q_1 = q_2$ .*

SP<sub>3</sub> : *Let  $a \in \Sigma$  and  $q \in Q$ . If for each  $p \in dom(a)$ , there exist  $q_p, q'_p \in Q$  such that  $q_p \xrightarrow{a} q'_p$  and  $q \equiv_p q_p$ , then for each choice of such  $q_p$ 's and  $q'_p$ 's, there exists  $q' \in Q$  such that  $q \xrightarrow{a} q'$  and  $q' \equiv_p q'_p$  for each  $p \in dom(a)$ .*

This result is used below to show that the implementability problem for synchronous products is hard even for acyclic specifications (see column 2 of Table 4.1). The result holds for nondeterministic synchronous products with multiple initial states (Theorem 4.3), but also for the case of only one initial state (Corollary 4.7).

**Theorem 4.3** *The implementability problem for synchronous products of transition systems modulo isomorphism is NP-complete, even for acyclic specifications.*

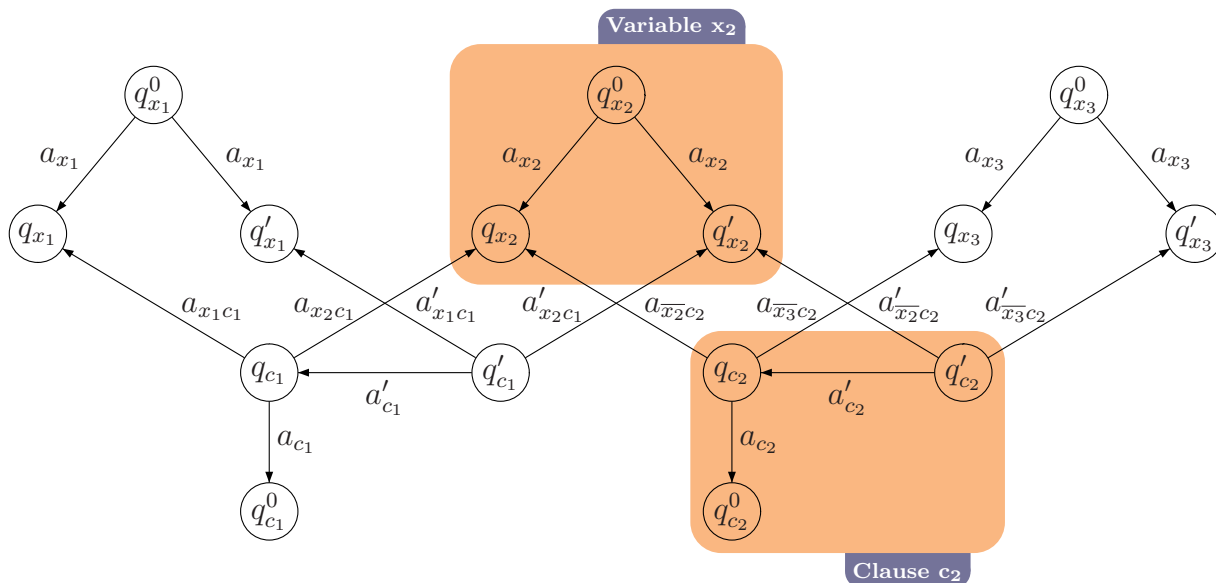


Figure 4.1: The transition system  $TS$  associated to  $\phi = (x_1 \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3})$

**Proof.** First, it is easy to see that the problem is in NP: Given a distribution  $(\Sigma, Proc, \Delta)$  and a transition system  $TS$ , a nondeterministic machine can ‘guess’ a family of equivalences  $(\equiv_p)_{p \in Proc}$  and then verify in *polynomial* time (in the size of the distribution and of the transition system), whether the properties  $SP_1$ – $SP_3$  from Theorem 4.3 are satisfied or not.

For the NP-hardness part, we use a polynomial reduction from the classical *Boolean satisfiability problem* (SAT). Before going into details, we present an overview of the translation (from a Boolean formula to a transition system together with a distribution): Given a formula in conjunctive normal form, we associate to each variable and each clause, a group of three states and two transitions (see Figure 4.1). The nondeterminism is used in the transition system to implement a choice gadget between the Boolean values **True** and **False** for each variable. We connect then the triples with edges according to the occurrence of variables as literals in the clauses (these edges will be the wires that will transmit the ‘values’ of the variables to the clauses). The distribution is chosen such that a clause will evaluate to **False** if and only if the condition  $SP_3$  will be violated for the triple associated to the given clause. The use of Theorem 4.2 finishes the job.

Now the details. Let  $\phi$  be a formula in conjunctive normal form with variables  $x_1, \dots, x_n$  appearing in the clauses  $c_1, \dots, c_m$ . For technical reasons and without loss of generality, we assume that no clause contains some variable as both positive and negative literal.

We construct a distribution  $(\Sigma_\phi, Proc_\phi, \Delta_\phi)$  and a (nondeterministic) transition system  $TS_\phi = (Q_\phi, \Sigma_\phi, \rightarrow_\phi, I_\phi)$  such that:

$\phi$  is satisfiable if and only if  $TS_\phi$  is isomorphic to a synchronous product of transition systems over  $\Delta_\phi$ .

To relieve a bit the notation, we will drop all the  $\phi$ -indices.

First, the set of processes  $Proc$  consists of two processes for each variable and one process for each clause:

$$Proc := \{p_{x_i}, p_{\bar{x}_i} \mid i \in [1..n]\} \cup \{p_{c_j} \mid j \in [1..m]\}.$$

Then, the set  $\Sigma$  of actions and their domains (which determine  $\Delta$ ) consist of:

- one action for each variable:  
 $\{a_{x_i} \mid i \in [1..n]\}$  with  $dom(a_{x_i}) := \{p_{x_i}, p_{\bar{x}_i}\}$ ,
- two actions for each positive literal of each clause:  
 $\{a_{x_i c_j}, a'_{x_i c_j} \mid j \in [1..m], x_i \in c_j\}$  with  $dom(a_{x_i c_j}) = dom(a'_{x_i c_j}) := Proc \setminus \{p_{x_i}\}$ ,
- two actions for each negative literal of each clause:  
 $\{a_{\bar{x}_i c_j}, a'_{\bar{x}_i c_j} \mid j \in [1..m], \bar{x}_i \in c_j\}$  with  $dom(a_{\bar{x}_i c_j}) = dom(a'_{\bar{x}_i c_j}) := Proc \setminus \{p_{\bar{x}_i}\}$ , and
- two actions for each clause:  
 $\{a_{c_j}, a'_{c_j} \mid j \in [1..m]\}$  with  $dom(a_{c_j}) := \{p_{c_j}\} \cup \{p_{x_i} \mid x_i \in c_j\} \cup \{p_{\bar{x}_i} \mid \bar{x}_i \in c_j\}$   
(i.e., the domain of  $a_{c_j}$  consists of the process associated to  $c_j$  and the processes associated to the literals of  $c_j$ ) and  $dom(a'_{c_j}) := Proc \setminus \{p_{c_j}\}$ .

Finally, we construct the transition system  $TS = (Q, \Sigma, \rightarrow, I)$ . The state space  $Q$  consists of:

- three states for each variable:  
 $\{q_{x_i}^0, q_{x_i}, q'_{x_i} \mid i \in [1..n]\}$  and
- three states for each clause:  
 $\{q_{c_j}, q'_{c_j}, q_{c_j}^0 \mid j \in [1..m]\}$ .

The transition relation  $\rightarrow \subseteq Q \times \Sigma \times Q$  is defined as follows:

- for each  $i \in [1..n]$ :  
 $q_{x_i}^0 \xrightarrow{a_{x_i}} q_{x_i}$  and  $q_{x_i}^0 \xrightarrow{a_{x_i}} q'_{x_i}$  (remember that nondeterminism is allowed).
- for each  $j \in [1..m]$ :  
 $q_{c_j} \xrightarrow{a_{x_i c_j}} q_{x_i}$  for  $x_i \in c_j$ ,  $q_{c_j} \xrightarrow{a_{\bar{x}_i c_j}} q_{x_i}$  for  $\bar{x}_i \in c_j$ , and  $q_{c_j} \xrightarrow{a_{c_j}} q_{c_j}^0$ .
- for each  $j \in [1..m]$ :  
 $q'_{c_j} \xrightarrow{a'_{x_i c_j}} q'_{x_i}$  for  $x_i \in c_j$ ,  $q'_{c_j} \xrightarrow{a'_{\bar{x}_i c_j}} q'_{x_i}$  for  $\bar{x}_i \in c_j$ , and  $q'_{c_j} \xrightarrow{a'_{c_j}} q_{c_j}$ .

The set of initial states  $I$  is chosen such that all states of  $Q$  are reachable from  $I$  using the above transition relation (remember that we work with reachable (distributed) transition systems). For instance, we can take

$$I := \{q_{x_i}^0 \mid i \in [1..n]\} \cup \{q'_{c_j} \mid j \in [1..m]\}$$

(it is not difficult to modify the construction such that there is only one initial state – see Corollary 4.7).

An example is provided in Figure 4.1 (the initial states are not marked).

The ‘choice gadget’ is provided by the three states for each variable  $x_i$  and their associated transitions. The Boolean ‘value’ of each choice (this will correspond to a local equivalence relation: either  $q_{x_i} \equiv_{p_{x_i}} q'_{x_i}$  or  $q_{x_i} \equiv_{p_{\bar{x}_i}} q'_{x_i}$ ) is then propagated further to the clauses using the transitions labeled  $a_{x_i c_j}$  and  $a_{\bar{x}_i c_j}$ , respectively. More precisely, to each clause we forward only the information that a variable was set to **False** in such a way that the clause  $c_j$  is not satisfied if and only if  $q_{c_j}$  and  $q'_{c_j}$  are equivalent on all processes in the domain of  $a_{c_j}$ . Thus, a clause  $c_j$  will have all its literals evaluated to **False** if and only if the condition  $\text{SP}_3$  will be violated for  $a := a_{c_j}$ ,  $q := q'_{c_j}$ , and  $q_p := q_{c_j}$ ,  $q'_p := q'_{c_j}$ .

The above construction is polynomial in the size of the initial formula  $\phi$  and we claim that  $\phi$  is satisfiable if and only if  $TS$  is isomorphic to a synchronous product of transition systems over  $\Delta$  (given by  $dom$ ).

**First Implication.** We prove first the easier part:  $\phi$  is not satisfiable implies that  $TS$  is not isomorphic to a synchronous product of transition systems over  $\Delta$ . If  $\phi$  is not satisfiable, then for any assignment of the variables  $x_1, \dots, x_n$  there exists a clause that is evaluated to **False**. We must show that in this case, there are no local equivalences  $(\equiv_p)_{p \in Proc}$  satisfying all  $\text{SP}_1$ – $\text{SP}_3$ .

By contradiction, assume that there exist  $(\equiv_p)_{p \in Proc}$  satisfying all  $\text{SP}_1$ – $\text{SP}_3$ .

For each  $i \in [1..n]$ , we use first the condition  $\text{SP}_3$  which we have assumed to hold. Let  $a := a_{x_i}$  and  $q := q'_{x_i}$ . We choose  $q_p \xrightarrow{a} q'_p$  from  $\text{SP}_3$  for each  $p \in dom(a_{x_i}) = \{p_{x_i}, p_{\bar{x}_i}\}$  as follows:  $q_{x_i} \xrightarrow{a_{x_i}} q_{x_i}$  for  $p = p_{x_i}$  and  $q_{x_i} \xrightarrow{a_{x_i}} q'_{x_i}$  for  $p = p_{\bar{x}_i}$ . Since  $q \equiv_{p_{x_i}} q_{x_i}$  and  $q \equiv_{p_{\bar{x}_i}} q'_{x_i}$  (recall that  $q = q'_{x_i}$ ), the hypothesis of  $\text{SP}_3$  is satisfied, so there must exist a state  $q'$  such that  $q_{x_i} \xrightarrow{a_{x_i}} q'$ , and also  $q' \equiv_{p_{x_i}} q_{x_i}$  and  $q' \equiv_{p_{\bar{x}_i}} q'_{x_i}$ .

There are only two possible cases for the choice of  $q'$ :

1.  $q' = q_{x_i}$ . In this case, we have  $q_{x_i} \equiv_{p_{x_i}} q_{x_i}$  and  $q_{x_i} \equiv_{p_{\bar{x}_i}} q'_{x_i}$ .
2.  $q' = q'_{x_i}$ . In this case, we have  $q'_{x_i} \equiv_{p_{x_i}} q_{x_i}$  and  $q'_{x_i} \equiv_{p_{\bar{x}_i}} q'_{x_i}$ .

So, we have that either  $q_{x_i} \equiv_{p_{\bar{x}_i}} q'_{x_i}$  (case 1) or  $q_{x_i} \equiv_{p_{x_i}} q'_{x_i}$  (case 2), but *not both at the same time* (otherwise, on one hand we have that  $q_{x_i} \equiv_{dom(a_{x_i})} q'_{x_i}$  and on the other hand, by  $\text{SP}_1$  applied to the transitions  $q_{x_i} \xleftarrow{a_{x_i}} q_{x_i} \xrightarrow{a_{x_i}} q'_{x_i}$ , we have  $q_{x_i} \equiv_{Proc \setminus dom(a_{x_i})} q'_{x_i}$ , so  $q_{x_i} \equiv_{Proc} q'_{x_i}$  which contradicts  $\text{SP}_2$ ).

Let us choose an assignment of the variables given by the equivalences in the following way. For each  $i \in [1..n]$ ,

$x_i$  is evaluated to **False** if and only if  $q_{x_i} \equiv_{p_{x_i}} q'_{x_i}$ .

Since  $\phi$  is not satisfiable, there exists a clause, say  $c_k$ , that has all its literals evaluated to **False**. Let  $x_i$  be a positive literal in  $c_k$  (if any). Since the positive literal  $x_i$  is evaluated to **False**, we have that the variable  $x_i$  is **False**, so  $q_{x_i} \equiv_{p_{x_i}} q'_{x_i}$ . In addition, we have  $q_{c_k} \xrightarrow{a_{x_i c_k}} q_{x_i}$  and  $q'_{c_k} \xrightarrow{a'_{x_i c_k}} q'_{x_i}$  (see the construction of  $TS$ ) and, using  $\text{SP}_1$ , we deduce that  $q_{c_k} \equiv_{p_{x_i}} q_{x_i}$  and  $q'_{c_k} \equiv_{p_{x_i}} q'_{x_i}$ . By the transitivity of  $\equiv_{p_{x_i}}$ , we obtain that  $q_{c_k} \equiv_{p_{x_i}} q'_{c_k}$ .

Algorithm 4.1: Construction of local equivalences for the second part of the proof of Theorem 4.3

<b>Input</b>	a satisfiable formula $\phi$ with the associated distribution $(\Sigma, Proc, \Delta)$ , transition system $TS$ , and an assignment of $x_1, \dots, x_n$ validating $\phi$
Step 0	For each $p \in Proc$ , initialize the binary relation $\equiv_p \subseteq Q \times Q$ to $\emptyset$ .
Step 1	For each $q \xrightarrow{a} q'$ in $TS$ , set $q \equiv_p q'$ for every $p \in Proc \setminus dom(a)$ .
Step 2	For each $i \in [1..n]$ , if variable $x_i$ is evaluated to <b>False</b> , then set $q_{x_i} \equiv_{p_{x_i}} q'_{x_i}$ , otherwise set $q_{x_i} \equiv_{p_{\bar{x}_i}} q'_{x_i}$ .
Step 3	For each $p \in Proc$ , close $\equiv_p$ under reflexivity, symmetry, and transitivity.
<b>Output</b>	a set of local equivalences $(\equiv_p)_{p \in Proc}$ satisfying the $SP_1$ – $SP_3$ conditions

A similar argument for the negative literals  $\bar{x}_i$  in  $c_k$  (if any) proves that  $q_{c_k} \equiv_{p_{\bar{x}_i}} q'_{c_k}$  ( $q_{x_i} \equiv_{p_{\bar{x}_i}} q'_{x_i}$  is used). Moreover, using  $SP_1$  for  $q'_{c_k} \xrightarrow{a_{c_k}} q_{c_k}$ , we have that  $q_{c_k} \equiv_{p_{c_k}} q'_{c_k}$ . Summing up, we proved that  $q'_{c_k} \equiv_p q_{c_k}$ , for each  $p \in dom(a_{c_k})$  (recall the definition of  $dom(a_{c_k})$ ). But this contradicts  $SP_3$ , because  $q_{c_k} \xrightarrow{a_{c_k}} q_{c_k}^0$  and there is no state  $q'$  such that  $q'_{c_k} \xrightarrow{a_{c_k}} q'$ .

**Second Implication.** The second part of the proof is a bit technical. Assume that  $\phi$  is satisfiable. Then, there exists an assignment to the variables  $x_1, \dots, x_n$  such that each clause is **True**. Given this assignment, we construct a family of equivalences  $(\equiv_p)_{p \in Proc}$  following Algorithm 4.1 and prove that the generated equivalences satisfy the conditions  $SP_1$ – $SP_3$  of Theorem 4.2, and therefore the transition system  $TS$  is isomorphic to a synchronous product of transition systems over  $(\Sigma, Proc, \Delta)$ .

Table 4.3 describes the equivalence classes of the equivalences generated by Algorithm 4.1. Each cell gives the partition of the state space  $Q$  into the equivalence classes for each type of process (rows) depending on the value of the associated variable (columns). Each equivalence class is given as a set in curly brackets. It is tedious, but not hard to check the correctness of Table 4.3 (i.e., the equivalence classes presented there are indeed the equivalence classes of the local equivalences generated by Algorithm 4.1).

Using Table 4.3, we prove that the constructed  $(\equiv_p)_{p \in Proc}$  satisfy  $SP_1$ ,  $SP_3$ , and  $SP_2$  (in this order):

**$SP_1$  satisfaction** : Condition  $SP_1$  is fulfilled by construction (cf. Step 1 of Algorithm 4.1).

**$SP_3$  satisfaction** : To benefit the presentation, we make a couple of remarks.

**Remark 4.4**  $SP_3$  holds for action  $a := a_{x_i}$ , for each  $i \in [1..n]$ .

**Proof.** Let  $i \in [1..n]$  and  $q \in Q$ . Following  $SP_3$ , for each  $p \in dom(a_{x_i}) = \{p_{x_i}, p_{\bar{x}_i}\}$ , we try to choose  $q_p$  and  $q'_p$  such that  $q_p \xrightarrow{a_{x_i}} q'_p$  and  $q \equiv_p q_p$ . Since the only transitions

Table 4.3: The equivalence classes constructed by Algorithm 4.1

	$x_i = \text{False}$	$x_i = \text{True}$
$\equiv_{p_{x_i}}$	$\{q_{x_i}^0\}$ , $\{q_{x_{i'}}, q_{x_{i'}}, q'_{x_{i'}}\}$ for each $i' \neq i$ , $\{q_{x_i}, q'_{x_i}\} \cup \{q_{c_j}, q'_{c_j} \mid x_i \in c_j\}$ , $\{q_{c_j}^0\}$ for each $c_j$ containing lit. $x_i$ , and $\{q_{c_{j'}}, q_{c_{j'}}^0\}$ , $\{q'_{c_{j'}}\}$ for each $c_{j'}$ not containing the positive literal $x_i$	$\{q_{x_i}^0\}$ , $\{q_{x_{i'}}, q_{x_{i'}}, q'_{x_{i'}}\}$ for each $i' \neq i$ , $\{q_{x_i}\} \cup \{q_{c_j} \mid x_i \in c_j\}$ , $\{q'_{x_i}\} \cup \{q'_{c_j} \mid x_i \in c_j\}$ , $\{q_{c_j}^0\}$ for each $c_j$ containing lit. $x_i$ , and $\{q_{c_{j'}}, q_{c_{j'}}^0\}$ , $\{q'_{c_{j'}}\}$ for each $c_{j'}$ not containing the positive literal $x_i$
$\equiv_{p_{\bar{x}_i}}$	$\{q_{x_i}^0\}$ , $\{q_{x_{i'}}, q_{x_{i'}}, q'_{x_{i'}}\}$ for each $i' \neq i$ , $\{q_{x_i}\} \cup \{q_{c_j} \mid \bar{x}_i \in c_j\}$ , $\{q'_{x_i}\} \cup \{q'_{c_j} \mid \bar{x}_i \in c_j\}$ , $\{q_{c_j}^0\}$ for each $c_j$ containing lit. $\bar{x}_i$ , and $\{q_{c_{j'}}, q_{c_{j'}}^0\}$ , $\{q'_{c_{j'}}\}$ for each $c_{j'}$ not containing the negative literal $\bar{x}_i$	$\{q_{x_i}^0\}$ , $\{q_{x_{i'}}, q_{x_{i'}}, q'_{x_{i'}}\}$ for each $i' \neq i$ , $\{q_{x_i}, q'_{x_i}\} \cup \{q_{c_j}, q'_{c_j} \mid \bar{x}_i \in c_j\}$ , $\{q_{c_j}^0\}$ for each $c_j$ containing lit. $\bar{x}_i$ , and $\{q_{c_{j'}}, q_{c_{j'}}^0\}$ , $\{q'_{c_{j'}}\}$ for each $c_{j'}$ not containing the negative literal $\bar{x}_i$
$\equiv_{p_{c_j}}$	$\{q_{x_i}^0, q_{x_i}, q'_{x_i}\}$ for $i \in [1..n]$ , $\{q_{c_j}, q'_{c_j}\}$ , $\{q_{c_j}^0\}$ , and $\{q_{c_{j'}}, q_{c_{j'}}^0\}$ , $\{q'_{c_{j'}}\}$ for $j' \neq j$	

 Table 4.4: Details for the satisfaction of the  $\text{SP}_3$  property

$a$	$q$	Why $\text{SP}_3$ holds (a process $p$ from $\text{dom}(a)$ is given)
$a_{x_i}$		See Remark 4.4.
$a_{x_i c_j}$ (for $x_i \in c_j$ )	$\{q_{c_j}^0\}$ $Q \setminus \{q_{c_j}, q_{c_j}^0\}$	$p := p_{c_j}$ . Indeed, $q_{c_j} \not\equiv_{p_{c_j}} q_{c_j}^0$ . $p := p_{\bar{x}_i}$ . Indeed, $x_i \in c_j$ implies $\bar{x}_i \notin c_j$ (see assumptions on $\phi$ ), so $q_{c_j} \not\equiv_{p_{\bar{x}_i}} q$ , $\forall q \in Q \setminus \{q_{c_j}, q_{c_j}^0\}$ .
$a'_{x_i c_j}$ (for $x_i \in c_j$ )	$Q \setminus \{q'_{c_j}\}$	$p := p_{\bar{x}_i}$ . Same as above, $x_i \in c_j$ implies $\bar{x}_i \notin c_j$ and in this case $q'_{c_j} \not\equiv_{p_{\bar{x}_i}} q$ , $\forall q \in Q \setminus \{q'_{c_j}\}$ .
$a_{\bar{x}_i c_j}, a'_{\bar{x}_i c_j}$ (for $\bar{x}_i \in c_j$ )		Similar to the cases $a_{x_i c_j}, a'_{x_i c_j}$ above.
$a_{c_j}$	$\{q'_{c_j}\}$ $Q \setminus \{q_{c_j}, q'_{c_j}\}$	Since $c_j$ evaluates to <b>True</b> , there exists a literal $\ell$ of $c_j$ evaluated to <b>True</b> . Assume $\ell = x_i$ , such that $x_i \in c_j$ and $x_i = \text{True}$ (a similar analysis is made if $\ell$ is negative). Then, for $p := p_{x_i}$ we have that $p_{x_i} \in \text{dom}(a_{c_j})$ and $q_{c_j} \not\equiv_{p_{x_i}} q'_{c_j}$ . $p := p_{c_j}$ . Indeed, $q_{c_j} \not\equiv_{p_{c_j}} q$ , $\forall q \in Q \setminus \{q_{c_j}, q'_{c_j}\}$ .
$a'_{c_j}$	$Q \setminus \{q'_{c_j}\}$	For a given variable $x_i$ , the literals $x_i$ and $\bar{x}_i$ cannot appear both in $c_j$ . If $x_i \notin c_j$ , then $p := p_{x_i}$ (and indeed, $q'_{c_j} \not\equiv_{p_{x_i}} q$ , $\forall q \in Q \setminus \{q'_{c_j}\}$ ). If $\bar{x}_i \notin c_j$ , then $p := p_{\bar{x}_i}$ .



labeled with  $a_{x_i}$  are  $q_{x_i}^0 \xrightarrow{a_{x_i}} q_{x_i}$  and  $q_{x_i}^0 \xrightarrow{a_{x_i}} q'_{x_i}$ ,  $q_p$  must be equal to  $q_{x_i}^0$  and  $q'_p$  must belong to  $\{q_{x_i}, q'_{x_i}\}$ , for each  $p \in \text{dom}(a_{x_i})$ . Moreover, if in  $q \equiv_p q_p = q_{x_i}^0$  we instantiate  $p$  with  $p_{x_i}$  (which belongs to  $\text{dom}(a_{x_i})$ ), then we obtain  $q \equiv_{p_{x_i}} q_{x_i}^0$ . Since the  $\equiv_{p_{x_i}}$ -equivalence class of  $q_{x_i}^0$  is  $\{q_{x_i}^0\}$  (see Table 4.3), we deduce that  $q = q_{x_i}^0$ . We have now that the hypotheses of  $\text{SP}_3$  are satisfied.

For the various choices for  $q'_p \in \{q_{x_i}, q'_{x_i}\}$  with  $p \in \{p_{x_i}, p_{\bar{x}_i}\}$ , we will find a state  $q'$  such that  $q = q_{x_i}^0 \xrightarrow{a_{x_i}} q'$  and  $q' \equiv_p q'_p$  for each  $p \in \{p_{x_i}, p_{\bar{x}_i}\}$ : If  $q'_{p_{x_i}} = q'_{p_{\bar{x}_i}} := q_{x_i}$ , take  $q' := q_{x_i}$ . If  $q'_{p_{x_i}} = q'_{p_{\bar{x}_i}} := q'_{x_i}$ , take  $q' := q'_{x_i}$ . If  $q'_{p_{x_i}} := q_{x_i}$  and  $q'_{p_{\bar{x}_i}} := q'_{x_i}$ , we have two subcases: If  $x_i$  is **False**, take  $q' := q'_{x_i}$ . This is correct, because we have  $q'_{x_i} \equiv_{p_{x_i}} q_{x_i}$  (by Step 2 of Algorithm 4.1) and  $q'_{x_i} \equiv_{p_{\bar{x}_i}} q'_{x_i}$  (by reflexivity). If  $x_i$  is **True**, take  $q' := q_{x_i}$ . This is correct for similar reasons as above. The last case,  $q'_{p_{x_i}} := q'_{x_i}$  and  $q'_{p_{\bar{x}_i}} := q_{x_i}$ , is dual to the one above.  $\square$

Next, we note that each action  $a \in \Sigma \setminus \{a_{x_i} \mid i \in [1..n]\}$  has the property that there exists *exactly one* transition labeled with  $a$  in the transition system  $TS$ . In this case, the condition  $\text{SP}_3$  allows a simplified version:

**Remark 4.5** Let  $a \in \Sigma$  such that there is only one transition, denoted  $q(a) \xrightarrow{a} q'(a)$ , labeled with  $a$  in  $TS$ . Then,  $\text{SP}_3$  holds for the chosen  $a$  if and only if for each state  $q \neq q(a)$ , there exists a process  $p \in \text{dom}(a)$  such that  $q \not\equiv_p q(a)$ .

**Proof.** ( $\Rightarrow$ ) Assume that  $\text{SP}_3$  holds for the given action  $a$ . By contradiction, assume that there exists  $q \neq q(a)$  such that  $q \equiv_{\text{dom}(a)} q(a)$ . The hypothesis of  $\text{SP}_3$  holds for the above  $a$ ,  $q$ , and  $q_p, q'_p$  chosen to be  $q(a), q'(a)$ , respectively. Then, there must exist a state  $q'$  such that  $q \xrightarrow{a} q'$  (and also  $q' \equiv_p q'(a), \forall p \in \text{dom}(a)$ ). But this is a contradiction, because  $q \xrightarrow{a} q'$  would be an  $a$ -labeled transition *different* than the supposedly unique transition  $q(a) \xrightarrow{a} q'(a)$ .

( $\Leftarrow$ ) Assume now that  $\forall q \neq q(a) \exists p \in \text{dom}(a) : q \not\equiv_p q(a)$ . We must prove that  $\text{SP}_3$  holds for the given  $a$ . Let  $q, q_p, q'_p \in Q$  such that  $q_p \xrightarrow{a} q'_p$  and  $q \equiv_p q_p$ , for each  $p \in \text{dom}(a)$ . Since  $q(a) \xrightarrow{a} q'(a)$  is the only  $a$ -labeled transition, we have that  $q_p = q(a)$  and  $q'_p = q'(a), \forall p \in \text{dom}(a)$ . This implies  $q \equiv_{\text{dom}(a)} q(a)$  (because  $q \equiv_p q_p, \forall p \in \text{dom}(a)$ ). Using the assumption that  $\forall q \neq q(a) \exists p \in \text{dom}(a) : q \not\equiv_p q(a)$ , we necessarily have that  $q = q(a)$ . Now, it is easy to find a state  $q'$  satisfying  $q \xrightarrow{a} q'$  and  $q' \equiv_p q'_p = q'(a), \forall p \in \text{dom}(a)$ : We simply choose  $q' := q'(a)$ .  $\square$

Since Remark 4.4 shows that  $\text{SP}_3$  holds for  $a \in \{a_{x_i} \mid i \in [1..n]\}$ , the remaining cases are solved in Table 4.4 using Remark 4.5. The first column of Table 4.4 picks a value for  $a$ , while the second one gives a range to the state  $q \in Q \setminus \{q(a)\}$ , for the  $q(a)$  from the formulation of Remark 4.5. In the last column, we give a process  $p \in \text{dom}(a)$  with the property that  $q(a) \not\equiv_p q$ , for *all* the states  $q$  in the range given in the second column.

The correctness of the solutions provided is verified using the equivalence classes given in Table 4.3. For example, let us look at the second row of Table 4.4, where



$a = a_{x_i c_j}$  and  $q = q_{c_j}^0$ . Then, according to Remark 4.5, we must find a process  $p \in \text{dom}(a_{x_i c_j}) = \text{Proc} \setminus \{p_{x_i}\}$  such that  $q \not\equiv_p q(a_{x_i c_j})$  (in our case  $q = q_{c_j}^0$  and  $q(a_{x_i c_j}) = q_{c_j}$  because  $q_{c_j}$  is the only state with an outgoing transition labeled by  $a_{x_i c_j}$ ). Table 4.4 (row 2, column 3) gives  $p_{c_j}$  as possible choice for  $p$ . Indeed,  $q_{c_j}^0 \not\equiv_{p_{c_j}} q_{c_j}$ . This can be verified looking at the last row of Table 4.3, where we see that  $q_{c_j}^0$  and  $q_{c_j}$  belong to different equivalence classes.

**SP<sub>2</sub> satisfaction** : An immediate restatement of SP<sub>2</sub> is given below:

**Remark 4.6** SP<sub>2</sub> holds if and only if for each  $q_1 \neq q_2$ , there exists  $p \in \text{Proc}$  with  $q_1 \not\equiv_p q_2$ .

According to the above remark, for each pair of *distinct* states, we must find a process  $p$  where they are not  $\equiv_p$ -equivalent. Table 4.4 provides already such processes that ‘distinguish’ some pairs of different states. In Table 4.5 we give the rest of the cases: For  $q_1$  (column 1) and  $q_2$  (in the *range* given by column 2), we find a process  $p \in \text{Proc}$  such that  $q_1 \not\equiv_p q_2$  (column 3). More precisely, we only have to consider pairs of states from the subset:

$$Q' := \{q_{x_i}^0, q_{x_i}, q'_{x_i} \mid i \in [1..n]\} \cup \{q_{c_j}^0 \mid j \in [1..m]\}.$$

We mention that an alternative presentation for the second part of the proof may be given, and namely: instead of constructing local equivalence classes  $(\equiv_p)_{p \in \text{Proc}}$  (and prove SP<sub>1</sub>–SP<sub>3</sub>), to construct directly local transition systems whose synchronous product is isomorphic to the transition system  $TS$ . We think that this approach would not be more intuitive and according to (the proof of) Theorem 4.2 we would still have to consider all the cases in the current proof when proving the isomorphism between the synchronous product and  $TS$ .  $\square$

Looking at Theorem 4.2, we realize that the set of initial states does not really play a rôle in the implementability modulo isomorphism (where the structure is important), and therefore the result of Theorem 4.3 holds also for the synchronous products with only one initial state:

**Corollary 4.7** *The implementability problem modulo isomorphism for synchronous products with only one initial state is NP-complete, even for acyclic specifications.*

**Proof.** It is easy to modify the reduction in the proof of Theorem 4.3 such that the constructed transition system  $TS$  has only one initial state (and thus also the distributed implementation has only one initial state). The idea is to add a new state  $q^{in}$ , a set of new actions, one for each state of  $I$ ,  $\{a_q \mid q \in I\}$ , and transitions  $q^{in} \xrightarrow{a_q} q$  from  $q^{in}$  to each state of  $I$  (thus all the states will be reachable from the new state). We choose the domains of the new actions to be the set of all processes, i.e.,  $\text{dom}(a_q) := \text{Proc}$  for each  $q \in I$  (we do not modify the set of processes  $\text{Proc}$ ). Finally, we update the set of initial states as  $I := \{q^{in}\}$ .

It is not difficult to see that the arguments used for the proof of Theorem 4.3 still apply for the above additions. (For the first implication, the contradiction is obtained

similarly, whereas for the second implication, since the domain of each new action  $a_q$  is  $Proc$ , by Algorithm 4.1 (see Step 1),  $q^{in} \not\equiv_p q$  for any  $p \in Proc$  and  $q \neq q^{in}$  which easily implies that  $SP_1$ – $SP_3$  hold for the added part.)  $\square$

Going into the proof details of Theorem 4.2 given in [CMT99], we can show that if there exists a set of equivalences  $(\equiv_p)_{p \in Proc}$  satisfying only conditions  $SP_1$  and  $SP_3$  (but not necessarily  $SP_2$ ), then we can synthesize a synchronous product of transition systems accepting the same language as the initial transition system.<sup>1</sup> This simple trick widens the class of ‘implementable’ transition systems, while preserving the behavior. Yet, the new problem is as hard as the implementability modulo isomorphism for synchronous products (Theorem 4.3), from which we do the reduction:

**Corollary 4.8** *Let  $(\Sigma, Proc, \Delta)$  be a distribution and  $TS$  a transition system. The problem of finding a set of equivalences  $(\equiv_p)_{p \in Proc}$  satisfying only conditions  $SP_1$  and  $SP_3$  of Theorem 4.2, is NP-complete.*

**Proof.** The problem is in NP for the same reasons given in the proof of Theorem 4.3.

To prove that the problem is NP-hard, we use a reduction from the implementability problem for synchronous products modulo isomorphism (Theorem 4.3): For each  $(\Sigma, Proc, \Delta)$  and  $TS = (Q, \Sigma, \rightarrow, I)$ , we construct a distribution  $(\Sigma', Proc', \Delta')$  and a transition system  $TS' = (Q', \Sigma', \rightarrow', I')$  in the following way:

- $\Sigma' := \Sigma \cup \{a_q \mid q \in Q\}$ , i.e., we add one new action  $a_q$  for each state of  $q \in Q$ .
- $Proc' := Proc$ , i.e., we do not modify the set of processes  $Proc$ .
- $\Delta' := \Delta \cup \{(a_q, p) \mid q \in Q, p \in Proc\}$ , i.e., we have  $dom'(a) := dom(a)$  for all  $a \in \Sigma$  and  $dom'(a_q) := Proc$  for all  $q \in Q$ .
- $Q' := Q \cup \{q_0\}$  where  $q_0$  is a new state and  $I' := I$ .
- $\rightarrow' := \rightarrow \cup \{q \xrightarrow{a_q} q_0 \mid q \in Q\}$ , i.e., we add one transition from each state of  $TS$  to the new state  $q_0$ . For simplicity, we denote  $\rightarrow'$  also by  $\rightarrow$ .

We prove that: There exist  $(\equiv_p)_{p \in Proc}$  on the states of  $TS$  satisfying  $SP_1$ – $SP_2$ – $SP_3$  of Theorem 4.2 (i.e.,  $TS$  is isomorphic to a synchronous product over  $\Delta$ ) if and only if there exist  $(\equiv'_p)_{p \in Proc}$  on the states of  $TS'$  satisfying  $SP_1$  and  $SP_3$  for the new  $(\Sigma', Proc, \Delta')$ .

For the direct implication, let us assume that there exist  $(\equiv_p)_{p \in Proc}$  on the states of  $TS$  satisfying  $SP_1$ – $SP_2$ – $SP_3$ . We extend each  $\equiv_p \subseteq Q \times Q$  to  $\equiv'_p \subseteq Q' \times Q'$ , simply by choosing

$$\equiv'_p := \equiv_p \cup \{(q_0, q_0)\}$$

(i.e., the new state  $q_0$  is equivalent only to itself). We prove that  $SP_1$  and  $SP_3$  hold for  $(\equiv'_p)_{p \in Proc}$ :

<sup>1</sup>In fact, the synthesized synchronous product is even bisimilar (Definition 2.12) to the initial transition system.

$SP_1$  : Because  $SP_1$  holds for the transitions in  $TS$  and  $dom'(a) := dom(a)$  for all  $a \in \Sigma$ , we only have to prove that  $SP_1$  holds for the newly added transitions  $q \xrightarrow{a_q} q_0$ , but this is trivial given the fact that  $dom'(a_q) := Proc$ .

$SP_3$  : We prove that  $SP_3$  holds for each  $a \in \Sigma' = \Sigma \cup \{a_q \mid q \in Q\}$  distinguishing two cases:

- For  $a \in \Sigma$ ,  $SP_3$  holds in  $TS'$  because  $SP_3$  holds for  $TS$  (this is easy).
- For  $a = a_q$  with  $q \in Q$ , by construction, we have  $dom(a) = Proc$  and  $q \xrightarrow{a} q_0$ . Since the transition  $q \xrightarrow{a} q_0$  is *the only* one labeled with  $a = a_q$  in  $TS'$ , we prove that  $SP_3$  holds for  $a$  in  $TS'$  using Remark 4.5. Thus, we have to show that for each  $r \in Q'$  with  $r \neq q$ , there exists a process  $p \in dom(a) = Proc$  such that  $r \not\equiv'_p q$ . We have two subcases:
  - For  $r \in Q$ , we have that  $r$  and  $q$  belong to  $Q$  and  $r \neq q$ . From the hypothesis,  $(\equiv_p)_{p \in Proc}$  satisfies  $SP_2$ . In this case, we can apply Remark 4.6 for  $r \neq q$  and obtain that there exists a process  $p \in Proc$  such that  $r \not\equiv_p q$ , so also  $r \not\equiv'_p q$ .
  - For  $r = q_0$ , by construction,  $r = q_0 \not\equiv'_p q$  for any  $q \in Q$  and  $p \in Proc$ .

For the reverse implication, assume that there exist  $(\equiv'_p)_{p \in Proc}$  on the states of  $TS'$  satisfying  $SP_1$  and  $SP_3$ . Let us choose  $(\equiv_p)_{p \in Proc}$  as the projection of  $(\equiv'_p)_{p \in Proc}$  on  $Q \times Q$ . It is easy to see that  $SP_1$  and  $SP_3$  are satisfied for  $TS$  and  $(\equiv_p)_{p \in Proc}$  because the same properties hold for  $TS'$  and  $(\equiv'_p)_{p \in Proc}$ . To prove  $SP_2$ , we use the equivalent condition given by Remark 4.6:  $SP_2$  holds for  $TS$  and  $(\equiv_p)_{p \in Proc}$  if and only if for each pair of states  $q \neq r$  from  $Q$ , there exists a process  $p \in Proc$  such that  $q \not\equiv_p r$ . Let  $q \neq r$  in  $Q$ . Since  $q \xrightarrow{a_q} q_0$  is the only transition labeled with  $a_q$  from  $TS'$  and  $SP_3$  holds for  $TS'$  and  $(\equiv'_p)_{p \in Proc}$ , we can apply Remark 4.5 and obtain that there exists a process  $p \in dom'(a_q) = Proc$  such that  $q \not\equiv'_p r$ . But this implies also that  $q \not\equiv_p r$  (because  $q, r \in Q$ ).  $\square$

Efficient decision procedures for the implementability problem modulo isomorphism are available in case the specification is deterministic (see column 2 of the overview Table 4.1) as the following result shows:

**Proposition 4.9** [Mor98] *The implementability problem for synchronous products modulo isomorphism is decidable in polynomial time, if the input transition system is deterministic.*

**Proof.** See Theorem 3.29 and the accompanying comments.

### 4.2.2 Asynchronous Automata

Same complexity results for implementability modulo isomorphism to those for synchronous products of transition systems (Section 4.2.1) apply to asynchronous automata (see column 2 of the overview Table 4.2). The proofs will follow more or less the same pattern, but since the synchronization mechanism is different, so will be the proof details.

In Section 3.3.3 we presented a characterization result for the global transition systems of asynchronous automata (Theorem 3.30). For convenience, we recall below the text of the theorem and refer the reader to Section 3.3.3 for additional explanations.

**Theorem 4.10** [Mor99b, Muk02] *Let  $(\Sigma, Proc, \Delta)$  be a distribution and  $TS = (Q, \Sigma, \rightarrow, I)$  be a transition system. Then, the following are equivalent:*

- (i) *TS is isomorphic to an asynchronous automaton over  $\Delta$*
- (ii) *For each  $p \in Proc$ , there exists an equivalence relation  $\equiv_p \subseteq Q \times Q$  such that the following conditions hold (for any  $q_1, q_2 \in Q$  and  $a \in \Sigma$ ):*

**AA<sub>1</sub>** : *If  $q_1 \xrightarrow{a} q_2$ , then  $q_1 \equiv_{Proc \setminus dom(a)} q_2$ .*

**AA<sub>2</sub>** : *If  $q_1 \equiv_{Proc} q_2$ , then  $q_1 = q_2$ .*

**AA<sub>3</sub>** : *If  $q_1 \xrightarrow{a} q'_1$  and  $q_1 \equiv_{dom(a)} q_2$ , then there exists  $q'_2 \in Q$  such that  $q_2 \xrightarrow{a} q'_2$  and  $q'_2 \equiv_{dom(a)} q'_1$ .*

This result is used below to show that the implementability problem for asynchronous automata is hard even for acyclic specifications (see column 2 of Table 4.1). The result holds for nondeterministic synchronous products with multiple initial states (Theorem 4.11), but also for the case of only one initial state (Corollary 4.13).

**Theorem 4.11** *The implementability problem for asynchronous automata modulo isomorphism is NP-complete, even for acyclic specifications.*

**Proof.** First, it is easy to see that the problem is in NP: Given a distribution  $(\Sigma, Proc, \Delta)$  and a transition system  $TS$ , a nondeterministic machine can ‘guess’ a family of equivalences  $(\equiv_k)_{k \in Proc}$  and then verify in *polynomial* time (in the size of the distribution and of the transition system) whether the properties **AA<sub>1</sub>**–**AA<sub>3</sub>** from Theorem 4.10 are satisfied or not.

For the NP-hardness part, we use a polynomial reduction from the classical *Boolean satisfiability problem* (SAT). Before going into details, we present an overview of the translation (from a Boolean formula to a transition system together with a distribution): Given a formula in conjunctive normal form, we associate to each variable and each clause, a group of states and transitions (see Figure 4.2). The nondeterminism is used in the transition system to implement a choice gadget between the Boolean values **True** and **False** for each variable. We connect further with edges the group of states according to the occurrence of variables as literals in the clauses (these edges will be the wires that will transmit the ‘values’ of the variables to the clauses). The distribution is chosen such that a clause will evaluate to **False** if and only if the condition **AA<sub>3</sub>** will be violated for the triple associated to the given clause. The use of Theorem 4.10 finishes the job.

Let  $\phi$  be a formula in conjunctive normal form with variables  $x_1, \dots, x_n$  appearing in the clauses  $c_1, \dots, c_m$ . For technical reasons, we construct first a new formula  $\phi'$  that satisfy the following properties:

1. each variable appears in at least two different clauses,
2. each clause contains at least two different literals, and

3.  $\phi'$  is satisfiable if and only if  $\phi$  is satisfiable.

The formula  $\phi'$  is constructed from  $\phi$  as follows. We start with  $\phi'$  being equal to a new clause  $c_0 := x_0 \vee \bar{x}_0 \vee x_1 \vee x_2 \vee \dots \vee x_n$ , where  $x_0$  is a fresh variable. Then we add to  $\phi'$  the clauses of  $\phi$  that contain at least two different literals. Finally, if  $c_j := x_i$  (respectively,  $c_j := \bar{x}_i$ ) is a clause of  $\phi$ , we add to  $\phi'$  two new clauses  $x_i \vee x_0$  and  $x_i \vee \bar{x}_0$  (respectively,  $\bar{x}_i \vee x_0$  and  $\bar{x}_i \vee \bar{x}_0$ ). It is easy to see that  $\phi'$  satisfies the three properties above. For convenience, we denote  $\phi'$  also by  $\phi$  in the following.

We will construct a distribution  $(\Sigma_\phi, Proc_\phi, \Delta_\phi)$  and a (nondeterministic) transition system  $TS_\phi = (Q_\phi, \Sigma_\phi, \rightarrow_\phi, I_\phi)$  such that:

$\phi$  is satisfiable if and only if  $TS_\phi$  is isomorphic to an asynchronous automaton over  $\Delta_\phi$ .

To relieve a bit the notation, we will drop all the  $\phi$ -indices.

First, the set of processes  $Proc$  consists of one process for each (positive or negative) literal from each clause:

$$Proc := \{p_{x_i c_j} \mid j \in [0..m], x_i \in c_j\} \cup \{p_{\bar{x}_i c_j} \mid j \in [0..m], \bar{x}_i \in c_j\}.$$

Then, the set  $\Sigma$  of actions and their domains (which determine  $\Delta$ ) consist of:

- one action for each positive literal from each clause:  
 $\{a_{x_i c_j} \mid j \in [0..m], x_i \in c_j\}$  with  $dom(a_{x_i c_j}) := Proc \setminus \{p_{x_i c_j}\}$ .
- one action for each negative literal from each clause:  
 $\{a_{\bar{x}_i c_j} \mid j \in [0..m], \bar{x}_i \in c_j\}$  with  $dom(a_{\bar{x}_i c_j}) := Proc \setminus \{p_{\bar{x}_i c_j}\}$ .
- one action for each variable:  
 $\{a_{x_i} \mid i \in [0..n]\}$  with the domain of each  $a_{x_i}$  consisting of the processes associated to the literals where  $x_i$  appears:  

$$dom(a_{x_i}) := \bigcup_{j: x_i \in c_j} \{p_{x_i c_j}\} \cup \bigcup_{j: \bar{x}_i \in c_j} \{p_{\bar{x}_i c_j}\}.$$
- one action for each clause:  
 $\{a_{c_j} \mid j \in [0..m]\}$  with the domain of each  $a_{c_j}$  consisting of the processes associated to the literals of  $c_j$ :  

$$dom(a_{c_j}) := \bigcup_{i: x_i \in c_j} \{p_{x_i c_j}\} \cup \bigcup_{i: \bar{x}_i \in c_j} \{p_{\bar{x}_i c_j}\}.$$

Finally, we construct the transition system  $TS = (Q, \Sigma, \rightarrow, I)$ . The state space  $Q$  consist of:

- six states for each variable:  
 $\{q_{x_i}^0, q_{x_i}^1, q'_{x_i}, q_{x_i}, q_{x_i}^F, q_{x_i}^T \mid i \in [0..n]\}$  and
- three states for each clause:  
 $\{q_{c_j}, q'_{c_j}, q_{c_j}^0 \mid j \in [0..m]\}.$

The transition relation  $\rightarrow \subseteq Q \times \Sigma \times Q$  is defined below:

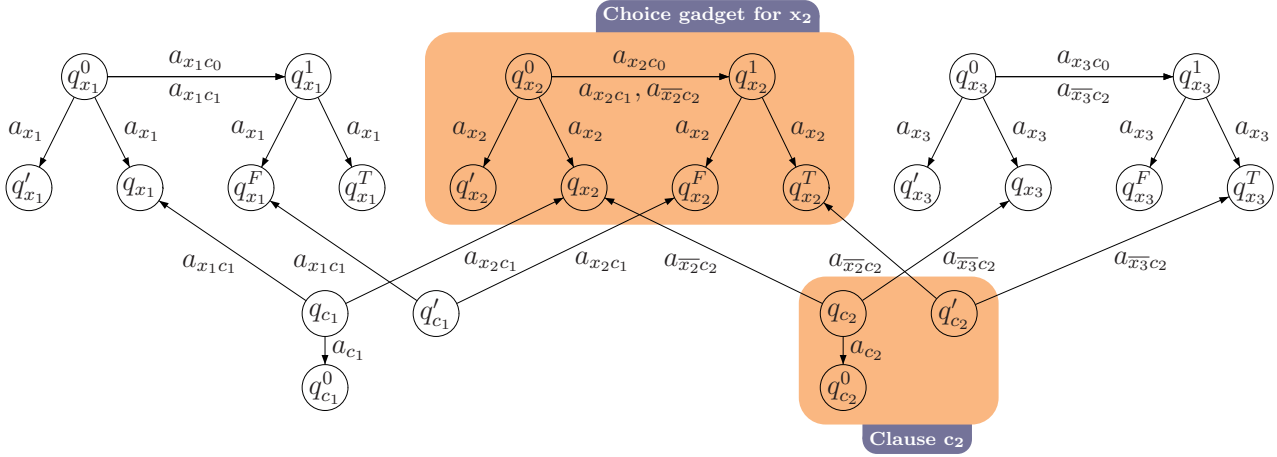


Figure 4.2: The transition system  $TS$  associated to  $\phi = (x_1 \vee x_2) \wedge (\bar{x}_2 \vee \bar{x}_3)$  (without the states and transitions associated to clause  $c_0$ )

- for each  $i \in [0..n]$ :  
 $q_{x_i}^0 \xrightarrow{a_{x_i}} q'_{x_i}$ ,  $q_{x_i}^0 \xrightarrow{a_{x_i}} q_{x_i}$ , and  
 $q_{x_i}^0 \xrightarrow{a} q_{x_i}^1$  for each action  $a \in \{a_{x_i c_j} \mid x_i \in c_j\} \cup \{a_{\bar{x}_i c_j} \mid \bar{x}_i \in c_j\}$ .
- for each  $i \in [0..n]$ :  
 $q_{x_i}^1 \xrightarrow{a_{x_i}} q_{x_i}^F$  and  $q_{x_i}^1 \xrightarrow{a_{x_i}} q_{x_i}^T$ .
- for each  $j \in [0..m]$ :  
 $q_{c_j} \xrightarrow{a_{x_i c_j}} q_{x_i}$  for  $x_i \in c_j$ ,  $q_{c_j} \xrightarrow{a_{\bar{x}_i c_j}} q_{x_i}$  for  $\bar{x}_i \in c_j$ , and  $q_{c_j} \xrightarrow{a_{c_j}} q_{c_j}^0$ .
- for each  $j \in [0..m]$ :  
 $q'_{c_j} \xrightarrow{a_{x_i c_j}} q_{x_i}^F$  for  $x_i \in c_j$  and  $q'_{c_j} \xrightarrow{a_{\bar{x}_i c_j}} q_{x_i}^T$  for  $\bar{x}_i \in c_j$ .

The set of initial states  $I$  is chosen such that all states of  $Q$  are reachable from  $I$  using the above transition relation (remember that we work with reachable (distributed) transition systems). For instance, we can take

$$I := \{q_{x_i}^0 \mid i \in [0..n]\} \cup \{q_{c_j}, q'_{c_j} \mid j \in [0..m]\}$$

(it is not difficult to modify the construction such that there is only one initial state – see Corollary 4.13).

An example is provided in Figure 4.2. We omitted, for legibility, the states  $q_{c_0}$ ,  $q'_{c_0}$ , and  $q_{c_0}^0$  and their associated transitions (these states were introduced by the clause  $c_0$  for some consistency reasons – see the assumptions on  $\phi$  at the beginning of the proof). Also, we did not mark the initial states, because they do not play any important rôle here.

The ‘choice gadget’ is provided by the six states for each variable  $x_i$  and their associated transitions. The Boolean ‘value’ of each choice (this will be reflected by local equivalence relations as: either  $q_{x_i} \equiv_{dom(a_{x_i})} q_{x_i}^F$  or  $q_{x_i} \equiv_{dom(a_{x_i})} q_{x_i}^T$ ) is then propagated further to the



clauses using the transitions labeled  $a_{x_i c_j}$  and  $a_{\bar{x}_i c_j}$ , respectively. More precisely, to each clause we forward only the information that a variable was set to **False** in such a way that the clause  $c_j$  is not satisfied if and only if  $q_{c_j}$  and  $q'_{c_j}$  are equivalent on all processes in the domain of  $a_{c_j}$ . Thus, a clause  $c_j$  will have all its literals evaluated to **False** if and only if the condition  $\text{AA}_3$  will be violated for  $q_1 := q_{c_j}$ ,  $q'_1 := q_{c_j}^0$ ,  $q_2 := q'_{c_j}$ .

The above construction is polynomial in the size of the initial formula  $\phi$  and we claim that  $\phi$  is satisfiable if and only if  $TS$  is isomorphic to an asynchronous automaton over  $\Delta$  (given by  $\text{dom}$ ).

**First Implication.** We first prove the easier part:  $\phi$  is not satisfiable implies that  $TS$  is not isomorphic to an asynchronous automaton over  $\Delta$ . If  $\phi$  is not satisfiable, then for any assignment of the variables  $x_0, \dots, x_n$  there exists a clause that evaluates to **False**. We must show that in this case, there are no local equivalences  $(\equiv_p)_{p \in \text{Proc}}$  satisfying all  $\text{AA}_1$ – $\text{AA}_3$ .

By contradiction, assume that there exist  $(\equiv_p)_{p \in \text{Proc}}$  satisfying all  $\text{AA}_1$ – $\text{AA}_3$ .

For each  $i \in [0..n]$ , from  $q_{x_i}^0 \xrightarrow{a} q_{x_i}^1$  for all  $a \in \{a_{x_i c_j} \mid x_i \in c_j\} \cup \{a_{\bar{x}_i c_j} \mid \bar{x}_i \in c_j\}$ , using  $\text{AA}_1$ , we have that  $q_{x_i}^0 \equiv_p q_{x_i}^1$  for all  $p \in \{p_{x_i c_j} \mid x_i \in c_j\} \cup \{p_{\bar{x}_i c_j} \mid \bar{x}_i \in c_j\}$ , and so  $q_{x_i}^0 \equiv_{\text{dom}(a_{x_i})} q_{x_i}^1$ . Next, from  $q_{x_i}^0 \equiv_{\text{dom}(a_{x_i})} q_{x_i}^1$  and  $q_{x_i}^0 \xrightarrow{a_{x_i}} q_{x_i}$ , using  $\text{AA}_3$ , we have that either  $q_{x_i} \equiv_{\text{dom}(a_{x_i})} q_{x_i}^F$ , or  $q_{x_i} \equiv_{\text{dom}(a_{x_i})} q_{x_i}^T$ , but *not both in the same time* (otherwise, by transitivity we have  $q_{x_i}^F \equiv_{\text{dom}(a_{x_i})} q_{x_i}^T$  and also  $q_{x_i}^F \equiv_{\text{Proc} \setminus \text{dom}(a_{x_i})} q_{x_i}^T$  by  $\text{AA}_1$  applied to  $q_{x_i}^F \xleftarrow{a_{x_i}} q_{x_i}^1 \xrightarrow{a_{x_i}} q_{x_i}^T$ , so  $q_{x_i}^F \equiv_{\text{Proc}} q_{x_i}^T$  and this would contradict  $\text{AA}_2$ ). Let us choose an assignment of the variables given by the equivalences in the following way. For each  $i \in [0..n]$ ,

$$x_i \text{ is evaluated to False if and only if } q_{x_i} \equiv_{\text{dom}(a_{x_i})} q_{x_i}^F.$$

Since  $\phi$  is not satisfiable, there exists a clause, say  $c_k$ , that has all its literals evaluated to **False**. Let  $x_i$  be a positive literal in  $c_k$  (if any). Since the literal  $x_i$  is evaluated to **False**, we have that the variable  $x_i$  is **False**, so  $q_{x_i} \equiv_{\text{dom}(a_{x_i})} q_{x_i}^F$  and this implies  $q_{x_i} \equiv_{p_{x_i c_k}} q_{x_i}^F$  (because  $p_{x_i c_k} \in \text{dom}(a_{x_i})$ ). In addition, we have  $q_{c_k} \xrightarrow{a_{x_i c_k}} q_{x_i}$  and  $q'_{c_k} \xrightarrow{a_{x_i c_k}} q_{x_i}^F$  (see the construction of  $TS$ ) and, using  $\text{AA}_1$ , we deduce that  $q_{c_k} \equiv_{p_{x_i c_k}} q_{x_i}$  and  $q'_{c_k} \equiv_{p_{x_i c_k}} q_{x_i}^F$ . By transitivity of  $\equiv_{p_{x_i c_k}}$ , we obtain that  $q_{c_k} \equiv_{p_{x_i c_k}} q'_{c_k}$ . A similar argument for the negative literals  $\bar{x}_i$  in  $c_k$  (if any), proves that  $q_{c_k} \equiv_{p_{\bar{x}_i c_k}} q'_{c_k}$ .

Summing up,  $q_{c_k}$  and  $q'_{c_k}$  are equivalent on all the processes associated to the literals in  $c_k$ , and so, by the definition of  $\text{dom}(a_{c_k})$ , we obtain that  $q_{c_k} \equiv_{\text{dom}(a_{c_k})} q'_{c_k}$ . But this contradicts  $\text{AA}_3$  because  $q_{c_k} \xrightarrow{a_{c_k}} q_{c_k}^0$  and there is no state  $q'$  such that  $q'_{c_k} \xrightarrow{a_{c_k}} q'$ .

**Second Implication.** We move now to the second part of the proof (which will be a bit technical). Assume that  $\phi$  is satisfiable. Then, there exists an assignment to the variables  $x_0, \dots, x_n$  such that each clause is **True**. Given this assignment, we construct a family of equivalences  $(\equiv_p)_{p \in \text{Proc}}$  following Algorithm 4.2 and prove that the generated equivalences satisfy the conditions  $\text{AA}_1$ – $\text{AA}_3$  of Theorem 4.10, and therefore the transition system  $TS$  is isomorphic to an asynchronous automaton over  $(\Sigma, \text{Proc}, \Delta)$ .

Let us start making some remarks regarding Algorithm 4.2:

Table 4.5: Details for the satisfaction of the  $SP_2$  property (only the cases not solved already by Table 4.4)

$q_1$	$q_2$	Why $SP_2$ holds (a process $p$ from $Proc$ is given)
$q_{x_i}^0$	$Q' \setminus \{q_{x_i}^0\}$	$p := p_{x_i}$ . Indeed, $q_{x_i}^0 \not\equiv_{p_{x_i}} q, \forall q \in Q' \setminus \{q_{x_i}^0\}$ .
$q_{x_i}$	$\{q_{x_i}^0, q'_{x_i}\}$	If $x_i$ is <b>True</b> , then $p := p_{x_i}$ , else $p := p_{\bar{x}_i}$ .
	$Q' \setminus \{q_{x_i}^0, q_{x_i}, q'_{x_i}\}$	$p := p_{c_j}$ for an arbitrary $j \in [1..m]$ .
$q'_{x_i}$	Similar to the case $q_{x_i}$ above.	
$q_{c_j}^0$	$Q' \setminus \{q_{c_j}^0\}$	$p := p_{c_j}$ .

Algorithm 4.2: Construction of local equivalences for the second part of the proof of Theorem 4.11

<b>Input</b>	a satisfiable formula $\phi$ with the associated distribution $(\Sigma, Proc, \Delta)$ , transition system $TS$ , and an assignment of $x_1, \dots, x_n$ validating $\phi$
Step 0	For each $p \in Proc$ , initialize the binary relation $\equiv_p \subseteq Q \times Q$ to $\emptyset$ .
Step 1	For each $q \xrightarrow{a} q'$ in $TS$ , set $q \equiv_p q'$ for every $p \in Proc \setminus dom(a)$ .
Step 2	For each $i \in [1..n]$ , if variable $x_i$ is evaluated to <b>False</b> , then set $q_{x_i} \equiv_p q_{x_i}^F$ and $q'_{x_i} \equiv_p q_{x_i}^T$ , for every $p \in dom(a_{x_i})$ , otherwise set $q_{x_i} \equiv_p q_{x_i}^T$ and $q'_{x_i} \equiv_p q_{x_i}^F$ , for every $p \in dom(a_{x_i})$ .
Step 3	For each $p \in Proc$ , close $\equiv_p$ under reflexivity, symmetry, and transitivity.
<b>Output</b>	a set of local equivalences $(\equiv_p)_{p \in Proc}$ satisfying the $AA_1$ – $AA_3$ conditions



Table 4.6: The equivalence classes constructed by Algorithm 4.2

	$x_i = \text{False}$	$x_i = \text{True}$
$\equiv_{p_{x_i c_j}}$ (for $x_i \in c_j$ )	$\{q_{x_i}^0, q_{x_i}^1\}, \{q'_{x_i}, q_{x_i}^T\},$ $\{q_{x_i}, q_{x_i}^F, q_{c_j}, q'_{c_j}\}, \{q_{c_j}^0\},$ for each $i' \neq i, \{q_{x_{i'}}^0, q'_{x_{i'}}, q_{x_{i'}}\},$ $\{q_{x_{i'}}^1, q_{x_{i'}}^F, q_{x_{i'}}^T\},$ and for each $j' \neq j, \{q_{c_{j'}}^0, q'_{c_{j'}}\}, \{q'_{c_{j'}}\}$	$\{q_{x_i}^0, q_{x_i}^1\}, \{q'_{x_i}, q_{x_i}^F, q'_{c_j}\},$ $\{q_{x_i}, q_{x_i}^T, q_{c_j}\}, \{q_{c_j}^0\}$ for each $i' \neq i, \{q_{x_{i'}}^0, q'_{x_{i'}}, q_{x_{i'}}\},$ $\{q_{x_{i'}}^1, q_{x_{i'}}^F, q_{x_{i'}}^T\},$ and for each $j' \neq j, \{q_{c_{j'}}^0, q'_{c_{j'}}\}, \{q'_{c_{j'}}\}$
$\equiv_{p_{\bar{x}_i c_j}}$ (for $\bar{x}_i \in c_j$ )	$\{q_{x_i}^0, q_{x_i}^1\}, \{q'_{x_i}, q_{x_i}^T, q'_{c_j}\},$ $\{q_{x_i}, q_{x_i}^F, q_{c_j}\}, \{q_{c_j}^0\}$ for each $i' \neq i, \{q_{x_{i'}}^0, q'_{x_{i'}}, q_{x_{i'}}\},$ $\{q_{x_{i'}}^1, q_{x_{i'}}^F, q_{x_{i'}}^T\},$ and for each $j' \neq j, \{q_{c_{j'}}^0, q'_{c_{j'}}\}, \{q'_{c_{j'}}\}$	$\{q_{x_i}^0, q_{x_i}^1\}, \{q'_{x_i}, q_{x_i}^F\},$ $\{q_{x_i}, q_{x_i}^T, q_{c_j}, q'_{c_j}\}, \{q_{c_j}^0\},$ for each $i' \neq i, \{q_{x_{i'}}^0, q'_{x_{i'}}, q_{x_{i'}}\},$ $\{q_{x_{i'}}^1, q_{x_{i'}}^F, q_{x_{i'}}^T\},$ and for each $j' \neq j, \{q_{c_{j'}}^0, q'_{c_{j'}}\}, \{q'_{c_{j'}}\}$

- Step 1 directly imposes  $\mathbf{AA}_1$  to be satisfied.
- Step 2 implements the choice gadget for the variables: For each  $i \in [0..n]$ , from  $q_{x_i}^0 \xrightarrow{a} q_{x_i}^1$  for all  $a \in \{a_{x_i c_j} \mid x_i \in c_j\} \cup \{a_{\bar{x}_i c_j} \mid \bar{x}_i \in c_j\}$ , using Step 1 (i.e.,  $\mathbf{AA}_1$ ), we have that  $q_{x_i}^0 \equiv_{\text{dom}(a_{x_i})} q_{x_i}^1$ . It is not difficult to check that  $\mathbf{AA}_3$  holds for the states  $q_{x_i}^0, q_{x_i}^1$  and each of the  $a_{x_i}$ -labeled edges coming out of them. For example, let us take  $q_{x_i}^0, q_{x_i}^1$ , for which we have  $q_{x_i}^0 \xrightarrow{a_{x_i}} q_{x_i}$  and  $q_{x_i}^0 \equiv_{\text{dom}(a_{x_i})} q_{x_i}^1$ . The first part of  $\mathbf{AA}_3$  is satisfied, so there must exist a state  $q'$  such that  $q_{x_i}^1 \xrightarrow{a_{x_i}} q'$  and  $q_{x_i} \equiv_{\text{dom}(a_{x_i})} q'$ . If  $x_i$  is **False**, we take  $q' := q_{x_i}^F$ , otherwise  $q' := q_{x_i}^T$ . Loosely speaking, we can tell the value of  $x_i$  checking which of the states  $q_{x_i}^F$  and  $q_{x_i}^T$  is equivalent on  $\text{dom}(a_{x_i})$  with  $q_{x_i}$ .
- Step 3 ensures that  $(\equiv_p)_{p \in \text{Proc}}$  are equivalences.
- Moreover, Step 3 transmits further the information from the variables to the literals in the clauses. In the example in Figure 4.2,  $x_2$  appears in  $c_1$ . If  $x_2$  is evaluated to **False**, then we know that  $q_{x_2} \equiv_{\text{dom}(a_{x_2})} q_{x_2}^F$  (by Step 2) and this implies  $q_{x_2} \equiv_{p_{x_2 c_1}} q_{x_2}^F$ . We also have  $q_{c_1} \xrightarrow{a_{x_2 c_1}} q_{x_2}$  and  $q'_{c_1} \xrightarrow{a_{x_2 c_1}} q_{x_2}^F$  and, using Step 1, we deduce that  $q_{c_1} \equiv_{p_{x_2 c_1}} q_{x_2}$  and  $q'_{c_1} \equiv_{p_{x_2 c_1}} q_{x_2}^F$ . By Step 3 (i.e., transitivity of  $\equiv_{p_{x_2 c_1}}$ ), we obtain that  $q_{c_1} \equiv_{p_{x_2 c_1}} q'_{c_1}$ . Therefore, we have that  $x_2 = \text{False}$  (at the logical level) implies  $q_{c_1} \equiv_{p_{x_2 c_1}} q'_{c_1}$  (at the level of equivalences).

Table 4.6 describes the equivalence classes of the equivalences generated by Algorithm 4.2. Each cell gives the partition of the state space  $Q$  into the equivalence classes for each type of process (rows) depending on the value of the associated variable (columns). Each equivalence class is given as a set in curly brackets. It is tedious, but not hard to check the correctness of Table 4.6 (i.e., the equivalence classes presented there are indeed the equivalence classes of the local equivalences generated by Algorithm 4.2).

Using Table 4.6, we prove that the constructed  $(\equiv_p)_{p \in \text{Proc}}$  satisfy  $\mathbf{AA}_1$ ,  $\mathbf{AA}_3$ , and  $\mathbf{AA}_2$  (in this order):

Table 4.7: Details for the satisfaction of the  $AA_3$  property

$q_1$	$q_1 \xrightarrow{a} q'_1$	$q_2$	Why $AA_3$ holds (either a state $q'_2$ or a process $p \in \text{dom}(a)$ are given)
$q_{x_i}^0$	$q_{x_i}^0 \xrightarrow{a_{x_i}} q'_{x_i}$	$\{q_{x_i}^1\}$	We have $q_{x_i}^0 \equiv_{\text{dom}(a_{x_i})} q_{x_i}^1$ (because of Step 1; see also Table 4.6). If $x_i$ is <b>True</b> , choose $q'_2 := q_{x_i}^F$ , otherwise choose $q'_2 := q_{x_i}^T$ .
		$Q \setminus \{q_{x_i}^0, q_{x_i}^1\}$	$p := p_{x_i c_0}$ .
	$q_{x_i}^0 \xrightarrow{a_{x_i}} q_{x_i}$	Similar to the case $q_{x_i}^0 \xrightarrow{a_{x_i}} q'_{x_i}$ above.	
	$q_{x_i}^0 \xrightarrow{a_{x_i c_j}} q_{x_i}^1$ (for $x_i \in c_j$ )	$\{q_{x_i}^1\}$	$p := p_{x_i' c_0}$ , where $i' \neq i$ (by constr. $x_{i'} \in c_0$ ).
	$Q \setminus \{q_{x_i}^0, q_{x_i}^1\}$	$p$ can be any process from $\text{dom}(a_{x_i}) \setminus \{p_{x_i c_j}\}$ . Such $p$ exists, because $ \text{dom}(a_{x_i})  \geq 2$ (remember that we forced each variable of $\phi$ to appear in at least two clauses).	
	$q_{x_i}^0 \xrightarrow{a_{\bar{x}_i c_j}} q_{x_i}^1$ (for $\bar{x}_i \in c_j$ )	Similar to the case $q_{x_i}^0 \xrightarrow{a_{x_i c_j}} q_{x_i}^1$ above.	
$q_{x_i}^1$	Similar to the case $q_{x_i}^0$ above.		
$q_{c_j}$	$q_{c_j} \xrightarrow{a_{x_i c_j}} q_{x_i}$ (for $x_i \in c_j$ )	$\{q_{c_j}^0\}$	We choose $p$ to be the process associated to a literal $\ell$ of $c_j$ different <sup>a</sup> from $x_i$ . So, $p := p_{\ell c_j}$ for $\ell \in c_j$ and $\ell \neq x_i$ .
		$Q \setminus \{q_{c_j}, q_{c_j}^0\}$	$p := p_{\ell c_j'}$ , for $c_j'$ different <sup>b</sup> from $c_j$ and $\ell \in c_j'$
	$q_{c_j} \xrightarrow{a_{\bar{x}_i c_j}} q_{x_i}$ (for $\bar{x}_i \in c_j$ )	Similar to the case $q_{c_j} \xrightarrow{a_{x_i c_j}} q_{x_i}$ above.	
	$q_{c_j} \xrightarrow{a_{c_j}} q_{c_j}^0$	Since $c_j$ evaluates to <b>True</b> , there exists a literal $\ell$ of $c_j$ evaluated to <b>True</b> . Assume $\ell = x_i$ , such that $x_i \in c_j$ and $x_i = \text{True}$ (a similar analysis is made if $\ell$ is negative). Then:	
	$\{q_{x_i}, q_{x_i}^T\}$	$p := p_{\ell c_j}$ for $\ell \in c_j$ and $\ell \neq x_i$ <sup>a</sup> .	
	$Q \setminus \{q_{c_j}, q_{x_i}, q_{x_i}^T\}$	$p := p_{x_i c_j}$ .	
$q'_{c_j}$	Similar to the case $q_{c_j}$ above.		

<sup>a</sup>Such literal  $\ell \neq x_i$  exists because we forced each clause to contain at least two different literals.

<sup>b</sup>Our formula  $\phi$  contains at least two clauses.

**AA<sub>1</sub> satisfaction** : Condition AA<sub>1</sub> is fulfilled by construction (Step 1 of Algorithm 4.2).

**AA<sub>3</sub> satisfaction** : We start with an observation regarding AA<sub>3</sub>:

**Remark 4.12** A *sufficient* condition for AA<sub>3</sub> to hold for  $q_1 \neq q_2$  and  $q_1 \xrightarrow{a} q'_1$  is that there exists  $p \in \text{dom}(a)$  such that  $q_1 \not\equiv_p q_2$ .

**Proof.** Immediate: If there exists  $p \in \text{dom}(a)$  such that  $q_1 \not\equiv_p q_2$ , then  $q_1 \not\equiv_{\text{dom}(a)} q_2$ , so the hypothesis of AA<sub>3</sub> does not hold, which gives that the implication of AA<sub>3</sub> holds ('false implies everything').  $\square$

Table 4.7 shows why AA<sub>3</sub> holds by systematically considering instantiations for the elements present in the formulation of AA<sub>3</sub>. Thus, the first column picks a value for  $q_1$ , the second one gives the transition  $q_1 \xrightarrow{a} q'_1$ , while the third column gives a range to  $q_2$ . In the last column we either find a state  $q'_2$  as in the formulation of AA<sub>3</sub> (in case  $q_1 \equiv_{\text{dom}(a)} q_2$ ) or we give a process  $p \in \text{dom}(a)$  such that  $q_1 \not\equiv_p q_2$  as in Remark 4.12.

The correctness of the solutions provided is verified using the equivalence classes from Table 4.6.

**AA<sub>2</sub> satisfaction** : Since AA<sub>2</sub> has the same formulation as SP<sub>2</sub>, we can reuse the restatement of SP<sub>2</sub> from Remark 4.6, which is:

AA<sub>2</sub> holds if and only if for each  $q_1 \neq q_2$ , there exists  $p \in \text{Proc}$  with  $q_1 \not\equiv_p q_2$ .

According to the above remark, for each pair of *distinct* states, we must find a process where they are not equivalent. Table 4.7 provides already such processes that 'distinguish' some pairs of different states: The columns 1 and 3 give the pairs  $q_1 \neq q_2$ , whereas column 4 gives a process  $p$  such that  $q_1 \not\equiv_p q_2$  (except for the column 1, row 1 ( $q_1 := q_{x_i}^0$ ,  $q_2 := q_{x_i}^1$ ) which gives a state  $q'_2$  as in the formulation of AA<sub>3</sub>; nevertheless, row 4 gives the process  $p := p_{x_i', c_0}$  with  $i' \neq i$  to distinguish the states  $q_{x_i}^0$  and  $q_{x_i}^1$ ). In Table 4.8 we solve the rest of the cases: For  $q_1$  (column 1 of Table 4.8) and  $q_2$  (in the *range* given by column 2 of Table 4.8), we give a process  $p \in \text{Proc}$  such that  $q_1 \not\equiv_p q_2$  (column 3 of Table 4.8). More precisely, we only have to consider pairs of states from the subset:

$$Q' := \{q'_{x_i}, q_{x_i}, q_{x_i}^F, q_{x_i}^T \mid i \in [0..n]\} \cup \{q_{c_j}^0 \mid j \in [0..m]\}.$$

$\square$

The result of Theorem 4.11 holds also for the asynchronous automata with only one initial state:

**Corollary 4.13** *The implementability problem modulo isomorphism for asynchronous automata with only one initial state is NP-complete, even for acyclic specifications.*

**Proof.** Similar to the proof of Corollary 4.7.  $\square$

Similar to Corollary 4.8 (and with a similar proof) we have:

**Corollary 4.14** *Let  $(\Sigma, Proc, \Delta)$  be a distribution and  $TS$  a transition system. The problem of finding a set of equivalences  $(\equiv_p)_{p \in Proc}$  satisfying only conditions  $AA_1$  and  $AA_3$  of Theorem 4.10, is NP-complete.*

Efficient decision procedures for the implementability problem modulo isomorphism are available in case the specification is deterministic (see column 2 of the overview Table 4.2) as the following result shows:

**Proposition 4.15** [Mor98] *The implementability problem for asynchronous automata modulo isomorphism is decidable in polynomial time, if the input transition system is deterministic.*

**Proof.** See Theorem 3.32 and the accompanying comments.

### 4.2.3 Implementability for Concurrent Alphabets

A small variation to the implementability problem (Problem 4.1) is obtained by giving in the initial specification a concurrent alphabet  $(\Sigma, \parallel)$  (cf. Definition 3.1) instead of a distribution:

**Problem 4.16** *Given a concurrent alphabet  $(\Sigma, \parallel)$  and a transition system  $TS$  over  $\Sigma$ , does there exist a distributed transition system over a distribution  $(\Sigma, Proc, \Delta)$  equivalent to  $TS$  and such that the independence relation generated by  $\Delta$  is equal to  $\parallel$ ?*

The problem was considered in several papers: [Mor98, CMT99, Mor99a, Mor99b, Muk02]. Since, according to Section 3.2.1, there exist several possible distributions generating the same independence relation, we expect Problem 4.16 to be ‘harder’ than Problem 4.1.

We will only take a look at one specific case which requires a small clarification (in our opinion). More precisely, Morin [Mor98, Section 3] shows that Problem 4.16 modulo isomorphism with the extra requirement that the number of processes of the distribution is minimal, is NP-complete, even for deterministic specifications (i.e.,  $TS$  is deterministic). Moreover, he mentions that Problem 4.16 modulo isomorphism for *deterministic specifications* is in NP and a lower bound was left open. We tried our luck in finding a lower bound to the problem, but the attempt was unsuccessful. Nevertheless, we will provide a proof for the NP upper bound for asynchronous automata (as we will see later the hint for the proof given in [Mor99b] is not complete).

According to Propositions 4.9 and 4.15, the distributed implementability (Problem 4.1) modulo isomorphism can be solved in polynomial time for deterministic specifications and the test is given by Theorems 3.29 and 3.32. A naïve NP algorithm for solving Problem 4.16 would then be: Guess a distribution generating the concurrent alphabet and test in polynomial time whether the transition system is distributable. This does not work because, for a fixed concurrent alphabet  $(\Sigma, \parallel)$  there might exist several distributions generating it (cf. Section 3.2.1) and, more importantly, the size of these distributions may be *exponential* in the size of the (in)dependence relation between actions: The set of the local alphabets must be a covering by cliques of the dependence graph  $(\Sigma, \parallel)$  and the

number of all maximal cliques of a graph may be exponential in the size of the graph (cf. Section 2.1.4). A solution to this issue is given in [Mor98, Mor99a] for synchronous products by the following lemma:

**Lemma 4.17** *If there exists a solution to Problem 4.16 for deterministic specifications over a distribution  $\Delta$ , then there exists also a solution over a distribution  $\Delta'$  such that the size of  $\Delta'$  is smaller than a polynomial in the size of the specification (i.e.,  $\|$  and  $TS$ ).*

For synchronous products, the smaller distribution  $\Delta'$  is constructed by selecting some of the processes of  $\Delta$  using Theorem 3.29. We do not insist further, but provide more details in the same direction for the case of asynchronous automata.

In [Mor99b], the same idea of finding a solution with less processes from one with more processes (Lemma 4.17) is used to prove that:

**Proposition 4.18** *Problem 4.16 modulo isomorphism for asynchronous automata and deterministic specifications is in NP.*

The NP algorithm runs as follows: Guess a distribution  $\Delta$  generating  $(\Sigma, \|\|)$  with the number of processes bounded by a certain polynomial in the size of  $\|$  and  $TS$  and then test in polynomial time using Theorem 3.32 whether  $TS$  is isomorphic to an asynchronous automaton over  $\Delta$ . By Lemma 4.17 we know that considering only a certain subclass of (polynomially bounded) distributions it is enough to cover the problem for all possible distributions generating  $(\Sigma, \|\|)$ . So the only duty now is to make sure that Lemma 4.17 holds for asynchronous automata. As a preliminary, we recall below for convenience the characterization theorem modulo isomorphism for asynchronous automata in case the specification is deterministic:

**Theorem 4.19** [Mor98] *Let  $(\Sigma, Proc, \Delta)$  be a distribution and  $TS = (Q, \Sigma, \rightarrow, \{q^{in}\})$  be a deterministic transition system. Then, we have:*

1. *There exists the least family of equivalences  $(\equiv_p)_{p \in Proc}$  over the states of  $Q$  such that the following conditions hold (for any  $q_1, q'_1, q_2, q'_2 \in Q$ ,  $a \in \Sigma$ , and  $p \in Proc$ ):*

**DAA<sub>1</sub>** : *If  $q_1 \xrightarrow{a} q_2$ , then  $q_1 \equiv_{Proc \setminus dom(a)} q_2$ .*

**DAA<sub>2</sub>** : *If  $q_1 \xrightarrow{a} q'_1$ ,  $q_2 \xrightarrow{a} q'_2$ , and  $q_1 \equiv_{dom(a)} q_2$ , then  $q'_1 \equiv_{dom(a)} q'_2$ .*

2. *The following are equivalent:*

(i)  *$TS$  is isomorphic to an asynchronous automaton over  $\Delta$ .*

(ii) *For the equivalences  $(\equiv_k)_{k \in Proc}$  from 1. the following conditions hold:*

**DAA<sub>3</sub>** : *If  $q_1 \equiv_{Proc} q_2$ , then  $q_1 = q_2$ .*

**DAA<sub>4</sub>** : *If  $q_1 \xrightarrow{a} q'_1$  and  $q_1 \equiv_{dom(a)} q_2$ , then there exists  $q'_2 \in Q$  such that  $q_2 \xrightarrow{a} q'_2$ .*

The construction for the proof of Lemma 4.17 proposed in [Mor99b] makes use of the above theorem in the following way. The hypothesis of Lemma 4.17 says that there exists an asynchronous automata  $\mathcal{AA}$  over a distribution  $(\Sigma, Proc, \Delta)$  such that  $\mathcal{AA}$  is

isomorphic to  $TS$  and  $(\Sigma, Proc, \Delta)$  generates the given  $\parallel$ . According to Theorem 4.19, the least family of equivalences  $(\equiv_p)_{p \in Proc}$  satisfying  $DAA_1$  and  $DAA_2$ , must necessarily also satisfy  $DAA_3$  and  $DAA_4$ . Using  $(\equiv_p)_{p \in Proc}$ , from  $(\Sigma, Proc, \Delta)$  one constructs a new distribution  $(\Sigma, Proc', \Delta')$  such that  $Proc'$  is obtained from  $Proc$  selecting only those processes involved in the satisfaction of  $DAA_3$  and  $DAA_4$  as follows. For each pair of distinct states  $(q, r)$ , there must exist a process  $p$  such that  $q \not\equiv_p r$  in order to fulfill  $DAA_3$  (cf. Remark 4.6). Furthermore, for each pair of states  $(q, r)$  and each action  $a \in \Sigma$  such that  $q \xrightarrow{a} q'$  and  $\neg(r \xrightarrow{a} r')$ , there must exist a process  $p \in dom(a)$  such that  $q \not\equiv_p r$  in order to fulfill  $DAA_4$ . Thus,  $Proc'$  consists of the processes  $p$  previously mentioned, in all at most  $|Q|^2 + |Q|^2|\Sigma|$  processes, and additionally another at most  $|\Sigma|^2$  processes to make sure that for each pair of dependent actions  $a \parallel b$  there exists a process  $p$  such that  $p \in dom(a) \cap dom(b)$ . (For each selected process  $p \in Proc' \subseteq Proc$ , we preserve the local alphabet, i.e.,  $\Sigma_{loc}'(p) := \Sigma_{loc}(p)$ , or to put it in another way,  $\Delta' := \Delta \cap (\Sigma \times Proc')$ .)

The above idea for the construction of a smaller distribution  $\Delta'$  works fine for synchronous products (using the DSP conditions of Theorem 3.29 instead of DAA), because everything ‘happens’ locally (not much interaction), but it does not really work for asynchronous automata, where a more complex interaction is involved. The following (counter)example tries to make this point. Consider the distribution  $(\Sigma, Proc, \Delta)$  and the transition system  $TS$  from Figure 4.3. It is easy to verify using Theorem 4.19 that there exists an asynchronous automaton over  $\Delta$  isomorphic to  $TS$ . Selecting processes from  $Proc$  to construct  $Proc'$  as described above, we can obtain for instance

$$Proc' := \{1, 2, 3\}$$

which gives the following new domains for the actions (by intersecting  $dom$  with  $Proc'$ ):

$$dom'(a) = \{1, 2\}, \quad dom'(b) = \{2, 3\}, \quad \text{and} \quad dom'(c) = \{1, 3\}.$$

However, using Theorem 4.19, we can show that for the new distribution,  $TS$  is *not* isomorphic to an asynchronous automaton over  $(\Sigma, Proc', \Delta')$ . Let us denote the least family of equivalences satisfying  $DAA_1$  and  $DAA_2$  for the new distribution by  $(\equiv'_p)_{p \in Proc'}$ . By  $DAA_1$  applied to  $1 \xrightarrow{b} 2$ , respectively  $2 \xrightarrow{c} 1$ , we have  $1 \equiv'_1 2$ , respectively  $1 \equiv'_2 2$ , so  $1 \equiv'_{dom'(a)} 2$ . From the last fact together with  $1 \xrightarrow{a} 3$  and  $2 \xrightarrow{a} 4$ , by  $DAA_2$ , we have  $3 \equiv'_{dom'(a)} 4$  and we obtain a contradiction with  $DAA_4$ , because  $3 \xrightarrow{a} 5$ , but  $\neg(4 \xrightarrow{a})$ . This contradiction cannot be obtained for the original distribution because  $DAA_2$  cannot be applied in the first place to the transitions  $1 \xrightarrow{a} 3$  and  $2 \xrightarrow{a} 4$  (the domain of  $a$ ,  $dom(a) = \{1, 2, 4\}$ , contains also the process 4 and we have  $1 \not\equiv_4 2$ ). The point is that considering processes involved in the fulfillment of  $DAA_3$  and  $DAA_4$  is not enough and we must adapt the construction to take also  $DAA_2$  into account. This will be done in the new proof for Lemma 4.17 proposed below.

**Proof.** (of Lemma 4.17)

Let  $(\Sigma, \parallel)$  a fixed concurrent alphabet and  $TS = (Q, \Sigma, \rightarrow, \{q^{in}\})$  a deterministic transition system. Let also  $(\Sigma, Proc, \Delta)$  be a distribution generating  $(\Sigma, \parallel)$  such that the least family of equivalences  $(\equiv_p)_{p \in Proc}$  satisfying  $DAA_1$  and  $DAA_2$ , also satisfies  $DAA_3$  and  $DAA_4$ . We construct a new distribution  $(\Sigma, Proc', \Delta')$  in the following way (note that the alphabet of actions remains the same):

The distribution	Transition system								
$\Sigma := \{a, b, c\},$ $Proc := \{1, 2, 3, 4\}$ <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th></th> <th><i>dom</i></th> </tr> </thead> <tbody> <tr> <td><i>a</i></td> <td>{1, 2, 4}</td> </tr> <tr> <td><i>b</i></td> <td>{2, 3, 4}</td> </tr> <tr> <td><i>c</i></td> <td>{1, 3, 4}</td> </tr> </tbody> </table>		<i>dom</i>	<i>a</i>	{1, 2, 4}	<i>b</i>	{2, 3, 4}	<i>c</i>	{1, 3, 4}	<pre> graph TD     1 -- b --&gt; 2     2 -- c --&gt; 1     1 -- a --&gt; 3     2 -- a --&gt; 4     3 -- a --&gt; 5 </pre>
	<i>dom</i>								
<i>a</i>	{1, 2, 4}								
<i>b</i>	{2, 3, 4}								
<i>c</i>	{1, 3, 4}								

Figure 4.3: Example related to the construction for Lemma 4.17

The new set of processes  $Proc'$  is a subset of  $Proc$  and is obtained by

$$Proc' := \bigcup_{q_1, q_2 \in Q, a \in \Sigma} \theta(q_1, q_2, a) \cup \bigcup_{a, b \in \Sigma} \eta(a, b),$$

where the elements involved in the above formula are:

$$\theta(q_1, q_2, a) := \begin{cases} \emptyset, & \text{if } q_1 \equiv_{dom(a)} q_2 \\ \{p\}, & \text{for a } p \in dom(a) \text{ such that } q_1 \not\equiv_p q_2 \text{ (so, } q_1 \not\equiv_{dom(a)} q_2) \end{cases}$$

and

$$\eta(a, b) := \begin{cases} \emptyset, & \text{if } a \parallel b \\ \{p\}, & \text{for a } p \in dom(a) \cap dom(b) \text{ (so, } a \not\parallel b). \end{cases}$$

The domains  $dom'$  of the actions for the new distributions are obtained by intersecting the original domains with the new set of processes, i.e., for all  $a \in \Sigma$ ,

$$dom'(a) := dom(a) \cap Proc'.$$

Since the size of  $Proc'$  is at most  $|Q|^2|\Sigma| + |\Sigma|^2$ , the size of the new distribution is polynomially bounded by the size of the specification. Also, it is easy to see that the independence relation generated by the *new* distribution is exactly  $\parallel$ : The processes introduced by  $\eta(a, b)$  serve precisely this purpose.

We show that the deterministic transition system  $TS$  is isomorphic to an asynchronous automaton over the new distribution using of course Theorem 4.19. That is, we show that the least family of equivalences  $(\equiv'_p)_{p \in Proc'}$  satisfying  $DAA_1$  and  $DAA_2$ , also satisfies  $DAA_3$  and  $DAA_4$ . The turning point of the proof is based on the following implication holding for all  $q_1, q_2 \in \Sigma, a \in \Sigma$ :

$$\text{if } q_1 \equiv'_{dom'(a)} q_2, \text{ then } q_1 \equiv_{dom(a)} q_2. \quad (*)$$

Assuming for the moment that the above implication  $(*)$  is true, let us prove that  $DAA_3$  and  $DAA_4$  hold for  $(\equiv'_p)_{p \in Proc'}$ :

**DAA<sub>3</sub> satisfaction** : Let  $q_1, q_2$  be two states such that  $q_1 \equiv'_{Proc'} q_2$ . Then, we have that  $q_1 \equiv'_{dom'(a)} q_2$  for all  $a \in \Sigma$ . Applying  $(*)$ , we have that  $q_1 \equiv_{dom(a)} q_2$  for all  $a \in \Sigma$



Algorithm 4.3: Construction of the least family of equivalences satisfying the  $\text{DAA}_1$  and  $\text{DAA}_2$  rules (of Theorem 4.19)

<b>Input</b>	a distribution $(\Sigma, Proc', \Delta')$ and a <i>deterministic</i> transition system $TS = (Q, \Sigma, \rightarrow, \{q^{in}\})$
Step 0	For each $p \in Proc'$ , initialize the binary relation $\equiv_p^0 \subseteq Q \times Q$ to $\emptyset$ and $i$ to 0.
Step 1	For each $q \xrightarrow{a} q'$ in $TS$ , set $q \equiv_p^0 q'$ for every $p \in Proc' \setminus dom'(a)$ . ( $\text{DAA}_1$ ) Then, close each $\equiv_p^0$ under reflexivity, symmetry, and transitivity for $p \in Proc'$ .
Step 2	Initialize $(\equiv_p^{i+1})_{p \in Proc'}$ to $(\equiv_p^i)_{p \in Proc'}$ .
Step 3	If there exist $q_1 \xrightarrow{a} q'_1$ and $q_2 \xrightarrow{a} q'_2$ such that $q_1 \equiv_{dom'(a)}^i q_2$ , but not $q'_1 \equiv_{dom'(a)}^i q'_2$ , then <ol style="list-style-type: none"> <li>3.1 for each <math>p \in dom'(a)</math>, set <math>q'_1 \equiv_p^{i+1} q'_2</math>,</li> <li>3.2 close each <math>\equiv_p^{i+1}</math> under symmetry and transitivity for <math>p \in dom'(a)</math>,</li> <li>3.3 increment <math>i</math> by 1, and go back to Step 2.</li> </ol> Otherwise, output $(\equiv_p^i)_{p \in Proc'}$ .
<b>Output</b>	the least set of local equivalences satisfying $\text{DAA}_1$ and $\text{DAA}_2$

and, since  $Proc = \bigcup_{a \in \Sigma} dom(a)$  (we supposed that each process can execute at least one action), we have that  $q_1 \equiv_{Proc} q_2$ . Applying  $\text{DAA}_3$  to  $(\equiv_p)_{p \in Proc}$ , we obtain that  $q_1 = q_2$ .

**DAA<sub>4</sub> satisfaction** : Let  $q_1, q'_1, q_2$  be three states and  $a$  an action such that  $q_1 \equiv'_{dom'(a)} q_2$  and  $q_1 \xrightarrow{a} q'_1$ . Using (\*), we have that  $q_1 \equiv_{dom(a)} q_2$ , which together with  $q_1 \xrightarrow{a} q'_1$  and the application of  $\text{DAA}_4$  to  $(\equiv_p)_{p \in Proc}$ , implies that there exists a state  $q'_2$  such that  $q_2 \xrightarrow{a} q'_2$ .

Our only duty now is to prove that the implication (\*) is indeed true, and we do this by induction on the number of times the condition  $\text{DAA}_2$  is used in the construction of  $(\equiv'_p)_{p \in Proc'}$ . The processes selected by  $\theta(q_1, q_2, a)$  will make this possible.

By construction,  $(\equiv'_p)_{p \in Proc'}$  is the least family of equivalences satisfying  $\text{DAA}_1$  and  $\text{DAA}_2$  and can be computed as the fixed point of an increasing sequence of sets of equivalences

$$(\equiv_p^0)_{p \in Proc'} \subseteq (\equiv_p^1)_{p \in Proc'} \subseteq \dots \subseteq (\equiv_p^i)_{p \in Proc'} = (\equiv_p^{i+1})_{p \in Proc'} =: (\equiv'_p)_{p \in Proc'},$$

where, by definition  $(\equiv_p^i)_{p \in Proc'} \subseteq (\equiv_p^j)_{p \in Proc'}$  if  $\equiv_p^i \subseteq \equiv_p^j$  for all  $p \in Proc'$ . The first  $(\equiv_p^0)_{p \in Proc'}$  is obtained using  $\text{DAA}_1$  for all the transitions of  $TS$  and then, iteratively,  $(\equiv_p^{i+1})_{p \in Proc'}$  is obtained from  $(\equiv_p^i)_{p \in Proc'}$  by applying *once* the  $\text{DAA}_2$  rule. The details are given by Algorithm 4.3. It is immediate to see that the algorithm terminates and is correct (i.e., the set of equivalences at the output is the least set of equivalences to satisfy  $\text{DAA}_1$  and  $\text{DAA}_2$ ).



In order to prove that implication (\*) holds, we will prove by *induction* on  $i \geq 0$  that for all  $q_1, q_2 \in Q$  and  $p \in Proc'$ ,

$$\text{if } q_1 \equiv_p^i q_2, \text{ then } q_1 \equiv_p q_2. \quad (**)$$

and for all  $a \in \Sigma$ ,

$$\text{if } q_1 \equiv_{dom'(a)}^i q_2, \text{ then } q_1 \equiv_{dom(a)} q_2. \quad (***)$$

(Note that (\*\*) and (\*\*\*) are independent.)

Finally, (\*\*\*) implies (\*), because after reaching a fixed point in Algorithm 4.3, the sets  $(\equiv_p^i)_{p \in Proc'}$  and  $(\equiv_p)_{p \in Proc'}$  are equal.

We provide now the proof by *simultaneous* induction for (\*\*) and (\*\*\*) .

*Base case :*

Proof for (\*\*) : We suppose that  $q_1 \equiv_p^0 q_2$  and show that  $q_1 \equiv_p q_2$  (for  $p \in Proc'$ ).

Looking at the construction of  $\equiv_p^0$  done in Steps 0 and 1 of Algorithm 4.3, if  $q_1 \equiv_p^0 q_2$  holds, there must exist the states  $s_0, s_1, \dots, s_m$  with  $m \geq 0$  together with the actions  $b_1, \dots, b_m$  such that  $q_1 = s_0, q_2 = s_m$ , and for all  $k \in [1..m]$ ,  $p \in Proc' \setminus dom'(b_k)$  and either  $s_{k-1} \xrightarrow{b_k} s_k$  or  $s_{k-1} \xleftarrow{b_k} s_k$  (such that we can get by Step 1 that  $s_{k-1} \equiv_p^0 s_k$  and further by transitivity that  $q_1 = s_0 \equiv_p^0 \dots \equiv_p^0 s_m = q_2$ ). Now, since  $p \in Proc' \setminus dom'(b_k)$  and  $Proc' \setminus dom'(b_k) = Proc' \setminus (dom(b_k) \cap Proc') = Proc' \setminus dom(b_k) \subseteq Proc \setminus dom(b_k)$ , we have that  $p \in Proc \setminus dom(b_k)$  for each  $k \in [1..m]$ . Moreover, since by hypothesis  $(\equiv_p)_{p \in Proc}$  satisfies DAA<sub>1</sub>, we have that  $p \in Proc \setminus dom(b_k)$  and  $s_{k-1} \xrightarrow{b_k} s_k$  (or  $s_{k-1} \xleftarrow{b_k} s_k$ ) implies that  $s_{k-1} \equiv_p s_k$ , again for all  $k \in [1..m]$ . Finally, this gives (by the transitivity of  $\equiv_p$ ) that  $q_1 = s_0 \equiv_p \dots \equiv_p s_m = q_2$ , so  $q_1 \equiv_p q_2$ .

Proof for (\*\*\*) : We suppose that  $q_1 \equiv_{dom'(a)}^0 q_2$  and show that  $q_1 \equiv_{dom(a)} q_2$ .

By contradiction, assume that  $q_1 \not\equiv_{dom(a)} q_2$ . Then, by the construction of  $\theta(q_1, q_2, a)$ , there exists a process  $p \in dom(a)$  such that  $\theta(q_1, q_2, a) = \{p\}$  and

$$q_1 \not\equiv_p q_2. \quad (4.1)$$

Since  $p \in \theta(q_1, q_2, a) \subseteq Proc'$ , we have that  $p \in dom'(a)$  (because by definition  $dom'(a) = dom(a) \cap Proc'$ ). Thus, from  $p \in dom'(a)$  and  $q_1 \equiv_{dom'(a)}^0 q_2$ , we have that  $q_1 \equiv_p^0 q_2$ . This implies, using (\*\*) proved above, that  $q_1 \equiv_p q_2$  and we end up contradicting (4.1).

*Induction step :* We suppose that (\*\*) and (\*\*\*) hold for  $i \geq 0$  and show that (\*\*) and (\*\*\*) hold also for  $i + 1$ .

Proof for (\*\*) : We must show that  $q_1 \equiv_p^{i+1} q_2$  implies  $q_1 \equiv_p q_2$ .

First, if  $q_1 \equiv_p^i q_2$ , then by the induction hypothesis we have indeed that  $q_1 \equiv_p q_2$ . Otherwise, if  $q_1 \not\equiv_p^i q_2$ , then Step 3.1 of Algorithm 4.3 was necessarily involved

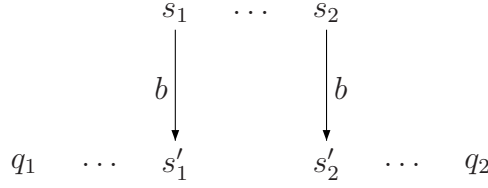


Figure 4.4: Visual aid for the inductive proof of Lemma 4.17

in the computation of  $\equiv_p^{i+1}$  (in order to obtain that  $q_1 \equiv_p^{i+1} q_2$ ). This implies the existence of two transitions  $s_1 \xrightarrow{b} s'_1$  and  $s_2 \xrightarrow{b} s'_2$  such that

$$s_1 \equiv_{dom'(b)}^i s_2, \quad (4.2)$$

$s'_1 \not\equiv_{dom'(b)}^i s'_2$ , and  $p \in dom'(b)$ . Moreover, we have that

$$q_1 \equiv_p^{i+1} s'_1 \equiv_p^{i+1} s'_2 \equiv_p^{i+1} q_2 \text{ and } q_1 \equiv_p^i s'_1 \not\equiv_p^i s'_2 \equiv_p^i q_2$$

(see also Figure 4.4). Using (\*\*\*) for  $i$  (induction hypothesis), from the equivalences  $q_1 \equiv_p^i s'_1$ , respectively  $s'_2 \equiv_p^i q_2$  above, we obtain that  $q_1 \equiv_p s'_1$ , respectively  $s'_2 \equiv_p q_2$ . Once we prove that also  $s'_1 \equiv_p s'_2$ , we are done (we apply the transitivity of  $\equiv_p$  and obtain indeed that  $q_1 \equiv_p q_2$ ).

But  $s'_1 \equiv_p s'_2$  is now easy to prove: From (4.2), applying the induction hypothesis for  $i$  and (\*\*\*), we have that  $s_1 \equiv_{dom(b)} s_2$ , which together with the existence of  $s_1 \xrightarrow{b} s'_1$  and  $s_2 \xrightarrow{b} s'_2$ , and the fact that  $(\equiv_p)_{p \in Proc}$  satisfies  $DAA_2$ , further implies that  $s'_1 \equiv_{dom(b)} s'_2$ . Since  $p \in dom'(b) \subseteq dom(b)$ , we obtain that  $s'_1 \equiv_p s'_2$ .

Proof for (\*\*\*) : We assume that  $q_1 \equiv_{dom'(a)}^{i+1} q_2$  and show that  $q_1 \equiv_{dom(a)} q_2$ .

First, if  $q_1 \equiv_{dom'(a)}^i q_2$ , then by the induction hypothesis we have indeed that  $q_1 \equiv_{dom(a)} q_2$ . Otherwise, if  $q_1 \not\equiv_{dom'(a)}^i q_2$ , then Step 3.1 of Algorithm 4.3 was necessarily involved in the computation of  $(\equiv_p^{i+1})_{p \in Proc'}$  (in order to obtain that  $q_1 \equiv_{dom'(a)}^{i+1} q_2$ ). This means that there exist two transitions  $s_1 \xrightarrow{b} s'_1$  and  $s_2 \xrightarrow{b} s'_2$  such that

$$s_1 \equiv_{dom'(b)}^i s_2 \quad (4.3)$$

and  $s'_1 \not\equiv_{dom'(b)}^i s'_2$ . Moreover, we necessarily have that

$$q_1 \equiv_{dom'(a)}^{i+1} s'_1 \equiv_{dom'(a)}^{i+1} s'_2 \equiv_{dom'(a)}^{i+1} q_2 \quad (4.4)$$

and

$$q_1 \equiv_{dom'(a)}^i s'_1 \not\equiv_{dom'(a)}^i s'_2 \equiv_{dom'(a)}^i q_2 \quad (4.5)$$

(see also Figure 4.4). Using (\*\*\*) for  $i$  (induction hypothesis), from the equivalences  $q_1 \equiv_{dom'(a)}^i s'_1$ , respectively  $s'_2 \equiv_{dom'(a)}^i q_2$ , of (4.5), we obtain that  $q_1 \equiv_{dom(a)} s'_1$ , respectively  $s'_2 \equiv_{dom(a)} q_2$ . Once we prove that also  $s'_1 \equiv_{dom(a)} s'_2$ ,

we are done (we apply the transitivity of  $(\equiv_p)_{p \in Proc}$  and obtain indeed that  $q_1 \equiv_{dom(a)} q_2$ ).

By contradiction, assume that  $s'_1 \not\equiv_{dom(a)} s'_2$ . Then, by the construction of  $\theta(s'_1, s'_2, a)$ , there exists a process  $p \in dom(a)$  such that  $\theta(s'_1, s'_2, a) = \{p\}$  and

$$s'_1 \not\equiv_p s'_2. \quad (4.6)$$

Since  $p \in \theta(s'_1, s'_2, a) \subseteq Proc'$ , we have that  $p \in dom'(a)$  (because by definition  $dom'(a) = dom(a) \cap Proc'$ ). From (4.6), applying the induction hypothesis for  $i$  and (\*\*), we have that  $s'_1 \not\equiv_p^i s'_2$  (otherwise,  $s'_1 \equiv_p^i s'_2$  implies  $s'_1 \equiv_p s'_2$ ). On the other hand,  $s'_1 \equiv_p^{i+1} s'_2$  (because  $s'_1 \equiv_{dom'(a)}^{i+1} s'_2$  – cf. (4.4) – and  $p \in dom'(a)$ ). Hence, we have that  $s'_1 \not\equiv_p^i s'_2$  and  $s'_1 \equiv_p^{i+1} s'_2$ , so necessarily  $s'_1 \equiv_p^{i+1} s'_2$  was obtained by Step 3.1 (applied to the transitions  $s_1 \xrightarrow{b} s'_1$  and  $s_2 \xrightarrow{b} s'_2$ ), which means that  $p \in dom'(b)$ . Thus, from  $p \in dom'(b)$  and  $dom'(b) = dom(b) \cap Proc' \subseteq dom(b)$ , we have that

$$p \in dom(b). \quad (4.7)$$

Finally, from (4.3) and the induction hypothesis saying that (\*\*\*) holds for  $i$ , we obtain that  $s_1 \equiv_{dom(b)} s_2$ , which together with the existence of the transitions  $s_1 \xrightarrow{b} s'_1$  and  $s_2 \xrightarrow{b} s'_2$ , and the fact that the family  $(\equiv_p)_{p \in Proc}$  satisfies the DAA<sub>2</sub> condition, implies that  $s'_1 \equiv_{dom(b)} s'_2$ . This last fact together with (4.7), shows that  $s'_1 \equiv_p s'_2$  and we end up contradicting (4.6).  $\square$

## 4.3 Implementability modulo Language Equivalence

This section presents complexity results for the test whether there exists a distributed transition system exhibiting the same behavior as a given transition system.

### 4.3.1 Synchronous Products of Transition Systems

Section 3.4.3 provides characterizations for the languages accepted by various classes of synchronous products of transition systems (see also Figures 3.13–3.16). The central ingredient there is the notion of *product language* (Definition 3.40), which is a language with the property that it is equal to the synchronization of its projections on the sets of local alphabets (Proposition 3.43). Thus, by Theorem 3.50, the class of the languages of *deterministic* synchronous products, denoted by  $\mathcal{L}(DSP)$ , is equal to the class of prefix-closed regular product languages, whereas for *nondeterministic* synchronous products,  $\mathcal{L}(NSP)$  is the closure under finite unions of the class  $\mathcal{L}(DSP)$ . However, according to Proposition 3.52, the class of *nondeterministic* synchronous products with *local initial states* (i.e., the set of global initial states is the cartesian product of local sets of initial states) is equal to  $\mathcal{L}(DSP)$ . In particular, the class of *nondeterministic* synchronous products with *only one initial state* is also equal to  $\mathcal{L}(DSP)$  (cf. Corollary 3.53).

In this section we study the implementability problem modulo language equivalence for synchronous products with *one initial state* (i.e.,  $|I| = 1$  in Definition 3.15) and we will consider the general case (i.e., multiple global initial states) at the end of the next section in order to profit from some of the proof techniques introduced there. Therefore we consider first the following problem:

**Problem 4.20** *Given a distribution  $(\Sigma, Proc, \Delta)$  and a transition system  $TS$  over  $\Sigma$ , does there exist a synchronous product of transition systems over  $\Delta$  with only one initial state<sup>1</sup> that is language equivalent to  $TS$ ?*

According to Corollary 3.53, a language  $L \subseteq \Sigma^*$  is accepted by a synchronous product with one initial state if and only if  $L$  is accepted by a deterministic synchronous product, and this happens (according to Theorem 3.50) if and only if  $L$  is a prefix-closed regular product language. If we assume  $L$  to be the language of a transition system, then  $L$  is a prefix-closed regular language (Corollary 2.15), therefore solving Problem 4.20 boils down to checking whether  $L$  is a product language. Thus, according to Proposition 3.43, we have to check whether  $L$  is equal to the synchronization of its projections  $L \upharpoonright_{\Sigma_{loc}(p)}$  on the local alphabets  $\Sigma_{loc}(p)$  associated with the processes. Since it is easy to see that  $L$  is always included in the synchronization of its projections, we only have to check the other inclusion.

Given a distribution  $(\Sigma, Proc, \Delta)$  and a transition system  $TS$ , Algorithm 4.4 presents, following [Muk02], a decision procedure for the test whether  $L(TS)$  is a product language. According to the above discussion, Algorithm 4.4 also decides Problem 4.20.

We are now ready to present the results advertised in column 3 of Table 4.1. For this, we need to introduce first the reachability problem for synchronous products, that will be used in a subsequent reduction:

**Problem 4.21 (Reachability in synchronous products)**

INSTANCE: *Given a distribution  $(\Sigma, Proc, \Delta)$ , a set of local transition systems  $(TS_p)_{p \in Proc}$  with  $TS_p = (Q_p, \Sigma_{loc}(p), \rightarrow_p, \{q_p^{in}\})$ , and a global state  $q \in \prod_{p \in Proc} Q_p$ ,*

QUESTION: *is the global state  $q$  reachable from the global initial state  $(q_p^{in})_{p \in Proc}$  via the global synchronization on common actions of the  $\rightarrow_p$ 's as in Definition 3.15?*

*The complement of the reachability problem for synchronous products (that is, checking whether the global state  $q$  above is not reachable) is called the non-reachability problem for synchronous products.*

**Lemma 4.22** *The non-reachability problem for synchronous products can be in polynomial time reduced to Problem 4.20.*

<sup>1</sup>Since the class of synchronous products with one initial state shares the characterization of the accepted languages with the deterministic synchronous products and the synchronous products with local initial states, we could have posed the problem with either of the models, but we chose the synchronous products with one initial state for simplicity. Nonetheless, the complexity results for synchronous products with one initial state obtained in this section **apply equally** to deterministic synchronous products, respectively synchronous products with local initial states.

Table 4.8: Details for the satisfaction of the  $AA_2$  property (only cases not solved already by Table 4.7)

$q_1$	$q_2$	Why $AA_2$ holds (a process $p \in Proc$ is given)
$q'_{x_i}$	$\{q_{x_i}\}$	$p := p_{x_i c_0}$ (by construction $x_i \in c_0$ ).
	$Q' \setminus \{q'_{x_i}, q_{x_i}\}$	$p := p_{x_{i'} c_0}$ , where $i' \neq i$ (by construction $x_{i'} \in c_0$ ).
$q_{x_i}$	$Q' \setminus \{q_{x_i}, q'_{x_i}\}$	$p := p_{x_{i'} c_0}$ , where $i' \neq i$ (by construction $x_{i'} \in c_0$ ).
$q^F_{x_i}$	$\{q^T_{x_i}\}$	$p := p_{x_i c_0}$ (by construction $x_i \in c_0$ ).
	$Q' \setminus \{q^F_{x_i}, q^T_{x_i}\}$	$p := p_{x_{i'} c_0}$ , where $i' \neq i$ (by construction $x_{i'} \in c_0$ ).
$q^T_{x_i}$	$Q' \setminus \{q^T_{x_i}, q^F_{x_i}\}$	$p := p_{x_{i'} c_0}$ , where $i' \neq i$ (by construction $x_{i'} \in c_0$ ).
$q^0_{c_j}$	$Q' \setminus \{q^0_{c_j}\}$	$p := p_{\ell c_j}$ , where $\ell$ is a literal of $c_j$ .

Algorithm 4.4: A test whether the language of a transition system is a product language

<b>Instance</b>	A distribution $(\Sigma, Proc, \Delta)$ and a transition system $TS = (Q, \Sigma, \rightarrow, I)$
<b>Question</b>	Is $L(TS)$ a product language over $\Delta$ ?
Step 0	<p>W.l.o.g. suppose that <math>TS</math> has only one initial state. Otherwise,</p> <ul style="list-style-type: none"> <li>▪ add a new state <math>q^{in}</math> and <math>\varepsilon</math>-transitions from <math>q^{in}</math> to the initial states of <math>I</math>,</li> <li>▪ apply an <math>\varepsilon</math>-closure<sup>a</sup> to the new transition system, and</li> <li>▪ set <math>q^{in}</math> as the only initial state, i.e., <math>I := \{q^{in}\}</math>.</li> </ul>
Step 1	Let $TS = (Q, \Sigma, \rightarrow, \{q^{in}\})$ . For each process $p \in Proc$ , construct a projection $TS_p := (Q, \Sigma_{loc}(p), \rightarrow_p, \{q^{in}\})$ obtained from a copy of $TS$ in which the labels from $\Sigma \setminus \Sigma_{loc}(p)$ are replaced by $\varepsilon$ and $\rightarrow_p$ is the $\varepsilon$ -closure <sup>a</sup> of $\rightarrow$ .
Step 2	If the language of the synchronous product over $\Delta$ of the transition systems $(TS_p)_{p \in Proc}$ with one global initial state $(q^{in}, \dots, q^{in})$ is included in the language of $TS$ , then answer “yes”, otherwise answer “no”.

<sup>a</sup>A *polynomial-time* algorithm for  $\varepsilon$ -closure (i.e., the elimination of the transitions labeled by  $\varepsilon$  such that the obtained transition system is language equivalent to the original one) can be found in [HU79, Chapter 2.4].

**Proof.** According to Problem 4.21, we are given a distribution  $(\Sigma, Proc, \Delta)$ , a local transition system  $TS_p = (Q_p, \Sigma_{loc}(p), \rightarrow_p, \{q_p^{in}\})$  for each  $p \in Proc$  and a global state  $q \in \prod_{p \in Proc} Q_p$ . Moreover, we suppose for convenience that  $Proc := \{1, \dots, n\}$ .

We construct a distribution  $(\Sigma', Proc', \Delta')$  and a transition system  $R$  over  $\Sigma'$  such that:

Problem 4.20 has a solution for  $\Delta'$  and  $R$  if and only if the global state  $q := (q_1, \dots, q_n)$  is *not* reachable from the global initial state  $(q_1^{in}, \dots, q_n^{in})$ .

The new distribution  $(\Sigma', Proc', \Delta')$  is chosen as follows:

- $\Sigma' := \Sigma \cup \{a_p \mid p \in [1..n]\} \cup \{\checkmark\}$ .  
I.e., we add a new action for each process  $p$  plus an extra one. (Note that  $\Sigma = \bigcup_{p \in [1..n]} \Sigma_{loc}(p)$ .)
- $Proc' := Proc \cup \{p_0\}$ , and
- $\Delta' \subseteq \Sigma' \times Proc'$  is given by the local alphabets  $\Sigma'_{loc}(p)$  as follows:
  - $\Sigma'_{loc}(p) := \Sigma_{loc}(p) \cup \{a_{p'} \mid p' \in [1..n] \wedge p' \neq p\} \cup \{\checkmark\}$  for every  $p \in [1..n]$  and
  - $\Sigma'_{loc}(p_0) := \Sigma' \setminus \{\checkmark\}$ .

This gives the following domains  $dom'$  for the actions of  $\Sigma'$ :

- $dom'(a) = dom(a) \cup \{p_0\}$ , for all  $a \in \Sigma$   
(where  $dom(a)$  is given by  $\Delta$ ),
- $dom'(a_p) = Proc' \setminus \{p\}$ , for all  $p \in [1..n]$ , and
- $dom'(\checkmark) = Proc' \setminus \{p_0\} = Proc$ .

The transition system  $R := (Q', \Sigma', \rightarrow, \{q_0\})$  is sketched in Figure 4.5 and defined as:

- $Q' := \{q_0, q'_0\} \cup \bigcup_{p \in [1..n]} Q_p \cup \{q'_p \mid p \in [1..n]\}$   
(w.l.o.g., we assume that  $Q_p \cap Q_{p'} = \emptyset$  for  $p \neq p'$ ),
- $\rightarrow := \{q_0 \xrightarrow{a} q'_0 \mid a \in \Sigma\} \cup \{q'_0 \xrightarrow{a} q'_0 \mid a \in \Sigma\} \cup \bigcup_{p \in [1..n]} \left( \{q_0 \xrightarrow{a_p} q_p^{in}\} \cup \rightarrow_p \cup \{q_p \xrightarrow{\checkmark} q'_p\} \right)$ .

**Remark 4.23** As previously mentioned, Problem 4.20 is decided by Algorithm 4.4. Thus, following Step 1 of the procedure, we construct the projections  $R_p$  of  $R$  onto the local alphabets  $\Sigma'_{loc}(p)$  as follows:

- For  $p \in [1..n]$ ,  $R_p$  is obtained from  $R$  replacing the labels from  $\Sigma' \setminus \Sigma'_{loc}(p)$  by  $\varepsilon$  and applying an  $\varepsilon$ -closure. Since  $a_p \notin \Sigma'_{loc}(p)$  and  $\Sigma_{loc}(p) \cup \{\checkmark\} \subseteq \Sigma'_{loc}(p)$ , we have that all the states reachable by a run  $v$  from  $q_p^{in}$  in  $TS_p$  (with  $v \in (\Sigma_{loc}(p))^*$ ) can also be reached by *the same* run  $v$  from  $q_0$  in  $R_p$ .
- For  $p := p_0$ , since  $\Sigma'_{loc}(p_0) = \Sigma' \setminus \{\checkmark\}$ , the projection  $R_{p_0}$  is just  $R$  without the  $\checkmark$ -labeled transitions.

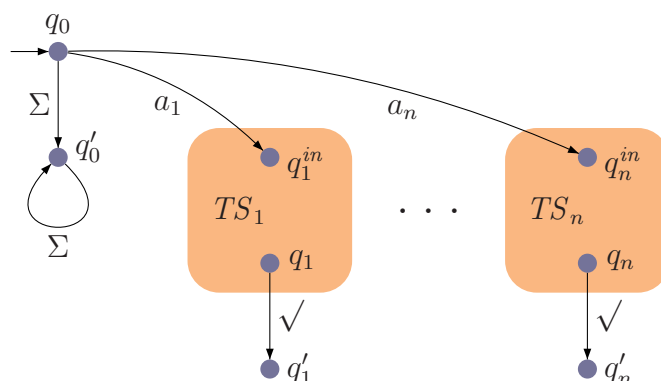


Figure 4.5: A schematic representation of the reduction in the proof of Lemma 4.22

**First Implication.** We assume that  $L(R)$  is a product language over  $\Delta'$  and we prove that the global state  $q := (q_1, \dots, q_n)$  is not reachable from  $q^{in} := (q_1^{in}, \dots, q_n^{in})$  in the synchronous product of the  $TS_p$ 's over  $\Delta$ .

By contradiction, assume that there exists a run  $w \in \Sigma^*$  such that  $q$  is reachable from  $q^{in}$  after executing the sequence  $w$  of actions. Then, we show that the language of the synchronous product over  $\Delta'$  of the projections  $R_p$  is not included in the language of  $R$ . Thus, according to Step 2 of Algorithm 4.4, we obtain a negative answer and this contradicts the assumption that  $L(R)$  is a product language.

On one hand, the run  $w\sqrt{\notin} L(R)$  because  $w \in \Sigma^*$  and all the runs of  $R$  containing  $\sqrt{\phantom{x}}$  start with an  $a_p$  action and  $a_p \notin \Sigma$ , for any  $p \in [1..n]$ .

On the other hand, we can show that  $w\sqrt{\phantom{x}}$  is a run of the synchronization of the  $R_p$ 's. Informally, we will simulate the synchronizations of the  $TS_p$ 's on  $w \in \Sigma^*$  by synchronizations of  $R_p$ 's and at the end we will also have a synchronization of the local transitions  $q_p \xrightarrow{\sqrt{\phantom{x}}} q'_p$ :

In the synchronous product of the  $TS_p$ 's, we can execute  $w$  from  $q^{in}$  and we reach  $q$  (we assumed so by contradiction). According to Definition 3.15, the synchronization on each  $a \in \Sigma$  involves only the processes of  $dom(a)$ . When synchronizing the projections  $R_p$  on  $a \in \Sigma$ , we must observe  $dom'(a) = dom(a) \cup \{p_0\}$ . For  $p := p_0$ , we can always execute  $a \in \Sigma$  from  $q_0 \xrightarrow{a} q'_0 \xrightarrow{a} q'_0$  which is part of  $R_{p_0}$ . For  $p \in dom(a)$ , we can move in  $R_p$  (starting in  $q_0$ ) similarly to moving in  $TS_p$  (starting in  $q_p^{in}$ ) according to Remark 4.23. In this way, we are able to execute  $w$  in the synchronous product of the  $R_p$ 's starting from the global state  $(q_0, \dots, q_0)$  and to reach the state  $q_p$  in  $R_p$  for each  $p \in [1..n]$ . Since  $dom'(\sqrt{\phantom{x}}) = Proc = [1..n]$  and we have  $q_p \xrightarrow{\sqrt{\phantom{x}}} q'_p$  in each  $p \in [1..n]$ , we can finally have a  $\sqrt{\phantom{x}}$ -synchronization. Therefore, the run  $w\sqrt{\phantom{x}}$  belongs to the synchronization of the  $R_p$ 's over  $\Delta'$ .

**Second Implication.** We assume that  $q$  is not reachable from  $q^{in}$  in the synchronization of the  $TS_p$ 's, and we prove that the language of the synchronous product over  $\Delta'$  of the projections  $R_p$  is included in the language of  $R$ . Therefore, for each run  $v \in (\Sigma')^*$  of the synchronization of the  $R_p$ 's, we show that  $v \in L(R)$  as well.



Let  $v$  be a run of the synchronization of the  $R_p$ 's. Then, depending whether  $\surd$  appears or not in the run  $v$ , it is easy to see that  $v$  can only have two forms and in both cases,  $v$  will also belong to  $L(R)$ :

$v \in (\Sigma' \setminus \{\surd\})^*$  : From  $\Sigma'_{loc}(p_0) = \Sigma' \setminus \{\surd\}$ , we necessarily have that  $v \in L(R_{p_0})$ . Then, with the help of Remark 4.23,  $L(R_{p_0}) \subseteq L(R)$ , so  $v \in L(R)$ .

$v = w\surd$  with  $w \in (\Sigma' \setminus \{\surd\})^*$  : Again, from  $\Sigma'_{loc}(p_0) = \Sigma' \setminus \{\surd\}$ , we necessarily have that  $w \in L(R_{p_0})$ . Looking at  $R_{p_0}$ ,  $w$  can only have two forms (depending on the first action of  $w$ ):

$w \in \Sigma^*$  : We show that this is not possible, given the fact that  $w\surd$  is a run of the synchronization of the  $R_p$ 's. Since the action  $\surd$  can be executed only if all  $R_p$ 's with  $p \in \text{dom}'(\surd) = [1..n]$  will execute a  $\surd$ -labeled transition, we have that no  $R_p$  with  $p \in [1..n]$  will ever synchronize on a  $q_0 \xrightarrow{a} q'_0$  transition for  $a \in \Sigma$ , because no run from  $q'_0$  can contain  $\surd$ . This means that the synchronization of the  $R_p$ 's on  $w \in \Sigma^*$  simulates a synchronization of the  $TS_p$ 's on  $w$ . From the hypothesis,  $q = (q_1, \dots, q_n)$  is not reachable, so no  $\surd$ -synchronization will be possible after executing  $w$ .

$w = a_i u$  with  $i \in [1..n]$  and  $u \in L(TS_i)$  : Since the first action of  $w$  (and  $v$ ) is  $a_i$  and  $\text{dom}'(a_i) = \text{Proc}' \setminus \{i\}$ , all  $R_p$ 's *except*  $R_i$  must execute their local  $q_0 \xrightarrow{a_i} q_p^{in}$  transition (this transition belongs to  $R_p$  for  $p \neq i$ , because in this case  $a_i \in \Sigma'_{loc}(p)$ ). Then, the  $R_p$ 's must synchronize on  $u$  such that at the end also a  $\surd$ -synchronization is possible. Since  $u \in L(TS_i)$ , we have that  $u \in (\Sigma'_{loc}(i))^*$  and also  $u\surd \in (\Sigma'_{loc}(i))^*$ . That means that  $R_i$  will take part in all synchronizations on  $u\surd$  starting from  $q_0$  and the only possibility for  $R_i$  to do this is by  $q_0 \xrightarrow{u} q_i \xrightarrow{\surd} q'_i$ . This necessarily implies a run  $q_i^{in} \xrightarrow{u} q_i$  in  $TS_i$  (because  $\Sigma_{loc}(i) \subseteq \Sigma'_{loc}(i)$ ), which further implies a run  $q_0 \xrightarrow{a_i} q_i^{in} \xrightarrow{u} q_i \xrightarrow{\surd} q'_i$  in  $R$ , so  $v = a_i u \surd$  belongs indeed to  $L(R)$ .  $\square$

Based on Lemma 4.22, we are now able to prove the complexity results we are after. In this process, we will also use complexity results from [SHRS96] for checking non-reachability and language inclusion for synchronous products.

**Theorem 4.24** *The implementability problem modulo language equivalence for synchronous products of transition systems with  $|I| = 1$  is PSPACE-complete.*

**Proof.** In Lemma 4.22 we have shown how the non-reachability problem for synchronous products can be in polynomial time reduced to the problem of deciding the implementability of a *single* transition system as a language equivalent synchronous product of transition systems. Since the non-reachability problem for synchronous products is proved in [SHRS96, Theorem 3.10] to be PSPACE-hard, we immediately deduce the PSPACE-hardness of our problem.

Furthermore, according to Step 2 of Algorithm 4.4, in order to decide our problem, it is enough to check whether the language of the input transition system  $TS$  includes the language of the synchronization of its projections  $TS_p$ . But this test can be done in PSPACE as proved by [SHRS96, Theorem 3.12], so our problem is in PSPACE.  $\square$



**Proposition 4.25** *The implementability problem for synchronous products with  $|I| = 1$  modulo language equivalence remains PSPACE-complete, when the input transition system  $TS$  is deterministic.*

*For acyclic specifications (i.e.,  $TS$  is acyclic) the problem is coNP-complete, and it remains so even for deterministic specifications.*

**Proof.** The PSPACE-hardness proof of [SHRS96, Theorem 3.10] works in fact for *deterministic*  $TS_p$ 's. Moreover, the reduction of Lemma 4.22 constructs a deterministic input transition system  $R$  if the components  $TS_p$ 's are all deterministic (see Figure 4.5). Therefore, our problem is PSPACE-hard also for *deterministic* specifications. The PSPACE-completeness follows directly from the PSPACE-completeness of Theorem 4.24.

Let us consider now the second part of the proposition, supposing that the input  $TS$  is acyclic. Then, we want to use the coNP-hardness of the non-reachability problem for synchronous products of *acyclic* and *deterministic* transition systems from [SHRS96, Theorem 3.16]. For this, we modify the construction of  $R$  in the proof of Lemma 4.22 by replacing the loops  $\{q'_0 \xrightarrow{a} q'_0 \mid a \in \Sigma\}$  by a set of new transitions

$$\bigcup_{j \in [0..M]} \{s_j \xrightarrow{a} s_{j+1} \mid a \in \Sigma\},$$

where  $M := \max\{|w| \mid w \in L(TS_p), p \in [1..n]\}^1$  (this maximum exists because all the  $TS_p$ 's are acyclic when  $TS$  is assumed acyclic),  $s_0 = q'_0$ , and each  $s_j$  is a new state (for  $j \in [1..M+1]$ ). In this way  $R$  is acyclic if all the  $TS_p$ 's are acyclic and it is easy to see that the reduction is still correct. Therefore, our problem for deterministic and acyclic specifications is coNP-hard. The coNP-completeness follows from [SHRS96, Theorem 3.17], which easily proves that checking language inclusion for synchronous products of acyclic transition systems can be solved in coNP.  $\square$

### 4.3.2 Asynchronous Automata

Section 3.4.4 provides characterizations for the languages accepted by various classes of asynchronous automata (see also Figures 3.13–3.16). The central ingredient there is the notion of *independence* of actions and the trace, respectively forward, closure of a language (cf. Section 3.4.2). Thus, by Theorem 3.62, the class of the languages of (deterministic) asynchronous automata is equal to the class of prefix-closed regular (forward-closed) trace-closed languages.

Using the above mentioned characterizations, we obtain complexity bounds for the implementability problem for asynchronous automata modulo languages equivalence similar to the bounds for synchronous products:

**Theorem 4.26** *The implementability problem modulo language equivalence for asynchronous automata is PSPACE-complete.*

**Proof.** According to Theorem 3.62, a language  $L$  is accepted by an asynchronous automaton if and only if  $L$  is a prefix-closed regular trace-closed language. In the implementability problem modulo language equivalence, the specification is given as the language of

<sup>1</sup> $M$  is the maximum on all the lengths of the runs of the (acyclic) local transition systems  $TS_p$ .

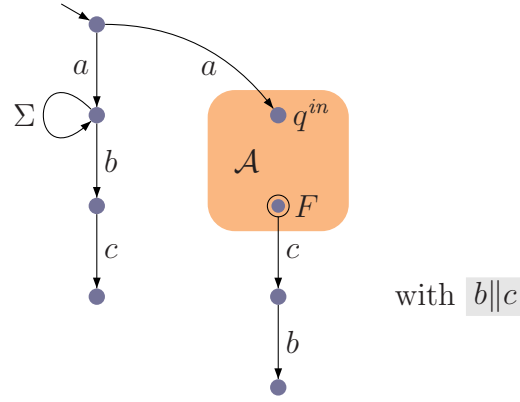


Figure 4.6: A schematic representation of the reduction in the proof of Theorem 4.26

a transition system, which is already a prefix-closed regular language, so we only have to check whether the given language is trace-closed. But this test can be done in PSPACE, as shown by [PWW98, Theorem 8 with Corollary 10] which proved that checking whether the language of a finite automaton (seen, according to Definition 2.13, as a transition system with final states) is a trace language can be done in PSPACE. Hence, our problem is in PSPACE.

To show PSPACE-hardness, we use a simple reduction from the *totality problem* ‘ $= \Sigma^*$ ?’ for nondeterministic finite automata, known to be PSPACE-hard [GJ79], which is defined as: ‘Given a nondeterministic finite automaton  $\mathcal{A} = (Q, \Sigma, \rightarrow, I, F)$  over the alphabet  $\Sigma$ , is  $L(\mathcal{A}, F) = \Sigma^*$ ?’ In fact, we can assume w.l.o.g. that  $\mathcal{A}$  has only one initial state,  $I := \{q^{in}\}$  (just follow Step 0 of Algorithm 4.4, with the extra care to make  $q^{in}$  also final if any of the original initial states were final, i.e.,  $\varepsilon \in L(\mathcal{A}, F)$ ).

We reduce the above totality problem to the implementability problem for asynchronous automata modulo language equivalence. Thus, for each nondeterministic finite automaton  $\mathcal{A} = (Q, \Sigma, \rightarrow, \{q^{in}\}, F)$ , we build a distribution  $(\Sigma', Proc, \Delta)$  and a transition system  $TS$  over  $\Sigma'$  such that:  $L(\mathcal{A}, F) = \Sigma^*$  if and only if there exists an asynchronous automaton  $\mathcal{AA}$  over  $\Delta$  such that  $L(\mathcal{AA}) = L(TS)$ . According to Theorem 3.62 and the fact that  $L(TS)$  is a prefix-closed regular language, the  $\mathcal{AA}$  above exists if and only if  $L(TS)$  is trace-closed, so it is enough to prove the following equivalence:

$$L(\mathcal{A}, F) = \Sigma^* \text{ if and only if } L(TS) \text{ is trace-closed. } (\#)$$

We choose  $\Sigma' := \Sigma \cup \{a, b, c\}$  with  $a, b, c \notin \Sigma$ ,  $Proc := \{1, 2\}$ , and  $\Delta$  such that  $\Sigma_{loc}(1) := \Sigma \cup \{a, b\}$  and  $\Sigma_{loc}(2) := \Sigma \cup \{a, c\}$ . We can see that  $b||c$  is the only independence generated by  $\Delta$ .

Then, we choose the transition system  $TS := (Q', \Sigma', \rightarrow', \{q_0\})$  as in Figure 4.6. More precisely, we have:

- $Q' := Q \cup \{q_0, q_1, q_2, q_3, q'_2, q'_3\}$  and
- $\rightarrow' := \{q_0 \xrightarrow{a} q_1\} \cup \{q_1 \xrightarrow{\alpha} q_1 \mid \alpha \in \Sigma\} \cup \{q_1 \xrightarrow{b} q_2 \xrightarrow{c} q_3\} \cup \{q_0 \xrightarrow{a} q^{in}\} \cup (\rightarrow) \cup \{q^{fin} \xrightarrow{c} q'_2 \xrightarrow{b} q'_3 \mid q^{fin} \in F\}$ .

Then, it is easy to see that

$$L(TS) = \text{Prefix}(a\Sigma^*bc + aL(\mathcal{A}, F)cb).$$

Now we can prove ( $\sharp$ ): If we assume that  $L(\mathcal{A}, F) = \Sigma^*$ , then  $L(TS) = \text{Prefix}(a\Sigma^*(bc+cb))$ , which is obviously a trace-closed language (the only independence is  $b||c$ ). On the other hand, if  $L(\mathcal{A}, F) \neq \Sigma^*$ , then there exists  $w \in \Sigma^* \setminus L(\mathcal{A}, F)$ , which implies that  $L(TS)$  is not a trace-closed language because  $awbc \in L(TS)$  and  $b||c$ , but  $awcb \notin L(TS)$ .

In the end, we mention that the above proof is reminiscent of the PSPACE-hardness proof for checking the trace closure of the language of a nondeterministic I-diamond Büchi automaton [Mus94, Theorem 7.2.3].

**Alternative proof** (for the PSPACE-hardness part). An alternative proof for the PSPACE-hardness of the implementability problem can be obtained modifying the PSPACE-hardness proof of checking the trace closure of the language of a nondeterministic automaton [PWW98, Theorem 11]. The difference is that the language of a transition system is always prefix-closed, so the constructions must accommodate this detail and this is not immediate. We give this alternative proof below. (In fact, we hoped that the proof presented below could be modified to provide a lower bound for the implementability problem modulo bisimulation in the general case (i.e., when the distributed implementation is not required to be deterministic) which is an open problem [CMT99, Muk02], but until now we have not managed to do so.)

We will use a reduction from the ‘Linear Space Acceptance’ problem, known to be PSPACE-hard [GJ79], which is: ‘Given a deterministic linear bounded Turing machine  $M$  and a finite word  $x$  over the (input) alphabet of  $M$ , does  $M$  accept  $x$ ?’

For each deterministic linear bounded Turing machine  $M$  and finite word  $x$ , we build a distribution  $(\Sigma, Proc, \Delta)$  and a transition system  $TS$  over  $\Sigma$  such that:  $x \notin L(M)$  if and only if  $L(TS)$  is a prefix-closed regular trace-closed language. The PSPACE-hardness of our problem is then obtained from the PSPACE-hardness of the ‘Linear Space Acceptance’ problem [GJ79], the fact that PSPACE=coPSPACE [Pap94], and Theorem 3.62.

In fact, instead of giving a transition system  $TS$ , we will give a regular expression  $R_{M,x}$  such that

$$x \notin L(M) \text{ if and only if } L(R_{M,x}) \text{ is a prefix-closed regular trace-closed language} \quad (\natural)$$

(this is enough according to Corollary 2.15 and using the fact that translating a regular expression into a transition system accepting the same language can be done in polynomial time [HMU01, Section 3.2.3]). More precisely, we construct  $R_{M,x}$  such that

- if  $x \notin L(M)$ , then  $L(R_{M,x}) = \Sigma^*$ , and
- if  $x \in L(M)$ , then  $L(R_{M,x}) = \Sigma^* \setminus (u\Sigma^*)$ ,  
where  $u := u_1 \dots u_k$  is the accepting computation of  $M$  on  $x$  having its first two letters  $u_1, u_2$  independent,  $u_1 || u_2$ .<sup>1</sup>

<sup>1</sup>A similar idea was used in the proof of [PWW98, Theorem 9] (showing that it is PSPACE-hard to check whether the language of a nondeterministic finite automaton is trace-closed). In our case though we must take extra care that the language  $L_{M,x}$  is prefix-closed (cf. the  $\Sigma^*$  after  $u$  in the second item) and this detail adds difficulty to the proof.

For  $R_{M,x}$  satisfying the above properties it is easy to see that the equivalence (†) holds: If  $x \notin L(M)$ , then  $L(R_{M,x}) = \Sigma^*$ , which is obviously a prefix-closed regular trace-closed language, whereas if  $x \in L(M)$ , then  $L(R_{M,x}) = \Sigma^* \setminus (u_1 u_2 u_3 \dots u_k \Sigma^*)$ , which is not trace-closed because  $u_1 \parallel u_2$  and  $u_2 u_1 u_3 \dots u_k \in L(R_{M,x})$ , but  $u_1 u_2 u_3 \dots u_k \notin L(R_{M,x})$ .

We give now the construction for  $R_{M,x}$ : Let us fix an  $n$  space-bounded Turing machine  $M := \langle Q, \Gamma, \Gamma_0, q_0, q_F, \delta \rangle$ , where  $Q$  is the finite state space,  $\Gamma$  the tape alphabet,  $\Gamma_0 \subseteq \Gamma$  the input alphabet,  $q_0$  the initial state,  $q_F$  the final state such that  $q_F \neq q_0$ , and  $\delta$  the transition function. We also fix an input word of length  $n$ ,  $x := x_1 \dots x_n \in \Gamma_0^*$ .

We write configurations of  $M$  in the usual way as strings over  $(Q \times \Gamma) \cup \Gamma$ ; e.g., the initial configuration is  $[q_0 x_1] x_2 \dots x_n$ . Since  $M$  is  $n$ -bounded, any configuration will have length  $n$ . A computation of  $M$  on  $x$  will be represented as a string over the alphabet  $\Sigma := (Q \times \Gamma) \cup \Gamma \cup \{\$\}$  of the form  $\$c_0\$c_1\$ \dots \$c_r \dots$ , where the  $c_i$ 's are successive configurations of  $M$ , and  $c_0 := [q_0 x_1] x_2 \dots x_n$  is the initial configuration. An *accepting* computation of  $M$  is a computation of the form  $\$c_0\$c_1\$ \dots \$c_{r-1}\$c_r$  which contains the final state  $q_F$  (inside a tuple  $[q_F y] \in \{q_F\} \times \Gamma$ ) and the final state  $q_F$  occurs *only* in the last configuration  $c_r$ . For technical reasons, we also define a *truncated accepting computation* as the prefix  $u$  of an accepting computation, in which  $q_F$  appears in the last letter (i.e., a  $[q_F y]$  is on the last position).

Now, we choose the distribution  $(\Sigma, Proc, \Delta)$  where  $\Sigma := (Q \times \Gamma) \cup \Gamma \cup \{\$\}$ ,  $Proc := \{1, 2\}$ , and  $dom(\$) := \{1\}$ ,  $dom([q_0 x_1]) := \{2\}$ , and  $dom(c) := \{1, 2\}$ , for all  $c \in \Sigma \setminus \{\$, [q_0 x_1]\}$ . We see that the only independent letters are the first two letters of any computation, i.e.,  $\$ \parallel [q_0 x_1]$ . The language  $L(R_{M,x})$  that we are looking for will contain all the strings that are *not* of the form: a truncated accepting computation  $u$  followed by any sequence from  $\Sigma^*$ . (To do this, we will modify the classical PSPACE-hardness proof for the totality problem for regular expressions (see for instance, [HU79, Theorem 13.15] and [MS72]) in order to accommodate the prefix and trace closure properties required in (†).)

We choose  $R_{M,x}$  to be the regular expression of the form

$$R_{M,x} := R_{\text{wrong\_start}} + R_{\text{wrong\_end}} + R_{\text{wrong\_move}},$$

whose three constituent parts are defined below. (For  $S := \{s_1, \dots, s_m\} \subseteq \Sigma$  a subset of actions, we also denote by  $S$  the regular expression  $s_1 + \dots + s_m$ .)

1. Since all the computations of  $M$  on  $x$  start with  $\$[q_0 x_1] x_2 \dots x_n \$$ , we choose

$$R_{\text{wrong\_start}} := (\Sigma - \$)\Sigma^* + \$(\Sigma - [q_0 x_1])\Sigma^* + \$[q_0 x_1](\Sigma - x_2)\Sigma^* + \dots + \$[q_0 x_1] x_2 \dots x_n (\Sigma - \$)\Sigma^*.$$

2. Let  $\Omega_{q_F} := \Sigma \setminus \{[q_F y] \mid y \in \Gamma\}$  be the set of actions not containing  $q_F$ . Then,

$$R_{\text{wrong\_end}} := \Omega_{q_F}^*.$$

3.  $R_{\text{wrong\_move}}$  is expressed in terms of a function which describes local correctness of moves of  $M$ . Each sequence of three adjacent letters  $a_1 a_2 a_3 \in \Sigma^3$  in a *valid*

computation, uniquely determines the letter, call it  $f(a_1a_2a_3)$ , appearing  $n + 1$  symbols to the right of  $a_2$ . Then, we choose

$$R_{\text{wrong\_move}} := \sum_{a_1a_2a_3 \in \Omega_{q_F}^3} (\Omega_{q_F}^* a_1a_2a_3 \Omega_{q_F}^n (\Sigma - f(a_1a_2a_3)) \Sigma^*).$$

We draw the attention to the use of  $\Omega_{q_F}$  in the above formula, which generates incorrect moves only until the final state occurs (this should unveil the reason why we worked with truncated accepting computations).

It is not difficult to show that: If  $x \notin L(M)$ , then  $L(R_{M,x}) = \Sigma^*$ , otherwise  $L(R_{M,x}) = \Sigma^* \setminus u\Sigma^*$  where  $u$  is the truncated accepted computation of  $M$  on input  $x$ .  $\square$

We provide now complexity bounds for specifications of special form. Interesting enough, for deterministic specifications, the implementability problem for asynchronous automata is much cheaper than for synchronous products:

**Proposition 4.27** *The implementability problem for asynchronous automata modulo language equivalence for deterministic specifications is decidable in polynomial time.*

*For nondeterministic acyclic specifications, the problem is coNP-complete.*

**Proof.** The first part follows directly from Theorem 3.62 and [PWW98, Theorem 7] proving that it is decidable in polynomial time whether the language of a *deterministic* finite automaton is a trace-closed language. (Note that the restriction regarding the determinism applies only to the specification, while the distributed implementation may as well be nondeterministic.) The fact that it is decidable in polynomial time whether the language of a deterministic transition system is trace-closed can be explained without appealing to the literature: We simply minimize in polynomial time the deterministic input transition system (cf. Corollary 2.19) and check locally (in polynomial time) if the ID rule holds. This is correct according to Proposition 3.39.

For the second part dealing with acyclic specifications (i.e.,  $TS$  is acyclic), we show first that the problem is in coNP. Using Corollary 3.64, in order to decide the implementability problem for acyclic specification, it is enough to verify whether the language of the *acyclic* input transition system  $TS$  is trace-closed. To prove that the problem can be solved in coNP, we show that its negation can be solved in NP. But the fact that the language of an acyclic transition system is *not* trace-closed can be decided by the following *nondeterministic* algorithm in polynomial time: Given a distribution and a transition system  $TS$ , a machine can guess a pair of independent actions  $a||b$  and two strings  $w, w'$  and check in polynomial time whether  $wabw' \in L(TS)$  and  $wbaw' \notin L(TS)$  (notice that, since  $TS$  is acyclic, the lengths of  $w$  and  $w'$  are bounded by the size of  $TS$ ).

For the coNP-hardness part, we use a reduction from the problem of ‘language inequivalence for acyclic finite nondeterministic automata’, which is: ‘Given two *acyclic* nondeterministic finite automata  $\mathcal{A}_1$  and  $\mathcal{A}_2$  over the same alphabet  $\Sigma$ , do they accept different languages?’ This problem is known to be NP-complete [GJ79].

The proof will be similar to the (PSPACE-hardness) proof of Theorem 4.26. Given  $\mathcal{A}_1 = (Q_1, \Sigma, \rightarrow_1, \{q_1^{in}\}, F_1)$  and  $\mathcal{A}_2 = (Q_2, \Sigma, \rightarrow_2, \{q_2^{in}\}, F_2)$  two acyclic nondeterministic

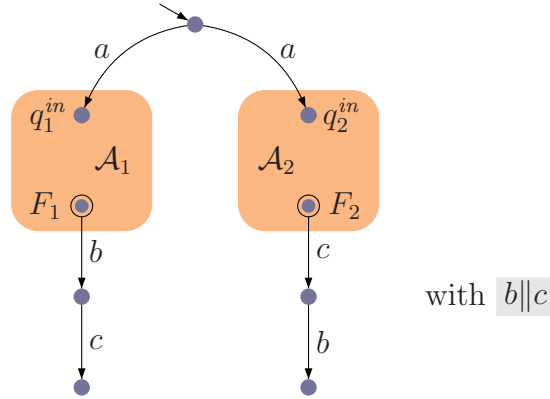


Figure 4.7: A schematic representation of the reduction in the proof of Proposition 4.27

finite automata assumed w.l.o.g. to have only one initial state, we construct a (nondeterministic acyclic) transition system  $TS$  as in Figure 4.7 and a distribution such that  $b||c$ . We have then:

$$L(TS) = \text{Prefix}(a L(\mathcal{A}_1, F_1) bc + a L(\mathcal{A}_2, F_2) cb),$$

which easily implies that  $L(TS)$  is a *not* a trace language if and only if the languages  $L(\mathcal{A}_1, F_1)$  and  $L(\mathcal{A}_2, F_2)$  are different.  $\square$

**Synchronous products with multiple global initial states** Based on the observation that the language accepted by a synchronous product is necessarily a trace-closed language (Corollary 3.35), we can recycle the constructions in Figures 4.6 and 4.7 to derive complexity bounds for synchronous products with *multiple* global initial states (this is a debt from Section 4.3.2).

**Theorem 4.28** *The implementability problem for synchronous products of transition systems (with  $|I| \geq 1$ ) modulo language equivalence is PSPACE-hard.*

*For nondeterministic acyclic specifications, the problem is coNP-complete, whereas for the deterministic acyclic ones is in P.*

**Proof.** To prove the PSPACE-hardness of the implementability problem for synchronous products of transition systems with  $|I| \geq 1$  modulo language equivalence, we use the construction of  $TS$  and the distribution  $(\Sigma', Proc, \Delta)$  from the (first) PSPACE-hardness proof of Theorem 4.26 (see Figure 4.6). More precisely, we show that  $L(\mathcal{A}, F) = \Sigma^*$  if and only if  $L(TS)$  is the language of a synchronous product over  $\Delta$ : If  $L(\mathcal{A}, F) = \Sigma^*$ , then  $L(TS) = \text{Prefix}(a\Sigma^*(bc + cb))$  and we can easily construct a synchronous product accepting  $L(TS)$  (we simply choose two local transition systems accepting the projections  $\text{Prefix}(a\Sigma^*b)$ , respectively  $\text{Prefix}(a\Sigma^*c)$ ). If  $L(\mathcal{A}, F) \neq \Sigma^*$ , then  $L(TS)$  is not a trace-closed language and therefore cannot be the language of a synchronous product (because the language of a synchronous product of transition systems is always trace-closed according to Corollary 3.35).

Let us consider now the case of acyclic specifications (accepting finite languages). We know that class of *finite* languages accepted by synchronous products is equal to the class



Table 4.9: Complexity results for the implementability of synchronous products of transition systems with multiple initial states ( $|I| \geq 1$ )

Specification ( $TS$ )	Isomorphism	Language Equivalence
Nondeterministic	NP-complete	PSPACE-hard
Deterministic	P [Mor98]	?
Acyclic & Nondet.	NP-complete	coNP-complete
Acyclic & Determ.	P [Mor98]	P

of *finite* languages accepted by asynchronous automata, viz. the class of prefix-closed trace-closed finite languages [Zie87, Theorem 5.1] (see also Figure 3.14 and the footnote of Proposition 3.58). Therefore, the complexity of the implementability problem for non-deterministic (respectively, deterministic) acyclic specifications for synchronous products is settled by the coNP-completeness (respectively, in P) result of the same problem for asynchronous automata (Proposition 4.27).

As a remark, the above construction *cannot* be used for the proof of Proposition 4.25 treating the case of *deterministic* specifications<sup>1</sup>, because the constructions of Figures 4.6 and 4.7 are fundamentally nondeterministic.  $\square$

We gather the results for synchronous products with multiple initial states in Table 4.9 (complementing thus Table 4.1). The complexity for the implementability modulo isomorphism is already solved by Theorem 4.3 (column 2 of Table 4.9), while the implementability for bisimulation is open in the general case (we do not extend the table for it; for *deterministic* synchronous products results are given in column 4 of Table 4.1). The results for implementability problem modulo language equivalence from Theorem 4.28 above are given in column 3 of Table 4.9. In the general case we have only a lower bound, while an upper bound is still to be found. Moreover, we do not know anything about its complexity when the specification is deterministic.

**Remark 4.29** In fact, we suspect that in the general case (i.e., nondeterministic specifications) the synthesis problem for synchronous product with multiple initial state may be even undecidable. The difficulty of the problem can be guessed from the characterization of the class of languages accepted by synchronous products as the class of finite unions of prefix-closed regular product languages (cf. Theorem 3.50). Thus, given a transition system  $TS$ , one should test whether there exists a set of prefix-closed regular languages  $L_1, \dots, L_m$  such that  $L = L_1 \cup \dots \cup L_m$  and each  $L_i$  is a product language, for  $i \in [1..m]$ . Since  $L$  may be infinite, there may be an *infinite* number of ‘decompositions’ of  $L$  as  $L_1 \cup \dots \cup L_m$  for which we should test whether each  $L_i$  is a product language.

Recent results [BBL05] confirm indeed that the problem at hand is difficult to tackle. More precisely, [BBL05] is dedicated to the decidability of the question if a regular language is a finite union of product languages. Using non-trivial techniques it was shown that in the particular case where the distribution has only *two* processes, given a fixed

<sup>1</sup>As we will see in Section 4.4, this deterministic case is important to settle the implementability problem modulo bisimulation for deterministic implementations.

Algorithm 4.5: A test whether the language of a regular expression is prefix-closed

<b>Instance</b>	A regular expression $E$ over a set of actions $\Sigma$
<b>Question</b>	Is $L(E)$ prefix-closed?
Step 1	Construct in polynomial time a nondeterministic finite automaton $\mathcal{A} = (Q, \Sigma, \rightarrow, I, F)$ such that $L(\mathcal{A}, F) = L(E)$ (see [HMU01, Section 3.2.3] for details).
Step 2	Define a new set of final states $F'$ as the set of all states that are on a path from an initial state $q^{in} \in I$ to a final one $q^{fin} \in F$ .
Step 3	If $L(\mathcal{A}, F) = L(\mathcal{A}, F')$ , then answer “yes”, otherwise answer “no”.

number  $k$ , it is decidable whether a regular language  $L$  is equal to the union of  $k$  product languages. The problem is still open for general distributions as well as for unrestricted number of product languages in the union.

**Regular expressions as specifications** When defining synthesis/implementability problem modulo language equivalence, one can give the regular specification as a *regular expression* (cf. page 15) instead of a transition system. Unlike the language of a transition system, the language of a regular expression is not necessarily prefix-closed. However the prefix-closure property can be verified in PSPACE:

**Lemma 4.30** *Given a regular expression  $E$ , checking whether the language  $L(E)$  is prefix-closed can be done in PSPACE.*

**Proof.** A PSPACE test for prefix-closure is given by Algorithm 4.5. The algorithm obviously terminates and belongs to PSPACE because Steps 1 and 2 need only polynomial time, while checking the language equivalence of two nondeterministic finite automata (Step 3) can be determined in PSPACE [GJ79]. The proof of correctness of the algorithm (i.e.,  $L(E)$  is prefix-closed if and only if  $L(\mathcal{A}, F) = L(\mathcal{A}, F')$ ) is the same as the second part of the proof of Corollary 2.15.  $\square$

We will show now that the complexity bounds from Theorems 4.26 and 4.24 are preserved for specifications given as regular expressions:

**Theorem 4.31** *The implementability problem modulo language equivalence for asynchronous automata (respectively, for synchronous products with  $|I| = 1$ ) with regular expressions as specifications is PSPACE-complete.*

**Proof.** Given a distribution  $(\Sigma, Proc, \Delta)$  and a regular expression  $E$  over  $\Sigma$ , we check whether there exists an asynchronous automaton (respectively, synchronous product with one initial state) over  $\Delta$  accepting  $L(E)$ . According to the characterizations of Section 3.4,



we have to check whether  $L(E)$  is a prefix-closed regular trace-closed (respectively, product) language.

Following Steps 1 and 2 of Algorithm 4.5, we can construct  $\mathcal{A}$ ,  $F$  and  $F'$  and test  $L(\mathcal{A}, F) = L(\mathcal{A}, F')$  to check if  $L(E)$  is prefix-closed. If  $L(E)$  is not prefix-closed, we know that the implementability problem has no solutions. Otherwise, we can construct a transition system  $TS$  obtained from  $\mathcal{A}$  preserving only the states of  $F'$ , so  $L(TS) = L(\mathcal{A}, F') = L(\mathcal{A}) = L(E)$ . Then, we can check in polynomial space starting with  $TS$  as specification if  $L(TS)$  is a trace-closed (respectively, product) language as done in the proofs of Theorem 4.26 (respectively, 4.24). From all the above we have that our problem is in PSPACE.

For the PSPACE-hardness part we use a reduction from the ‘totality problem for regular expressions’, i.e., ‘Given a regular expression  $E$  over a set  $\Sigma$  of actions, does  $L(E) = \Sigma^*$  hold?’. This problem is known to be PSPACE-complete [GJ79]. The reduction is an adaptation of the (first) PSPACE-hardness proof of Theorem 4.26 (see also Figure 4.6)<sup>1</sup>: Given a regular expression  $E$  over  $\Sigma$ , we construct a new regular expression  $E'$  over  $\Sigma' := \Sigma \cup \{a, b, c\}$  with  $b||c$  as follows:

$$E' := \varepsilon + a + a\Sigma^* + a\Sigma^*b + a\Sigma^*bc + aE + aEc + aEcb.$$

Mimicking the proof of Theorem 4.26 (respectively, 4.28), one can easily show that  $L(E) = \Sigma^*$  if and only if  $L(E')$  is a prefix-closed regular trace-closed (respectively, product<sup>2</sup>) language.  $\square$

### 4.3.3 Non-regular specifications

In this section we make a short detour and examine what happens when the specifications are not regular. The following result suggests that there is no hope to test the implementability once we move higher in Chomsky’s hierarchy:

**Theorem 4.32** *Checking that a context-free language is trace-closed is undecidable.*

**Proof.** The proof uses the fact that the set of invalid computations of a Turing machine is a context-free language [HU79, Lemma 8.7], together with the trick of making the first two letters of an accepting computation independent (see the technique in the alternative proof of Theorem 4.26).

We use a reduction from the ‘emptiness problem for Turing machines’, i.e., ‘Given a Turing machine  $M$ , is  $L(M)$  empty?’, which is a classical undecidable problem [Pap94]. Thus, given an arbitrary Turing machine  $M = (Q, \Gamma, \Gamma_0, \delta, q_0, B, F)$ , we construct a distribution and a context-free grammar  $G$  such that

$$L(M) = \emptyset \text{ if and only if } L(G) \text{ is a trace-closed language.}$$

<sup>1</sup>We cannot directly reduce (in polynomial time) the implementability problem modulo language equivalence with *transition systems* as specifications to the corresponding problem with *regular expressions* as specifications, because of the potential exponential blowup in the general translation from a transition system to a regular expression accepting the same language [HMU01, Section 3.2.1].

<sup>2</sup>For this part, we use the fact that every product language is trace-closed (Proposition 3.41).

We give first the definition of a *valid computation* of  $M$ , which is a string

$$\$ w_1 \$ w_2^R \$ w_3 \$ w_4^R \$ \dots$$

over the alphabet  $Q \cup \Gamma \cup \{\$\}$  (with  $Q$  and  $\Gamma$  disjoint and  $\$$  belonging to none of them) such that:

1. each  $w_i$  is an instantaneous description (ID) of  $M$ , that is, a string in  $\Gamma^* Q \Gamma^*$  not ending with  $B$  (where  $B$  is the blank symbol),
2.  $w_1$  is an initial ID, that is, one of the form  $q_0 x$  for  $x \in \Gamma_0^*$ ,
3.  $w_n$  is a final ID, that is, one in  $\Gamma^* F \Gamma^*$ , and
4.  $w_i \vdash_M w_{i+1}$  for  $1 \leq i < n$ , that is,  $w_i$  and  $w_{i+1}$  are consecutive IDs.

Let  $\Sigma := Q \cup \Gamma \cup \{\$\}$ . We choose a context-free grammar  $G$  over  $\Sigma$  able to compute exactly *all the invalid computations* of  $M$ . Such a grammar  $G$  exists according to the construction of [HU79, Lemma 8.7]. Then, we have then that:

- if  $L(M) = \emptyset$ , then  $L(G) = \Sigma^*$  and
- if  $L(M) \neq \emptyset$ , then  $L(G) = \Sigma^* \setminus \{\text{the set of all valid computations of } M\}$ .

The trick now is to choose a distribution over  $\Sigma$  such that  $\$$  and  $q_0$  are independent ( $\$ \parallel q_0$ ). We show that  $L(M) = \emptyset$  if and only if  $L(G)$  is a trace-closed language. If  $L(M) = \emptyset$ , it is obvious that  $L(G) = \Sigma^*$  is trace-closed. If  $L(M) \neq \emptyset$ , then there is a valid computation of  $M$ , say  $w$ , which will not belong to  $L(G)$ . On one hand, by definition,  $w$  has the form  $\$ q_0 w'$ . On the other hand,  $q_0 \$ w'$  is an invalid computation (any valid computation starts with  $\$$ ) and therefore belongs to  $L(G)$ . So, we have that  $q_0$  and  $\$$  are independent and  $q_0 \$ w' \in L(G)$ , but  $\$ q_0 w' \notin L(G)$ , which means that  $L(G)$  is not a trace-closed language.  $\square$

Since the context-free grammars (and the language equivalent model of pushdown automata) are used to describe programs with procedures (which need stacks for their callings), a possible interpretation of the above result in a programming language setting can be: It is undecidable to decide if a global program *with procedures* (given as specification) admits a behaviorally equivalent *distributed* implementation.

## 4.4 Implementability modulo Bisimulation

In this section we present complexity bounds for the test whether there exists a distributed transition system bisimilar to a given transition system. The results will cover the implementability modulo bisimulation only for *deterministic* distributed implementations (nevertheless, the specification is still allowed to be nondeterministic). The general problem for nondeterministic implementations is still open.

Based on the observation that bisimulation (Definition 2.12) and language equivalence coincide for the class of *deterministic* transition systems [vG90]<sup>1</sup>, Mukund *et al.* [CMT99, Muk02] provide characterizations modulo bisimulation for the global transition systems of the *deterministic* distributed transition systems. More precisely, for a given transition system  $TS$  and a distribution  $\Delta$ , there exists a *deterministic* distributed transition system over  $\Delta$  bisimilar to  $TS$  if and only if the quotient  $TS/\sim_{TS}$  (with  $\sim_{TS}$  the largest bisimulation on  $TS$ , see page 20) is deterministic and  $L(TS)$  is the language of a deterministic distributed transition system.

We will combine the characterizations for the classes of languages of deterministic distributed transition systems (see Figure 3.13) with the above characterization for implementability modulo bisimulation and we will basically infer the same complexity results as for implementability with deterministic *specifications* from Propositions 4.25 and 4.27 (see column 4 of Tables 4.1 and 4.2).

#### 4.4.1 Synchronous Products of Transition Systems

The implementability problem modulo bisimulation is solved in the literature for *deterministic* synchronous products by:

**Theorem 4.33** [CMT99, Muk02] *Let  $(\Sigma, Proc, \Delta)$  be a distribution and  $TS$  a transition system over  $\Sigma$ . Then, the following are equivalent:*

- (i)  *$TS$  is bisimilar to a deterministic synchronous product of transition systems over  $\Delta$*
- (ii) *the bisimulation quotient  $TS/\sim_{TS}$  is deterministic and  $TS$  is language equivalent to a deterministic synchronous product over  $\Delta$ .*

The above theorem and some of our proofs make use of the following easy remarks regarding bisimulation:

**Lemma 4.34** [Folklore] *The following properties hold for two arbitrary transition systems  $TS, TS'$  over the same alphabet  $\Sigma$ :*

1. *If  $TS$  is deterministic, then  $TS/\sim_{TS}$  is also deterministic.*
2. *If  $L(TS) = L(TS')$  and  $TS$  and  $TS'$  are both deterministic, then  $TS$  and  $TS'$  are bisimilar.*
3. *If  $TS$  and  $TS'$  are bisimilar, then  $TS/\sim_{TS}$  and  $TS'/\sim_{TS'}$  are isomorphic, therefore  $L(TS) = L(TS')$ .*

We show now that the implementability problem modulo bisimulation is computationally intractable for deterministic synchronous products:

**Theorem 4.35** *The implementation problem for deterministic synchronous products of transition systems modulo bisimulation is PSPACE-complete.*

---

<sup>1</sup>In fact, all the equivalences between bisimulation and language equivalence (called *trace equivalence* in the context of [vG90]) coincide for the class of deterministic transition systems.

**Proof.** To show that the problem is in PSPACE, it is enough to check (ii) of Theorem 4.33. First, checking that  $TS/\sim_{TS}$  is deterministic takes polynomial time because constructing the largest bisimulation  $\sim_{TS}$  takes polynomial time [MS03]. Second, checking that  $TS$  is language equivalent to a deterministic synchronous product boils down to checking whether  $L(TS)$  is a product language (see Theorem 3.50). But checking that the language of a transition system is a product language can be done in PSPACE as testified by Algorithm 4.4 (cf. proof of Theorem 4.24).

The PSPACE-hardness proof is the same as the (PSPACE-hardness) proof of Proposition 4.25, that is, we use the reduction from the non-reachability problem for synchronous products of *deterministic* transition systems proved to be PSPACE-complete in [SHRS96, Theorem 3.10]. The reduction of Lemma 4.22 constructs a *deterministic* input transition system  $R$  if the components  $TS_p$ 's are all deterministic (see Figure 4.5). According to (ii) of Theorem 4.33, the only extra condition to be considered is that  $TS/\sim_{TS}$  is deterministic for the input transition system. But in our reduction, the input transition system is  $R$ , which is *deterministic* by construction, so, using 1. of Lemma 4.34, we have that  $R/\sim_R$  is indeed deterministic.  $\square$

**Proposition 4.36** *The implementability problem for deterministic synchronous products modulo bisimulation remains PSPACE-complete, when the input transition system  $TS$  is deterministic.*

*For acyclic specifications (i.e.,  $TS$  is acyclic) the problem is coNP-complete, and it remains so even for deterministic specifications.*

**Proof.** The proof for the first part is the same as the proof of Theorem 4.35 because the transition system  $R$  of the reduction there is already deterministic.

For acyclic specification, the proof is similar to the (coNP-completeness) proof of Proposition 4.25, again using 1. of Lemma 4.34 as used in the proof of Theorem 4.35.  $\square$

## 4.4.2 Asynchronous Automata

The characterization of *deterministic* asynchronous automata modulo bisimulation follows a similar pattern to Theorem 4.33 characterizing the deterministic synchronous products modulo bisimulation:

**Theorem 4.37** [Muk02] *Let  $(\Sigma, Proc, \Delta)$  be a distribution and  $TS$  a transition system over  $\Sigma$ . Then, the following are equivalent:*

- (i)  $TS$  is bisimilar to a deterministic asynchronous automaton over  $\Delta$
- (ii) the bisimulation quotient  $TS/\sim_{TS}$  is deterministic and  $TS$  is language equivalent to a deterministic asynchronous automaton over  $\Delta$ .

Opposed to synchronous products, we show now that the implementability problem modulo bisimulation is tractable for deterministic synchronous products:

**Theorem 4.38** *The implementation problem for deterministic asynchronous automata modulo bisimulation can be decided in polynomial time.*

Algorithm 4.6: A polynomial test if a transition system is bisimilar to a deterministic asynchronous automaton

<b>Instance</b>	A distribution $(\Sigma, Proc, \Delta)$ and a transition system $TS$
<b>Question</b>	Is $TS$ bisimilar to a <i>deterministic</i> asynchronous automaton over $\Delta$ ?
1:	construct $TS/\sim_{TS}$
2:	<b>if</b> $TS/\sim_{TS}$ is deterministic <b>then</b>
3:	<b>if</b> $L(TS/\sim_{TS})$ is a forward-closed trace-closed language <b>then</b>
4:	answer “yes”
5:	<b>else</b> answer “no”
6:	<b>else</b> answer “no”.

**Proof.** A polynomial test for our problem is given by Algorithm 4.6.

The algorithm is polynomial because the construction of  $TS/\sim_{TS}$  in line 1 is polynomial [MS03], the test in line 2 is obviously polynomial, and the test in line 3 is polynomial for transition systems that are deterministic, which is the case for  $TS/\sim_{TS}$ , because of the if-nesting. More precisely, testing if the language of a deterministic transition system  $TS$  is forward- and trace-closed can be done in polynomial time in the following way: We compute from  $TS$  in polynomial time the minimal transition system  $TS_0$  such that  $L(TS_0) = L(TS)$  (see Corollary 2.19 and [HU79, Section 3.4]) and test (locally – in polynomial time) if  $TS_0$  satisfies the ID and FD rules (this is enough according to Proposition 3.36).

The algorithm is correct according to Theorem 4.37, Theorem 3.62, and the language equality  $L(TS/\sim_{TS}) = L(TS)$  (obtained from the fact  $TS/\sim_{TS}$  is bisimilar to  $TS$  in conjunction with 3. of Lemma 4.34).  $\square$

## 4.5 Relaxed Implementability

In most cases the implementability problem (Problem 4.1) modulo isomorphism has no solutions (the constraints that the specification must satisfy in order to be distributable are very strong, cf. Theorems 4.2 and 4.10), therefore in practice we usually consider the implementability problem modulo language equivalence. But what can we do when even in the case of language equivalence we have no solutions? One possibility that we consider in this section is to relax the problem to allow (*under-*)*approximated* solutions, that is, the language of the distribution implementation *is included* in (rather than equal to) the language of the specification. We preferred the under-approximations to the over-approximations (i.e., the language of the implementation *includes* that of the specification) because in most of our cases studies (Sections 6.1.4 and 6.1.5), the specification is given using (regular) sets of *forbidden* sequences of actions (i.e., we specify safety properties) and

an implementation behaviorally richer than the specification may hit one of the forbidden sequences, which is of course undesirable.

In this section we focus on asynchronous automata, but we show that some results apply to synchronous products as well<sup>1</sup>. Thus, in Section 4.5.1 we show that the relaxed implementability problem is unfortunately undecidable for asynchronous automata and the proof can be adapted to show that a similar result holds for synchronous products. As partial solution to the undecidability issue, we propose in Section 4.5.2 a structural-based heuristic that is shown to be ‘only’ NP-complete. The application of this approach will be presented in the next chapter.

### 4.5.1 Language Inclusion

The relaxed implementability problem (for asynchronous automata) that we consider in this section is:

**Problem 4.39 (Relaxed implementability)**

INSTANCE: *Given a distribution  $(\Sigma, Proc, \Delta)$  and a transition system  $TS$  over  $\Sigma$  with  $\Sigma(TS) = \Sigma$ ,*

QUESTION: *does there exist an asynchronous automaton  $\mathcal{AA}$  over  $\Delta$  such that  $L(\mathcal{AA}) \subseteq L(TS)$  and  $\Sigma(\mathcal{AA}) = \Sigma$ ?*

In the relaxed version above, the implementability problem allows the distributed implementability to be behaviorally *included* in the specification. Since we want the implementations to be interesting, we additionally require that all the actions in the alphabet  $\Sigma$  appear also in the implementation, i.e.,  $\Sigma(\mathcal{AA}) = \Sigma$  (the definition of the alphabet  $\Sigma(TS)$  of a transition system is given on page 19, and by definition, the alphabet  $\Sigma(\mathcal{AA})$  of an asynchronous automaton  $\mathcal{AA}$  is the alphabet of its global transition system). We impose this (natural and not very strong) restriction in order to prevent the occurrence of trivial solutions like  $L(\mathcal{AA}) = \emptyset$  or partial solutions in which only some of the processes are executing actions.

Before proving the main result, we give a short lemma relating language inclusion and the trace-closure of a language (denoted by  $[L]$ , see definition on page 30).

**Lemma 4.40** *Let  $(\Sigma, ||)$  be a concurrent alphabet and  $S, B, F \subseteq \Sigma^*$  three languages. If  $S \subseteq B \setminus F$  and  $S$  is a trace-closed language, then  $S \subseteq B \setminus [F]$ .*

**Proof.** By contradiction, suppose there exists  $w \in S$  such that  $w \in [F]$ . Since  $w \in [F]$ , there exists  $w' \in F$  such that  $w \sim w'$ . On the other hand,  $w \in S$  and  $S$  is a trace-closed language, so  $w' \in S$ . We have then shown that  $w' \in S \subseteq B \setminus F$  and  $w' \in F$ , which is a contradiction. □

**Theorem 4.41** *The relaxed implementability problem for asynchronous automata is undecidable.*

---

<sup>1</sup>It seems that the characterizations of the languages of asynchronous automata involving trace-closed languages are easier to deal with theoretically than the product languages that the class of synchronous products generates.



**Proof.**<sup>1</sup> According to the characterization from Theorem 3.62, the class of the languages of asynchronous automata is equal to the class of prefix-closed regular trace-closed languages. Therefore, it is enough to prove that the following problem is undecidable (below we use the notion of alphabet of a language  $L$ , denoted by  $\Sigma(L)$ , as defined on page 13):

**Problem 4.42** *Given a concurrent alphabet  $(\Sigma, \parallel)$  and a prefix-closed regular language  $L$  such that  $\Sigma(L) = \Sigma$ , does there exist a prefix-closed regular trace-closed language  $L'$  such that  $L' \subseteq L$  and  $\Sigma(L') = \Sigma$ ?*

We prove that Problem 4.42 is undecidable using a reduction from Post's Correspondence Problem (PCP) which is known to be undecidable [Pap94]. (The difficulty of Problem 4.42 lies in the fact that the class of 'sub-languages'  $L'$  of a prefix-closed regular (infinite) language  $L$  may be *infinite* and we should in principle find a trace-closed one in this class. The undecidability result says that there is no finite classification of the sub-languages of  $L$  complying with our conditions and one should enumerate all in order to find a trace-closed one.)

Let  $A$  and  $B$  be two disjoint alphabets. An instance of Post's correspondence problem is encoded by two homomorphisms  $f, g : A^* \rightarrow B^*$ . A solution for the instance  $(f, g)$  is a word  $w \in A^+$  such that  $f(w) = g(w)$ . For technical reasons, we use two letters  $a$  and  $b$  to encode the sets  $A = \{a_1, \dots, a_{|A|}\}$  as  $\{ab, aab, \dots, a^{|A|}b\}$  and  $B = \{b_1, \dots, b_{|B|}\}$  as  $\{a^{|A|+1}b, a^{|A|+2}b, \dots, a^{|A|+|B|}b\}$ .

We choose the alphabet  $\Sigma = \{a, b, c, d\}$ . Also, we choose the following independence over  $\Sigma$ :

$$a \parallel c \text{ and } b \parallel c.$$

Consider now the following languages  $W_f$  and  $W_g$  (which has been used in the reduction given in [Sak92], see also [MP96]):

$$W_f = \{wf(w)c^{|f(w)|} \mid w \in A^+\} \quad \text{and} \quad W_g = \{wg(w)c^{|g(w)|} \mid w \in A^+\}.$$

Let us also denote for convenience  $\{a, b, c\}$  by  $\Sigma_0$ . Then, it can be proved (see the construction in [MP96]) that there exist the regular languages  $L_f$  and  $L_g$  such that their (trace-)closures with respect to  $\parallel$  are exactly the *complements* w.r.t.  $\Sigma_0$  of the closures of  $W_f$  and  $W_g$  respectively, i.e.,

$$[L_f] = \Sigma_0^* \setminus [W_f] \text{ and } [L_g] = \Sigma_0^* \setminus [W_g].$$

Now, given an instance  $(f, g)$  of the PCP, we choose the regular language

$$L_{fg} := \text{Prefix}((\Sigma_0^* \setminus (L_f \cup L_g))d)$$

and we prove that

the PCP instance  $(f, g)$  has a solution iff p Problem 4.42 with  $L := L_{fg}$  has a solution.

---

<sup>1</sup>This proof is the result of joint work with Javier Esparza and profited from discussions with Volker Diekert and Holger Petersen.

First of all, we prove that  $\Sigma(L_{fg}) = \Sigma$  (this is required in the formulation of Problem 4.42), showing that  $\Sigma(\Sigma_0^* \setminus (L_f \cup L_g)) = \Sigma_0$ . Since  $ba$  corresponds to no encoding,  $[bac] = \{bac, bca, cba\} \subseteq \Sigma_0^* \setminus [W_f]$ . From  $[L_f] = \Sigma_0^* \setminus [W_f]$ , we have that  $[bac] \subseteq [L_f]$ . We can choose  $L_f$  such that  $bac \in L_f$ , but  $cba \in [L_f] \setminus L_f$ . Similarly, we can choose  $L_g$  such that  $cba \in [L_g] \setminus L_g$ . Then, we can deduce that  $cba \in ([L_f] \setminus L_f) \cap ([L_g] \setminus L_g) \subseteq (\Sigma_0^* \setminus L_f) \cap (\Sigma_0^* \setminus L_g) = \Sigma_0^* \setminus (L_f \cup L_g)$ . Therefore  $\Sigma(\Sigma_0^* \setminus (L_f \cup L_g)) = \Sigma_0$  because  $\Sigma(cba) = \{a, b, c\} = \Sigma_0$ .

Suppose now that the PCP instance  $(f, g)$  has a solution, say  $w_0$ . Then,  $f(w_0) = g(w_0)$ . We prove that Problem 4.42 with  $L := L_{fg}$  has also a solution. More precisely, if we denote  $u_0 := w_0 f(w_0) c^{|f(w_0)|}$ , then  $L' := \text{Prefix}([u_0]d)$  is a solution, i.e.,  $L'$  is a prefix-closed regular trace-closed language such that  $L' \subseteq L_{fg}$  and  $\Sigma(L') = \Sigma$ .

First,  $L'$  is a prefix-closed language by construction and is regular because  $L'$  is finite.  $L'$  is a trace-closed language because  $[u_0]$  is a trace-closed language and both the concatenation with letter  $d$  (dependent of all the other actions!) and the prefix-closure operator preserve the closure under the independence relation (Proposition 3.7). Then,  $L' \subseteq L_{fg}$  because  $[u_0] \subseteq [W_f] \cap [W_g] = (\Sigma_0^* \setminus [L_f]) \cap (\Sigma_0^* \setminus [L_g]) \subseteq (\Sigma_0^* \setminus L_f) \cap (\Sigma_0^* \setminus L_g) = \Sigma_0^* \setminus (L_f \cup L_g)$ . Finally,  $\Sigma(L') = \Sigma$  because  $a$  and  $b$  appear in the encoding of any element of  $A$  and  $B$ ,  $c$  appears in  $u_0$ , and  $d$  appears in the construction of  $L'$ .

For the inverse implication, if Problem 4.42 with  $L := L_{fg}$  has a solution, we prove that also the PCP instance  $(f, g)$  has a solution: Suppose  $L'$  is a solution of Problem 4.42, i.e.,  $L'$  is a prefix-closed regular trace-closed language such that  $L' \subseteq L_{fg}$  and  $\Sigma(L') = \Sigma$ . Since  $d \in \Sigma(L')$ , there exists  $w \in L'$  such that  $d$  appears in  $w$ . From  $L' \subseteq L_{fg}$  and the definition of  $L_{fg}$ , necessarily  $w = vd$ , with  $v \in \Sigma_0^* \setminus (L_f \cup L_g)$ . Because  $L'$  is trace-closed,  $[w] = [v]d \subseteq L' \subseteq L_{fg}$  and we can deduce that  $[v] \subseteq \Sigma_0^* \setminus (L_f \cup L_g)$ . Now we can apply Lemma 4.40 (because  $[v]$  represents a trace-closed language) and obtain that  $[v] \subseteq \Sigma_c^* \setminus [L_f \cup L_g]$ . It is easy to see that  $\Sigma_0^* \setminus [L_f \cup L_g] = [W_f \cap W_g]$ , and so  $[v] \subseteq [W_f \cap W_g]$ . This implies  $W_f \cap W_g \neq \emptyset$ , which is equivalent to the PCP instance  $(f, g)$  having a solution.  $\square$

**Corollary 4.43** *The undecidability result of Theorem 4.41 holds also for the class of deterministic asynchronous automata. Moreover, it holds also for deterministic specifications.*

**Proof.** For the first part, according to the characterization from Theorem 3.62, the class of the languages of *deterministic* asynchronous automata is equal to the class of prefix-closed regular *forward-closed* trace-closed languages. The construction given in the proof of Theorem 4.41 can be reused with the only extra duty of showing that  $L'$  from the direct implication is a forward-closed language. ( $L' := \text{Prefix}([u_0]d)$ , where  $u_0 := w_0 f(w_0) c^{|f(w_0)|}$  and  $f(w_0) = g(w_0)$ .) We prove that  $wy, wz \in L'$  and  $y||z$  implies  $wyz \in L'$ . Since the only independence relations are  $a||c$  and  $b||c$ , we necessarily have that one of  $y$  and  $z$  is  $c$  while the other belongs to  $\{a, b\}$ . Assume w.l.o.g. that  $z = c$  and  $y \in \{a, b\}$ . From the structure of  $L'$ ,  $wy$  and  $wc$  are two prefixes of elements of  $[u_0]d$  (note that all elements of this set have the same number of  $c$ 's, namely  $|f(w_0)|$ ). Therefore there exist  $w_1, w_2 \in \Sigma^*$  such that  $wyw_1, cw_2 \in [u_0]d$ . Since  $wyw_1$  and  $wcw_2$  have the same number of  $c$ 's, we have that  $yw_1$  and  $cw_2$  have the same number of  $c$ 's, so  $w_1$  contains at least one  $c$  (because



$y \neq c$ ). Since  $c$  commutes with  $\{a, b\}$  and  $d$  appears only on the last position, there exists  $w'_1$  such that  $wycw'_1$  belongs to  $[u_0]d$ , so  $wyc \in L'$ .

The second assertion of the corollary is immediate, following from the fact that the class of languages of nondeterministic transition systems and of the deterministic ones coincide (Corollary 2.17).  $\square$

**Remark 4.44** In case the specification is *acyclic*, i.e.,  $TS$  is acyclic, Problem 4.39 becomes easily *decidable*: If  $TS$  is acyclic, then  $L(TS)$  is *finite* and we simply enumerate all the languages  $L'$  included in  $L(TS)$  and test whether  $L'$  is prefix-closed, trace-closed, and  $\Sigma(L') = \Sigma$ .

**Synchronous products** The undecidability result of Theorem 4.41 holds also when we replace the model of asynchronous automata with that of synchronous products in Problem 4.39:

**Corollary 4.45** *The relaxed implementability problem for (deterministic) synchronous products is undecidable.*

**Proof.** The construction given in the proof of Theorem 4.41 can be reused for synchronous products using the observation that the language of (deterministic) synchronous product is a (forward-closed) trace-closed language (Corollary 3.35). Then, the only extra duty for us is to show that  $L'$  from the direct implication is accepted by a (deterministic) synchronous product. Using the characterization of Theorem 3.50, this reduces to show that  $L' := \text{Prefix}([u_0]d)$ , where  $u_0 := w_0f(w_0)c^{|f(w_0)|}$  is a product language (Definition 3.40). It is easy to see that  $L'$  is indeed the product of the (local) languages

$$L_1 = \text{Prefix}(w_0f(w_0)d) \text{ and } L_2 = \text{Prefix}(c^{|f(w_0)|}d).$$

(Note that  $w_0 \in A^+ \subseteq \{a, b\}^+$ ,  $f(w_0) \in B^+ \subseteq \{a, b\}^+$ , so  $L_1 \subseteq \{a, b, d\}^*$  and  $L_2 \subseteq \{c, d\}^*$ , and therefore the local alphabets  $\{a, b, d\}$  and  $\{c, d\}$  comply with the given independent relation  $\{a \parallel c, b \parallel c\}$ .)  $\square$

### 4.5.2 Isomorphic Embedding Heuristic

The reason why the relaxed implementability as formulated in Problem 4.39 is undecidable is the potential infinite number of sub-languages of a regular language. There are several possibilities of bounding the search for solutions within a finite domain (and thus to obtain an incomplete, but decidable result). In this section we choose to restrict our search only to the languages of *isomorphic embeddings* into the specification  $TS$ . More precisely, we will look for transition systems  $E$  satisfying ID obtained by cutting out some of the transitions of  $TS$  (thus there is an injective embedding of the state space of  $E$  into  $TS$  preserving the edges). On one hand,  $L(E)$  is included in  $TS$  (because of the embedding). On the other hand, the language of  $E$  is a trace-closed language using Proposition 3.34 in conjunction with the fact that  $E$  satisfies ID. Therefore, using Theorem 3.62, finding a solution to the isomorphic embedding problem implies the existence of a solution to the relaxed implementability problem for asynchronous automata. After giving the formal details, we will show that the new problem is NP-complete.

**Definition 4.46 (Isomorphic embedding)**

We say that the transition system  $E = (Q', \Sigma', \rightarrow', I')$  is an *isomorphic embedding* of the transition  $TS = (Q, \Sigma, \rightarrow, I)$ , denoted by  $E \sqsubset TS$ , if

$$Q' \subseteq Q, \quad \Sigma' \subseteq \Sigma, \quad \rightarrow' \subseteq \rightarrow, \quad \text{and} \quad \emptyset \neq I' \subseteq I.$$

An important observation is that  $E$  is supposed to be *reachable*! (One convention of this thesis is that ‘transition system’ means in fact ‘reachable finite labeled transition system’.) Therefore, the set of transitions  $\rightarrow'$  cannot be an arbitrary subset of  $\rightarrow$ . Also, it is clear that  $E \sqsubset TS$  implies  $L(E) \subseteq L(TS)$ .

**Problem 4.47 (Isomorphic ID embedding)**

INSTANCE: *Given a concurrent alphabet  $(\Sigma, \parallel)$  and a transition system  $TS$  over  $\Sigma$  with  $\Sigma(TS) = \Sigma$ ,*  
 QUESTION: *does there exist a reachable transition system  $E$  over  $\Sigma$  such that  $E \sqsubset TS$ ,  $E$  satisfies ID, and  $\Sigma(E) = \Sigma$ ?*

**Remark 4.48** The isomorphic ID embedding problem provides a heuristic for the relaxed implementability problem. That is, a solution of the former provides a solution for the latter. The easy proof is sketched in the discussion opening the current section.

We settle now the computational complexity of the above heuristic:

**Theorem 4.49** *The isomorphic ID embedding problem is NP-complete.*

**Proof.** First, it is easy to see that the problem is in NP: Given a concurrent alphabet and a transition system  $TS$ , a nondeterministic machine can ‘guess’ a subset  $\rightarrow'$  of transitions of  $TS$  and test in *polynomial* time if the transition system  $E$  generated by  $\rightarrow'$  is reachable, satisfies ID, and  $\Sigma(E) = \Sigma$ .

For the NP-hardness part, we use a polynomial reduction from the classical *Boolean satisfiability problem* (SAT). We give an overview of the construction: Given a formula in conjunctive normal form, we associate to each variable and each clause a group of states and transitions as exemplified in Figure 4.8. We choose the independence in such a way that the ID condition ‘implements’ a choice gadget between the Boolean values **True** and **False** for each variable (i.e., either  $pos_i$  or  $neg_i$ , but not both, appear in an isomorphic ID embedding). We connect the states associated to the variables with the states associated to the clauses according to the occurrence of the variables as literals in the clauses. A clause  $c_j$  will then evaluate to **True** if and only if the state  $q_{c_j}$  associated to the clause is reachable in an isomorphic ID embedding.

Let  $\phi$  be a formula in conjunctive normal form with variables  $x_1, \dots, x_n$  and clauses  $c_1, \dots, c_m$ . We construct a transition system  $TS_\phi = (Q_\phi, \Sigma_\phi, \rightarrow_\phi, I_\phi)$  together with an independence relation  $\parallel_\phi$  on  $\Sigma_\phi$  such that  $\phi$  is satisfiable if and only if  $TS_\phi$  admits an embedding  $E \sqsubset TS_\phi$  as in Problem 4.47. To relieve the notation, we will drop all  $\phi$ -indices.

The detailed construction is given below.

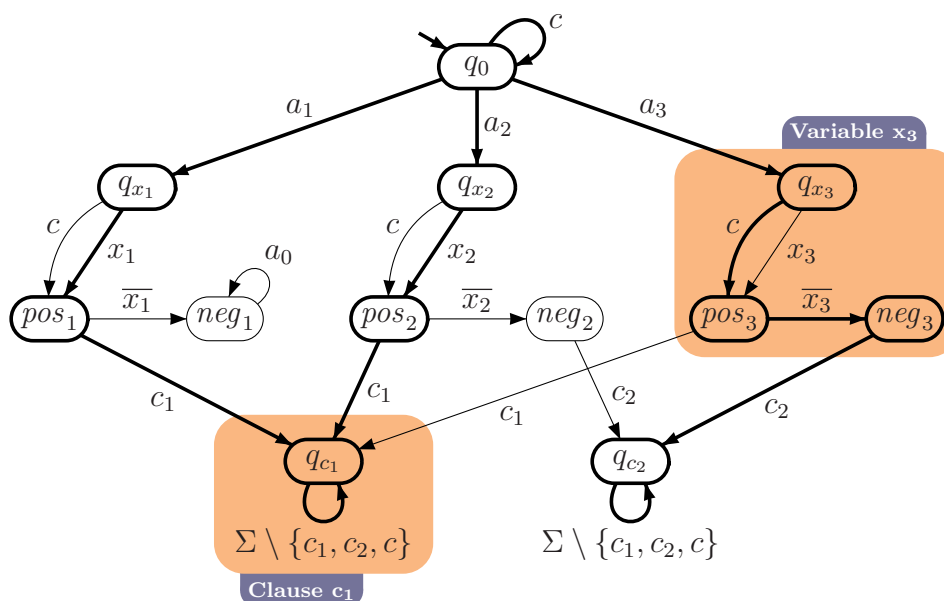


Figure 4.8: The transition system  $TS$  associated to  $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_2 \vee \bar{x}_3)$

- The alphabet  $\Sigma$  contains an action for each variable  $x_i$ , each positive and negative literal  $x_i$  and  $\bar{x}_i$ , and each clause  $c_j$ , plus two extra actions  $c$  and  $a_0$  introduced for technical reasons. Thus,

$$\Sigma := \{a_i \mid i \in [1..n]\} \cup \{x_i \mid i \in [1..n]\} \cup \{\bar{x}_i \mid i \in [1..n]\} \cup \{c_j \mid j \in [1..m]\} \cup \{c, a_0\}$$

and the independence relations on  $\Sigma$  are:

$$x_i \parallel \bar{x}_i \text{ for } i \in [1..n] \text{ and } c \parallel c_j \text{ for } j \in [1..m].$$

- the transition system  $TS$  consists of:
  - one initial state  $\{q_0\} = I$  with  $q_0 \xrightarrow{c} q_0$ ,
  - one state  $\{q_{x_i}\}$  for each variable  $x_i$  such that  $q_0 \xrightarrow{a_i} q_{x_i}$ ,
  - two states  $\{pos_i, neg_i\}$  for the positive, respectively negative, literal associated to each variable  $x_i$ , such that  $q_{x_i} \xrightarrow{x_i} pos_i$ ,  $q_{x_i} \xrightarrow{c} pos_i$ , and  $pos_i \xrightarrow{\bar{x}_i} neg_i$ . For the literals  $neg_i$  appearing in *none* of the clauses, we introduce a self-loop:  $neg_i \xrightarrow{a_0} neg_i$ .<sup>1</sup>
  - one state  $\{q_{c_j}\}$  for each clause  $c_j$  such that  $pos_i \xrightarrow{c_j} q_{c_j}$  for each positive literal  $x_i$  of  $c_j$  and  $neg_i \xrightarrow{c_j} q_{c_j}$  for each negative literal  $\bar{x}_i$  in  $c_j$ . Moreover, we introduce a self-loop  $q_{c_j} \xrightarrow{a} q_{c_j}$  for each  $a \in \Sigma \setminus (\{c_j \mid j \in [1..m]\} \cup \{c\})$ .

<sup>1</sup>These self-loops are not really necessary for the proof, but we add them in order to have at least one outgoing transition from each state. This will be later helpful when proving Corollary 4.52.

It is easy to see that the above construction is polynomial in the size of the input formula and that  $TS$  is reachable and  $\Sigma(TS) = \Sigma$ . Moreover,  $TS$  does *not* satisfy ID on the following spots:

- $q_{x_i} \xrightarrow{x_i} pos_i \xrightarrow{\overline{x_i}} neg_i$ , because  $x_i \parallel \overline{x_i}$  and there is no state  $q$  such that  $q_{x_i} \xrightarrow{\overline{x_i}} q \xrightarrow{x_i} neg_i$ , and
- $q_{x_i} \xrightarrow{c} pos_i \xrightarrow{c_j} q_{c_j}$  if the positive literal  $x_i$  appears in the clause  $c_j$ , because  $c \parallel c_j$  and there is no state  $q$  such that  $q_{x_i} \xrightarrow{c_j} q \xrightarrow{c} q_{c_j}$ .

Therefore, no isomorphic embedding  $E \sqsubset TS$  satisfying ID can contain the above pairs of transitions. In fact, this observation is the core of the choice gadget modeling the variables.

We are ready now to show that:  $\phi$  is satisfiable if and only if  $TS$  admits an isomorphic embedding  $E \sqsubset TS$  such that  $E$  is reachable, satisfies ID, and  $\Sigma(E) = \Sigma$ .

For the direct implication, if  $\phi$  is satisfiable, then there exists an assignment for the variables evaluating  $\phi$  to **True**, i.e., all the clauses are **True**. We can choose an isomorphic embedding  $E \sqsubset TS$  obtained from  $TS$  by the following systematic removal of transitions for each  $i \in [1..n]$ :

- if  $x_i = \mathbf{True}$ , remove all the incoming and outgoing transitions of  $neg_i$  and the transitions  $q_{x_i} \xrightarrow{c} pos_i$
- if  $x_i = \mathbf{False}$ , remove all the incoming and outgoing transitions of  $pos_i$  except  $q_{x_i} \xrightarrow{c} pos_i$  and  $pos_i \xrightarrow{\overline{x_i}} neg_i$ .

Figure 4.8 depicts using **bold lines** the isomorphic embedding obtained from the given  $TS$  (by removing transitions) for the assignment:

$$\langle x_1 = \mathbf{True}, x_2 = \mathbf{True}, x_3 = \mathbf{False} \rangle.$$

First, since each clause  $c_j$  is validated, so we have two possibilities:

- $x_i \in c_j$  and  $x_i = \mathbf{True}$ : According to the construction of  $E$ , the path  $q_0 \xrightarrow{a_i} q_{x_i} \xrightarrow{x_i} pos_i \xrightarrow{c_j} q_{c_j}$  belongs to  $E$ .
- $\overline{x_i} \in c_j$  and  $x_i = \mathbf{False}$ : According to the construction of  $E$ , the path  $q_0 \xrightarrow{a_i} q_{x_i} \xrightarrow{c} pos_i \xrightarrow{\overline{x_i}} neg_i \xrightarrow{c_j} q_{c_j}$  belongs to  $E$ .

In either case,  $q_{c_j}$  is reachable from  $q_0$  in  $E$  and  $c_j \in \Sigma(E)$ . Moreover,  $\Sigma \setminus (\{c_j \mid j \in [1..m]\} \cup \{c\}) \subseteq \Sigma(E)$  because the self-loops on  $q_{c_j}$  belong to  $E$ , and  $c \in \Sigma(E)$  because  $q_0 \xrightarrow{c} q_0 \in E$ . Hence,  $\Sigma(E) = \Sigma$ .

Then, it is easy to see that  $E$  is reachable (there are no isolated transitions). Finally,  $E$  satisfies ID because the proposed removal of transitions aimed to solve precisely the ID ‘conflicts’ of  $TS$  mentioned above. Moreover, ID holds also for  $x_i \parallel \overline{x_i}$  and each of the loops  $q_{c_j} \xrightarrow{x_i} q_{c_j} \xrightarrow{\overline{x_i}} q_{c_j} \xrightarrow{x_i} q_{c_j}$ , because  $q_{c_j} \xrightarrow{\overline{x_i}} q_{c_j} \xrightarrow{x_i} q_{c_j}$ .

For the converse implication, we assume there exists  $E \sqsubset TS$  an isomorphic embedding of  $TS$  that is reachable, satisfies ID, and  $\Sigma(E) = \Sigma$ , and we show that  $\phi$  is satisfiable.

Since  $E$  satisfies ID,  $E$  cannot contain both transitions  $q_{x_i} \xrightarrow{x_i} pos_i$  and  $pos_i \xrightarrow{\bar{x}_i} neg_i$  for any  $i \in [1..n]$  (because  $x_i \parallel \bar{x}_i$ ). We prove then that the following assignment validates  $\phi$ . For each  $i \in [1..n]$ ,

$$x_i := \begin{cases} \text{True,} & \text{if } q_{x_i} \xrightarrow{x_i} pos_i \text{ belongs to } E \\ \text{False,} & \text{otherwise} \end{cases} \quad (4.8)$$

Now let  $c_j$  be a clause of  $\phi$ . Since action  $c_j$  should belong to  $\Sigma(E)$  (because  $\Sigma(E) = \Sigma$ ), there must exist a path from  $q_0$  to  $q_{c_j}$  necessarily involving a transition labeled by  $c_j$ . There are two possibilities:

- $pos_i \xrightarrow{c_j} q_{c_j} \in E$  : In this case, on one hand, since  $c_j \parallel c$  and  $E$  satisfies ID,  $q_{x_i} \xrightarrow{c} pos_i$  should not belong to  $E$  (otherwise ID is violated for  $q_{x_i} \xrightarrow{c} pos_i \xrightarrow{c_j} q_{c_j}$ ). Then, since  $pos_i$  must be reachable in  $E$  and  $q_{x_i} \xrightarrow{c} pos_i \notin E$ ,  $q_{x_i} \xrightarrow{x_i} pos_i$  must belong to  $E$  and this implies by (4.8) that  $x_i$  is evaluated to **True**. On the other hand, by the construction of  $TS$ ,  $pos_i \xrightarrow{c_j} q_{c_j} \in TS$  if and only if the positive literal  $x_i$  appears in the clause  $c_j$ . Thus, we proved that  $x_i$  is evaluated to **True** and  $x_i$  appears in the clause  $c_j$ , so  $c_j$  is validated.
- $neg_i \xrightarrow{c_j} q_{c_j} \in E$  : In this case, on one hand, since  $neg_i$  must be reachable in  $E$ , we have that  $pos_i \xrightarrow{\bar{x}_i} neg_i \in E$ . Since  $x_i \parallel \bar{x}_i$  and  $E$  satisfies ID,  $q_{x_i} \xrightarrow{x_i} pos_i$  should not belong to  $E$  (otherwise ID is violated for  $q_{x_i} \xrightarrow{x_i} pos_i \xrightarrow{\bar{x}_i} neg_i$ ). This implies by (4.8) that  $x_i$  is evaluated to **False**. On the other hand, by the construction of  $TS$ ,  $neg_i \xrightarrow{c_j} q_{c_j} \in TS$  if only if the negative literal  $\bar{x}_i$  appears in the clause  $c_j$ . Thus, we proved that  $x_i$  is evaluated to **False** and  $\bar{x}_i$  appears in the clause  $c_j$ , so  $c_j$  is validated. (Note that since  $neg_i$  must be reachable in  $E$ , we also have that  $q_{x_i} \xrightarrow{c} pos_i \in E$ .)

Since in either case  $c_j$  is validated and this holds for every  $j \in [1..m]$ ,  $\phi$  is validated for the chosen assignment.  $\square$

The above reduction is versatile enough to be used in the proof of several variations of the problem.

**Corollary 4.50** *Problem 4.47 remains NP-complete even when we require that the specification  $TS$  is acyclic and deterministic.*

*Also, Problem 4.47 remains NP-complete when we require that the isomorphic embedding satisfies also FD.<sup>1</sup>*

<sup>1</sup>For a deterministic specification, an ID and FD isomorphic embedding gives a trace-closed forward-closed sub-language (Proposition 3.34). Thus we can solve the relaxed implementability problem for *deterministic* asynchronous automata (cf. Theorem 3.62).

**Proof.** It is clear that the transition system  $TS$  in the proof of Theorem 4.49 is already deterministic. Moreover,  $TS$  can be modified to be also acyclic by ‘stretching’ the self-loops (i.e., the only sources of cyclicity in  $TS$ ). For this, we add three new (distinct) states  $s_1, s_2, s_3$  replacing the self-loops as follows:  $q_0 \xrightarrow{c} q_0$  by  $q_0 \xrightarrow{c} s_1$ ,  $neg_i \xrightarrow{a_0} neg_i$  by  $neg_i \xrightarrow{a_0} s_1$ , and  $q_{c_j} \xrightarrow{a_k} q_{c_j}$  by  $q_{c_j} \xrightarrow{a_k} s_1$  for  $k \in [0..n]$ . Moreover, we replace  $q_{c_j} \xrightarrow{x_i} q_{c_j}$  by  $q_{c_j} \xrightarrow{x_i} s_1$ ,  $q_{c_j} \xrightarrow{\bar{x}_i} q_{c_j}$  by  $q_{c_j} \xrightarrow{\bar{x}_i} s_2$ , and we add  $s_1 \xrightarrow{\bar{x}_i} s_3$  and  $s_2 \xrightarrow{x_i} s_3$  for all  $j \in [1..m]$  and  $i \in [1..n]$  (in this way we construct ‘diamonds’ for the independent actions  $x_i \parallel \bar{x}_i$ ). The new transition system is acyclic, satisfies ID, and the proof of Theorem 4.49 still works.

For the second part of the corollary, we simply notice that the transition system  $TS$  in the proof of Theorem 4.49 already satisfies FD.  $\square$

In practice, we usually want to have deadlock-free implementations, that is, any run of the system can be further extended.

**Definition 4.51 (Deadlock-freeness)**

A transition system  $TS = (Q, \Sigma, \rightarrow, I)$  is *deadlock-free*, if for any  $q \in Q$ , there exist  $a \in \Sigma$  and  $q' \in Q$  such that  $q \xrightarrow{a} q'$ . Moreover, a *distributed* transition system is deadlock-free, if its global transition system is deadlock-free.

**Corollary 4.52** *Starting with a deadlock-free specification  $TS$ , Problem 4.47 with the extra requirement that the isomorphic embedding  $E \sqsubset TS$  is also deadlock-free is NP-complete.*

**Proof.** The  $TS$  and  $E$  constructed in the proof of Theorem 4.49 are both already deadlock-free.  $\square$

**Remark 4.53** It is not difficult to show that another heuristic for the relaxed implementability problem can be to look for isomorphic embeddings that are already isomorphic to an asynchronous automaton, respectively a synchronous product (theoretically, the new heuristic should be less successful than the isomorphic ID embedding one because the constraints are stronger). Using the same reduction used in the proof of Theorem 4.49 one can show also the *NP-hardness* of the new heuristic.

One needs only to prove that the isomorphic embedding  $E$  constructed in the direct implication satisfies not only ID and FD, but is in fact isomorphic to an asynchronous automaton, respectively a synchronous product. To prove this we can use the (polynomial-time expensive) characterizations for deterministic specifications from Theorem 3.32, respectively 3.29.

## Discussion

From this chapter, we learn that the models of synchronous products of transition systems and asynchronous automata have similar complexities for the implementability test.

- For both models, deciding the implementability modulo isomorphism is NP-complete (with an encouraging polynomial-time test for deterministic specifications). Moreover, a positive answer provides us already with a distributed implementation *for free* (the local equivalence classes involved in the test are used to build a distributed implementation – cf. Remark 3.28).
- The implementability modulo language equivalence is PSPACE-complete in the general case (i.e., nondeterministic specifications), with an advantage (polynomial-time test) for asynchronous automata in case the specification is deterministic. On the other hand, we have a distributed implementation for free for synchronous products with one initial state (the local components are the projections on local alphabets of the specification – recall Algorithm 4.4). This is not the case for asynchronous automata for which the computationally expensive construction of Zielonka [Zie87] is the best known approach to be used *after* we have decided that the specification is implementable. There is though a trade-off, since the asynchronous automata are strictly more expressive than the synchronous products, there will exist specifications distributable over the former model, but not over the latter (this will be the case for some of the examples we consider in the next chapters).
- For deterministic implementations, the computational complexity of implementability modulo bisimulation is similar to the case of implementability modulo language equivalence for *deterministic specifications*.
- While preparing the final version of this thesis, we found an (yet) unpublished manuscript [Tab06] providing a characterization for the implementability problem *modulo bisimulation for nondeterministic implementations* (for both synchronous products and asynchronous automata). Interesting enough, the characterization is similar to the characterization for implementability modulo isomorphism. As possible future research, we can study the computational complexity of implementability modulo bisimulation for nondeterministic implementations of the decision procedure derived from [Tab06].
- Finally, we showed that for non-regular specifications, as well as for relaxed implementability (language inclusion) the problem becomes undecidable. An NP-complete heuristic (isomorphism embedding) to relaxed implementability is provided.

As related work, we mention also that the complexity bounds of this chapter are similar for instance to checking the equivalence of distributed transition systems [Rab97, SHRS96, HKV02, VK02]. In that case, one checks if a sequential (respectively, distributed) transition system is equivalent to a *given* distributed transition system, while in our case we test if a sequential transition system is equivalent to *some* (i.e., ‘there exists one’) distributed transition system. A couple of other works touching on the complexity of the distributed implementability problem for related models of distributed systems are: [BBD95, BBD97] (Petri nets), [Roh04] (discrete-event systems), and [AEY01, BM03, Loh03] (message sequence charts).



Our motivation for exploring the complexity issues surrounding distributed implementability was based on the need to select the most appropriate implementation methods for synthesis tools. When we know the exact complexity of the subproblems of synthesis at hand we can use general rules of thumb for selecting suitable implementation techniques. For example, in the next chapters, we map the implementability problem modulo isomorphism for asynchronous automata (shown to be NP-complete) to the problem of finding a stable model of a logic program (another NP-complete problem) by using the SMODELS logic programming system [SNS02]. Same approach is adopted for the NP-complete problem of isomorphic ID embedding. Furthermore, the PSPACE-hardness of the implementability modulo language equivalence for synchronous products combined with the construction in Algorithm 4.4 suggests that using model checking algorithms can be fully appropriate. The above directions will be followed and exemplified in the next chapters.





---

It's kind of fun to do the impossible.

*Walt Disney*

## CHAPTER 5

# SYNTHESIS OF DISTRIBUTED TRANSITION SYSTEMS

---

IN the previous chapter we have studied the problem of distributed implementability, that is, given a sequential specification, we test whether there exists a distributed transition system equivalent to it. In this chapter, we proceed further and show how to effectively construct a distributed transition system from a specification that passes the implementability test.

We consider first the synthesis of synchronous products (Section 5.1) and then the synthesis of asynchronous automata (Section 5.2). Assuming the specification is distributable, the synthesis of synchronous products is easy, but not the same holds for asynchronous automata, whose synthesis proved over the years to be a hard nut to crack (Section 5.2.2). In an attempt to tackle this, we look for alternatives in special cases (Section 5.2.3).<sup>1</sup>

### 5.1 Synchronous Products of Transition Systems

Synthesizing synchronous products is easy once we have decided that the specification is equivalent to a synchronous product. The loose communication between the processes in the synchronous product model makes the distributivity test hard (see Chapter 4), but a positive answer will provide an implementation for free. We sketch below the constructions generated by the implementability tests.

#### 5.1.1 Synthesis modulo Isomorphism

For the synthesis modulo isomorphism, we have a transition system  $TS = (Q, \Sigma, \rightarrow, I)$  together with a distribution  $(\Sigma, Proc, \Delta)$  as specification and we want to construct a synchronous product  $\mathcal{SP}$  over  $\Delta$  that is isomorphic to  $TS$ . The test whether such synchronous product exists is NP-complete (Theorem 4.3) and the decision procedure is based on Theorem 3.26. In the particular case of deterministic specifications, the test can be performed in polynomial time (Proposition 4.9) relying on Theorem 3.29.

---

<sup>1</sup>Some of the results of this chapter were published in [SEM03].

The cornerstone of the above results is a family of local equivalences  $(\equiv_p)_{p \in Proc}$  over  $Q$  satisfying the properties in Theorem 3.26, respectively Theorem 3.29. For nondeterministic specifications, we can ‘guess’ the local equivalences (the ways of partitioning a finite state space using equivalence relations is bounded) and then test them against the conditions  $SP_1$ – $SP_3$  in Theorem 3.26. For deterministic specifications, we construct the least family of local equivalences meeting the criteria  $DSP_1$ – $DSP_2$  (using a fixed point procedure similar to Algorithm 4.3) and check if  $DSP_3$ – $DSP_4$  are also satisfied.

The local equivalences testifying the distributed implementability of  $TS$  immediately generate a synchronous product  $\mathcal{SP}$  (isomorphic to  $TS$ ) as the synchronization of the following local transition systems [CMT99]:

- for each  $p \in Proc$ , the local state set  $Q_p$  is the set of equivalence classes of  $\equiv_p$ , i.e.,

$$Q_p := Q / \equiv_p,$$

- for each  $p \in Proc$ , the local transition relation  $\rightarrow_p \subseteq Q_p \times \Sigma_{loc}(p) \times Q_p$  is defined as

$$\begin{aligned} [q]_p \xrightarrow{a}_p [q']_p \text{ if and only if there exists } s \xrightarrow{a} s' \text{ in } TS \text{ with } a \in \Sigma_{loc}(p), \\ s \in [q]_p \text{ and } s' \in [q']_p, \text{ and} \end{aligned}$$

- the set of global initial states of  $\mathcal{SP}$  is given by the set of initial states of  $TS$ :

$$\{ ([q]_p)_{p \in Proc} \mid q \in I \}.$$

Since  $\mathcal{SP}$  is isomorphic to  $TS$ , the size of the synthesized synchronous product is linear in the size of the specification.

### 5.1.2 Synthesis modulo Language Equivalence

In Section 4.3.1, we showed that the implementability problem for synchronous products with one initial state is PSPACE-complete, even for deterministic specifications. In Section 4.3.2, we showed that the same problem for synchronous products with multiple initial states is PSPACE-hard (Theorem 4.28), but unfortunately we do not have yet an upper bound and thus we do not even know if the problem is after all decidable. In this context, we consider only the synthesis of synchronous products with one initial state (in particular, the approach also works for *deterministic* synchronous products).

The synthesis of synchronous products modulo language equivalence is based on Algorithm 4.4 that decides whether the language of transition system is a product language. According to Algorithm 4.4, we start with a distribution  $\Delta$  and a transition system  $TS$  (supposed w.l.o.g. to have only one initial state – see Step 0). Then, for each process  $p \in Proc$ , we construct the projections  $TS_p$  of  $TS$  on the local alphabets  $\Sigma_{loc}(p)$  (such that  $L(TS_p) = L(TS) \upharpoonright_{\Sigma_{loc}(p)}$  – see Step 1). Putting the local transition systems  $(TS_p)_{p \in Proc}$  in parallel, we obtain a synchronous product  $\mathcal{SP}$  taking as unique global initial state the tuple of the local initial states (if  $TS$  is assumed to have only one initial state, its projections  $TS_p$  will also have only one initial state each). Finally, according to Step 3 of Algorithm 4.4, if  $L(\mathcal{SP}) \subseteq L(TS)$  then  $\mathcal{SP}$  is a synchronous product with one initial state that is language equivalent to  $TS$  (otherwise the specification is not implementable).

Hence, the synchronous product  $\mathcal{SP}$  constructed by Algorithm 4.4 is a solution of the synthesis problem and its size (i.e., the sum of the sizes of the local components) is linear in the size of the specification. Moreover, if we want to obtain a *deterministic* synchronous product, we simply determinize the local transition systems  $TS_p$  (in this case, an exponential blowup of the state space according to the powerset construction may occur).

**Synthesis modulo bisimulation** The approach above can be also used to obtain a *deterministic* synchronous product *bisimilar* to the implementation (for nondeterministic synchronous products the implementability problem is still open – although very recently [Tab06] claims to tackle this case). According to the proof of Theorem 4.33 [CMT99, Muk02], we construct the quotient transition system  $TS/\sim_{TS}$  (where  $\sim_{TS}$  is the largest bisimulation on  $TS$ ) and in case this quotient is deterministic we follow the above algorithm with  $TS/\sim_{TS}$  as specification. If the implementability test is positive, a *deterministic* synchronous product that is bisimilar to  $TS$  is obtained by projecting the quotient  $TS/\sim_{TS}$  on each local alphabets  $\Sigma_{loc}(p)$ , then *determinizing* (and optionally minimizing) each projection<sup>1</sup>, and finally taking their synchronous product. The synthesized synchronous product may be exponentially larger than the specification because of the powerset construction involved in the determinization of the local transition systems.

## 5.2 Asynchronous Automata

For the synthesis modulo isomorphism, the constructions of asynchronous automata are similar to those of synchronous products. On the other hand, for language equivalence, the synthesis problem for asynchronous automata becomes much more difficult compared to the case of synchronous products.

### 5.2.1 Synthesis modulo Isomorphism

Given the similar characterization results, all the remarks in Section 5.1.1 regarding the synchronous products apply also to asynchronous automata. The only difference is that instead of constructing local transition relations  $\rightarrow_p$  for each  $p \in Proc$ , we construct local transition relations  $\rightarrow_a \subseteq \prod_{p \in dom(a)} Q_p \times \prod_{p \in dom(a)} Q_p$  for each action  $a \in \Sigma$  (cf. Definition 3.18) in the following way:

$$([q]_p)_{p \in dom(a)} \rightarrow_a ([q']_p)_{p \in dom(a)} \text{ if and only if there exists } s \xrightarrow{a} s' \text{ in } TS \text{ with } s \in [q]_p \text{ and } s' \in [q']_p \text{ for all } p \in dom(a).$$

Again, the size of the synthesized asynchronous automaton is linear in the size of the specification.

For *deterministic* specifications, the synthesis problem modulo isomorphism is solved by Theorem 3.32. Furthermore, we can relax the synthesis problem by dropping condition DAA<sub>3</sub> of Theorem 3.32 as follows:

<sup>1</sup>Even if the original transition system  $TS/\sim_{TS}$  is deterministic, its projection on a local alphabet of actions may be nondeterministic, so a determinization procedure is required in order to obtain a deterministic implementation.

**Remark 5.1** Let  $TS$  be a *deterministic* transition system and  $(\Sigma, Proc, \Delta)$  be a distribution. According to Theorem 3.32, there is the least family of equivalences  $(\equiv_k)_{k \in Proc}$  satisfying DAA<sub>1</sub> and DAA<sub>2</sub>. If  $(\equiv_k)_{k \in Proc}$  satisfies DAA<sub>4</sub> (but not necessarily DAA<sub>3</sub>!), then there exists an asynchronous automaton  $\mathcal{AA}$  over  $\Delta$  accepting the same language as  $TS$  (in fact  $\mathcal{AA}$  is bisimilar to  $TS$ ). The asynchronous automaton  $\mathcal{AA}$  is constructed from the family of local equivalences  $(\equiv_k)_{k \in Proc}$  by quotients as showed at the beginning of this section and it can be easily proved that  $L(\mathcal{AA}) = L(TS)$ .

Hence, dropping DAA<sub>3</sub> we obtain a *sufficient* test for the implementability modulo language equivalence that produces asynchronous automata that are *linear* in the size of the specification. Moreover, the test can be performed in polynomial time (Proposition 4.9).<sup>1</sup>

However, the above relaxation does not hold for finite automata, i.e., when the transition system is equipped with a set of accepting states, as the following (counter)example shows:

**Example 5.2** Take the transition system  $TS$  in Example 3.23 (Figure 3.6) with 1 as the only accepting state.

On one hand, we have  $L(TS, \{1\}) = \{w \in \{a, b\}^* \mid |w| \text{ is odd}\}$ . On the other hand, using DAA<sub>1</sub>, the states 0 and 1 are equivalent on all processes of the distribution (see proof of Example 3.23), so the generated asynchronous automaton  $\mathcal{AA}$  will accept the whole language  $\{a, b\}^*$ , which means that the synthesized implementation is not language-equivalent to the specification.

## 5.2.2 Synthesis modulo Language Equivalence

The ‘backbone’ of synthesis of asynchronous automata is a result by Zielonka [Zie87] showing that for any regular trace-closed language  $T$  there exists a deterministic asynchronous automaton with global final states accepting  $T$  (Theorem 3.60). Since in our study we consider global final states inappropriate for distributed systems, the definition of the language of an asynchronous automaton implicitly assumes all states as final (Definition 3.20), with the consequence that the languages we work with are prefix-closed. In Chapter 3 we stated characterizations of the classes of languages accepted by deterministic, respectively nondeterministic, asynchronous automata (Theorem 3.62). We gave at that point only the proof for *nondeterministic* asynchronous automata (we used for that Theorem 3.61 from [Zie89]), while the deterministic case was postponed to this chapter. Based on Zielonka’s original construction [Zie87], we present below the proof for the deterministic case, in fact solving the synthesis problem modulo language equivalence for *deterministic* asynchronous automata.

Using Corollary 3.35, the language accepted by a *deterministic* asynchronous automaton is a prefix-closed regular *forward-closed* trace-closed language. We will show that the reverse also holds, that is, for any prefix-closed regular forward-closed trace-closed language  $T$ , there exists a deterministic asynchronous automaton  $\mathcal{AA}$  such that  $L(\mathcal{AA}) = T$ . To show this, we slightly modify Zielonka’s proof, that builds for any regular trace-closed

<sup>1</sup>For nondeterministic specifications, the similar test obtained by dropping AA<sub>2</sub> in Theorem 3.30 is NP-complete (Corollary 4.14).

A very similar remark holds for synchronous products as well.

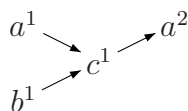


Figure 5.1: The partial order of a trace

language  $T$ , a deterministic asynchronous automaton with a set  $F$  of global final states such that  $L(\mathcal{AA}, F) = T$ .

We will present first the ingredients of Zielonka's approach together with some properties, then we modify the construction to have all states are final (for prefix-closed specifications), and finally prove that the modified asynchronous automaton is consistent and accepts the language of the specification.

Fixing a distribution  $(\Sigma, Proc, \Delta)$  that generates an independence relation  $\parallel$ , we can define the following notions [Zie87]:

**Partial order on the events of a trace** For  $t \in \Sigma^*$  we define the set of *events* of  $t$  or occurrences of actions of  $t$  as

$$O(t) = \{a^i \mid a \in \Sigma \text{ and } 1 \leq i \leq \#_a t\}.$$

We denote by  $\leq_t$  the total order on  $O(t)$  given by the order in which the actions occur in  $t$ .

Using now the independence relation  $\parallel$  (that associates to each  $t \in \Sigma^*$ , a trace  $[t]$ ), we can define on  $O(t)$  also a *partial order*  $\preceq$  such that for any  $a^i, b^j \in O(t)$ :

$$a^i \preceq b^j \text{ if and only if for all } w \in \Sigma^* \text{ with } w \in [t], \text{ we have } a^i \leq_w b^j.$$

Equivalently,  $\preceq = \bigcap_{w \in [t]} \leq_w$  (note that if  $w \in [t]$ , i.e.,  $w \sim t$ , then  $O(w) = O(t)$ ). The partial order  $\preceq$  defines the *causality* relation between events and is used in the following to generate the prefixes and suffixes of  $t$  associated with a subset of processes.

**Example 5.3** Consider, for instance, the distribution of the actions  $\Sigma := \{a, b, c\}$  over two processes  $Proc := \{1, 2\}$ , with  $dom(a) := \{1\}$ ,  $dom(b) := \{2\}$ , and  $dom(c) := \{1, 2\}$  (so  $a \parallel b$ ). Choosing the word

$$t := abca,$$

we have:

- $O(t) := \{a^1, a^2, b^1, c^1\}$  with the total order  $a^1 \leq_t b^1 \leq_t c^1 \leq_t a^2$ .
- the trace associated to  $t$  is  $[t] := \{abca, bac a\}$  (we have  $a \parallel b$ ), generating the partial order  $a^1 \preceq c^1$ ,  $b^1 \preceq c^1$ , and  $c^1 \preceq a^2$ . Figure 5.1 depicts the partial order  $(O(t), \preceq)$  using arrows  $x \rightarrow y$  to represent  $x \preceq y$ .

**Prefixes and suffixes of a trace** For  $t \in \Sigma^*$  and  $K \subseteq Proc$  we define the *prefix* of  $t$  with respect to  $K$  as

$$\text{Pref}_K(t) := \{x \in O(t) \mid \exists y \in O(t), x \preceq y \text{ and } \text{dom}(y) \cap K \neq \emptyset\}.$$

Informally,  $\text{Pref}_K(t)$  contains the events ‘visible’ (in the past) from the processes of  $K$ .

The *suffix* of  $t$  with respect to  $K$  is obtained by complementation:

$$\text{Suff}_K(t) := O(t) \setminus \text{Pref}_K(t).$$

Concatenating the events of  $\text{Pref}_K(t)$ , respectively  $\text{Suff}_K(t)$ , according to the total order  $\leq_t$ , we obtain two words denoted by

$$P_K(t), \text{ respectively } S_K(t).$$

According to [Zie87, Fact 4.11] we have that for any  $t \in \Sigma^*$  and  $K \subseteq Proc$ :

$$t \sim P_K(t)S_K(t) \tag{5.1}$$

Finally, for the singleton  $K := \{k\}$  with  $k \in Proc$ , we denote  $P_{\{k\}}(t)$  by  $P_k(t)$ .

**Example 5.3 (continued)** For  $t = abca$  (see also Figure 5.1), we have  $\text{Pref}_{\{1\}} = \text{Pref}_{\{1,2\}} = \{a^1, a^2, b^1, c^1\}$  and  $\text{Pref}_{\{2\}} = \{a^1, b^1, c^1\}$ , which implies  $\text{Suff}_{\{1\}} = \emptyset$ ,  $\text{Suff}_{\{1,2\}} = \emptyset$ , and  $\text{Suff}_{\{2\}} = \{a^2\}$ . Therefore,  $P_1 = P_{\{1,2\}} = abca$  and  $P_2 = abc$ , while  $S_1 = S_{\{1,2\}} = \varepsilon$  and  $S_2 = a$ .

**Zielonka’s equivalence** For each regular trace-closed language  $T \subseteq \Sigma^*$ , Zielonka constructs an equivalence  $\approx \subseteq \Sigma^* \times \Sigma^*$  including information regarding the syntactic congruence  $\sim_T$  as well as time-stamping [Zie87]. The equivalence  $\approx$  is the fundamental element in the construction of a deterministic asynchronous automaton with final states accepting  $T$ . We will not present the details behind  $\approx$ , because they are rather non-intuitive and for our purposes in this chapter, it is enough to know that  $\approx$  has finite index (for a regular trace-closed language  $T$ ) and that the following properties hold<sup>1</sup>.

(For more details regarding Zielonka’s construction [Zie87], we point the reader to the monograph [DR95], Pighizzini’s PhD thesis [Pig93a], and the technical report [MS94].)

First, Zielonka’s equivalence is coarser than the trace equivalence, but finer than the syntactic congruence:

**Proposition 5.4** *If  $t \sim t'$ , then  $t \approx t'$ , for any  $t, t' \in \Sigma^*$ .*

<sup>1</sup>In fact, the difficulty of the synthesis problem consists of finding an equivalence of finite index satisfying Propositions 5.4–5.7. Zielonka [Zie87] came up with such an equivalence and it is still an open problem whether a ‘simpler’ such equivalence exists.

**Proposition 5.5** *If  $t \approx t'$ , then  $t \sim_T t'$ , for any  $t, t' \in \Sigma^*$  (where  $\sim_T$  is the syntactic congruence of  $T$  on  $\Sigma^*$ ).*

Then, Zielonka's equivalence is preserved when partial views are combined:

**Proposition 5.6** *If  $P_K(t) \approx P_K(t')$  and  $P_L(t) \approx P_L(t')$ , then  $P_{K \cup L}(t) \approx P_{K \cup L}(t')$ , for any  $t, t' \in \Sigma^*$  and  $K, L \subseteq Proc$ .*

Finally, Zielonka's equivalence is preserved when partial views are extended by an event:

**Proposition 5.7** *If  $\forall k \in dom(a) : P_k(t) \approx P_k(t')$ , then  $\forall k \in dom(a) : P_k(ta) \approx P_k(t'a)$ , for any  $t, t' \in \Sigma^*$  and  $a \in \Sigma$ .*

**Zielonka's asynchronous automaton** Let  $(\Sigma, Proc, \Delta)$  be a distribution and  $T$  a regular trace-closed language over  $\Sigma^*$ . For  $t \in \Sigma^*$ , we denote by  $\llbracket t \rrbracket$  the equivalence class of  $t$  with respect to  $\approx$  (i.e., Zielonka's equivalence generated by  $T$  on  $\Sigma^*$ ). Also, we suppose that  $Proc = \{1, \dots, n\}$ .

Following (the first part of) Definition 3.18, we construct:

- for each  $k \in Proc$ , a local state space

$$Q_k := \{\llbracket P_k(t) \rrbracket \mid t \in \Sigma^*\}$$

( $Q_k$  is finite because  $\approx$  is of finite index) and

- for each action  $a \in \Sigma$ , a local transition relation  $\rightarrow_a$  such that for  $t \in \Sigma^*$  and  $dom(a) = \{k_1, \dots, k_m\}$ ,

$$(\llbracket P_{k_1}(t) \rrbracket, \dots, \llbracket P_{k_m}(t) \rrbracket) \rightarrow_a (\llbracket P_{k_1}(ta) \rrbracket, \dots, \llbracket P_{k_m}(ta) \rrbracket).$$

Using Proposition 5.7 we have that  $\rightarrow_a$  is deterministic (i.e., any tuple has via  $\rightarrow_a$  at most one successor).

Zielonka's asynchronous automaton  $\mathcal{AA} = (Q, \Sigma, \rightarrow, \{s_0\}, F)$  is based on the above local elements:

- the initial global state is  $s_0 := (\llbracket \varepsilon \rrbracket, \dots, \llbracket \varepsilon \rrbracket)$ .
- the global transition relation  $\rightarrow$  is generated by the local relations  $\rightarrow_a$  as in Definition 3.18. It can easily be proved that for any  $t \in \Sigma^*$ ,

$$(\llbracket \varepsilon \rrbracket, \dots, \llbracket \varepsilon \rrbracket) \xrightarrow{t} (\llbracket P_1(t) \rrbracket, \dots, \llbracket P_n(t) \rrbracket). \quad (5.2)$$

Since all the  $\rightarrow_a$ s are deterministic,  $\rightarrow$  is also deterministic by Proposition 3.25.

- the global state space  $Q \subseteq \prod_{k \in Proc} Q_k$  is obtained as the part of the cartesian product reachable (via  $\rightarrow$ ) from the initial global state  $s_0$ . Using (5.2) we have that

$$Q = \{(\llbracket P_1(t) \rrbracket, \dots, \llbracket P_n(t) \rrbracket) \mid t \in \Sigma^*\}.$$

- the set of final global states is  $F := \{(\llbracket P_1(t) \rrbracket, \dots, \llbracket P_n(t) \rrbracket) \mid t \in T\}$ .

Using (5.2) together with Propositions 5.6 and 5.5 we have that indeed  $L(\mathcal{AA}, F) = T$ .



**A modification of Zielonka's asynchronous automaton** We give now the proof for Theorem 3.62, i.e., we show how to construct a *deterministic* asynchronous automaton  $\mathcal{AA}'$  (with all states final) accepting a prefix-closed regular forward-closed trace-closed language  $T$ , i.e.,  $L(\mathcal{AA}') = T$ . To do this, we slightly change Zielonka's asynchronous automaton above by restricting the set of global states  $Q$  to those reachable by executing only sequences of actions from  $T$ .

Let  $(\Sigma, Proc, \Delta)$  be a distribution with  $Proc = \{1, \dots, n\}$  and  $T \subseteq \Sigma^*$  a prefix-closed regular forward-closed trace-closed language over  $\Sigma^*$ . Since  $T$  is a regular trace-closed language, we can construct Zielonka's equivalence  $\approx$  for  $T$  satisfying Propositions 5.4–5.7. We build an asynchronous automaton  $\mathcal{AA}' = (Q', \Sigma, \rightarrow', s'_0)$  similar to Zielonka's asynchronous automaton above with the difference that every occurrence of  $t \in \Sigma^*$  is restricted to  $t \in T$ . More precisely:

- for each  $k \in Proc$ , the local state space is  $Q'_k := \{\llbracket P_k(t) \rrbracket \mid t \in T\}$ ,
- for each action  $a \in \Sigma$ , the local transition relation  $\rightarrow'_a$  is defined for  $dom(a) = \{k_1, \dots, k_m\}$  and  $t \in T$  such that  $ta \in T$  as

$$(\llbracket P_{k_1}(t) \rrbracket, \dots, \llbracket P_{k_m}(t) \rrbracket) \rightarrow'_a (\llbracket P_{k_1}(ta) \rrbracket, \dots, \llbracket P_{k_m}(ta) \rrbracket).$$

Using Proposition 5.7 we have that  $\rightarrow_a$  is deterministic. Moreover, the given definition is proved to be sound by Proposition 5.8 below (we want the definition to be consistent given the extra condition  $ta \in T$  – see also Remark 5.9).

As before, the global state space  $Q'$  of  $\mathcal{AA}'$  is constructed using the synchronization  $\rightarrow'$  of the relations  $\rightarrow'_a$  starting in the initial state  $s'_0 := (\llbracket \varepsilon \rrbracket, \dots, \llbracket \varepsilon \rrbracket)$ . Under the new circumstances, (5.2) holds again (note that all the intermediary states during the execution of  $t$  are indeed in  $Q'$  because  $T$  is prefix-closed) and this implies that

$$Q' := \{(\llbracket P_1(t) \rrbracket, \dots, \llbracket P_n(t) \rrbracket) \mid t \in T\}. \quad (5.3)$$

It is not hard to show now that indeed  $L(\mathcal{AA}') = T$ .

The only thing to prove is the soundness of the definition of the local transition relations  $\rightarrow'_a$  which is implied by the following result (where the forward closure property is used):

**Proposition 5.8** *Let  $T \subseteq \Sigma^*$  be a prefix-closed regular forward-closed trace-closed language,  $t, t' \in T$ , and  $a \in \Sigma$ . If  $ta \in T$  and  $\forall k \in dom(a) : P_k(t) \approx P_k(t')$ , then  $t'a \in T$ .*

**Proof.** First, from (5.1) we have  $ta \sim P_{dom(a)}(ta)S_{dom(a)}(ta)$ , which together with the hypothesis  $ta \in T$  and  $T$  being trace-closed, implies that  $P_{dom(a)}(ta)S_{dom(a)}(ta) \in T$ . Since  $T$  is also prefix-closed, we further obtain:

$$P_{dom(a)}(ta) \in T. \quad (5.4)$$

Next, from the hypothesis  $P_k(t) \approx P_k(t')$  for all  $k \in dom(a)$ , by Proposition 5.7, we have that  $P_k(ta) \approx P_k(t'a)$ , for all  $k \in dom(a)$ . Then, repeatedly applying Proposition 5.6, we get  $P_{dom(a)}(ta) \approx P_{dom(a)}(t'a)$ . Using Proposition 5.5, we deduce that  $P_{dom(a)}(ta) \sim_T$



$P_{dom(a)}(t'a)$ . This (syntactic) equivalence and (5.4) give  $P_{dom(a)}(t'a) \in T$ . From this and the simple observation (see definition of the prefix of a trace) that  $P_{dom(a)}(t'a) = P_{dom(a)}(t')a$  we deduce that

$$P_{dom(a)}(t')a \in T. \quad (5.5)$$

In the last part, we prove that (5.5) implies  $t'a \in T$  using the forward closure of  $T$ .

First, according to (5.1) we have that

$$t' \sim P_{dom(a)}(t')S_{dom(a)}(t') \quad (5.6)$$

Moreover, if  $S_{dom(a)}(t') = b_1 \dots b_m$ , then  $dom(b_i) \cap dom(a) = \emptyset$  for each  $i \in [1..m]$  (otherwise  $b_i$  would belong to  $P_{dom(a)}(t')$ ). Therefore,

$$b_i \parallel a, \text{ for all } i \in [1..m]. \quad (5.7)$$

From (5.6) and the hypothesis  $t' \in T$ , we have (by the trace closure of  $T$ ) that

$$P_{dom(a)}(t')b_1 \dots b_m \in T.$$

Furthermore,  $P_{dom(a)}(t')b_1 \dots b_i \in T$  for all  $i \in [1..m]$  because  $T$  is prefix-closed.

Therefore, since  $P_{dom(a)}(t')b_1 \dots b_i \in T$  for all  $i \in [1..m]$  and  $P_{dom(a)}(t')a \in T$  (5.5) and  $b_i \parallel a$  for all  $i \in [1..m]$  (5.7), we can apply the forward closure property *iteratively* for  $i$  from 1 to  $m$  and finally obtain that  $P_{dom(a)}(t')b_1 \dots b_m a \in T$ . That is,

$$P_{dom(a)}(t')S_{dom(a)}(t')a \in T. \quad (5.8)$$

Finally, concatenating  $a$  to the terms of (5.6) we have that  $t'a \sim P_{dom(a)}(t')S_{dom(a)}(t')a$ . From this, (5.8) and the trace closure of  $T$ , we obtain that  $t'a \in T$ .  $\square$

**Remark 5.9** The global state space  $Q'$  is obtained from the global state space of Zielonka's automaton  $Q$  by selecting only the states reachable by words from  $T$ . Proposition 5.8 confirms in fact that the restricted state space is indeed isomorphic to a (deterministic) asynchronous automaton. More precisely, it shows that the condition  $DAA_4$  from the implementability test for asynchronous automata given in Theorem 3.32 holds.

**Synthesis modulo bisimulation** According to Algorithm 4.6, one can decide in polynomial time if a transition system  $TS$  is bisimilar to a *deterministic* asynchronous automaton (for nondeterministic asynchronous automata the implementability problem is still open – although very recently [Tab06] claims to tackle this case). If the test is positive,  $TS/\sim_{TS}$  is deterministic and  $L(TS/\sim_{TS})$  is a forward-closed trace-closed (prefix-closed regular) language. In this case, the deterministic asynchronous automaton bisimilar to the specification  $TS$  is simply given by the above modification of Zielonka's automaton for the language  $L(TS/\sim_{TS})$ .

**Size of the distributed implementation** The synthesized asynchronous automaton following Zielonka’s approach is unfortunately very large. For a regular trace-closed language  $T$  over the concurrent alphabet  $(\Sigma, \parallel)$ , a careful analysis conducted by Pighizzini in his PhD thesis [Pig93a, page 89] gives the following set as hint for the size of (a local component of) Zielonka’s asynchronous automaton for  $T$ :

$$Q^{Q \times \mathcal{P}(\Sigma)} \times (\mathcal{P}(\Sigma \times \{1, \dots, |\Sigma|\}))^{\mathcal{P}(\Sigma)} \times (\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma))^{\mathcal{P}(\Sigma)} \times (\Sigma \times \{1, \dots, |\Sigma|\})^{\Sigma \times \Sigma},$$

where  $Q$  is the state space of the minimal deterministic finite automaton accepting  $T$ . Subsequent endeavors [CMZ93, MS94] only presented slight improvements over the original proposal by Zielonka, the achievements being mainly a better presentation of the construction, respectively proof of correctness (but the core ideas remaining the same).

A different approach was followed by Pighizzini [Pig93b, Pig93a] who gave a construction of *nondeterministic* asynchronous automata accepting the same language as a concurrent regular expression<sup>1</sup> (the construction is double exponential, but only in the number of nested  $*$ -iterations). Nevertheless, when the specification is given as a transition system  $TS$ , there may be an exponential blowup involved in the translation to a concurrent regular expression accepting the language of  $TS$ . Moreover, as far as we can judge, the nondeterministic asynchronous automata synthesized in [Pig93b] are not *safe* in the sense of [Zie89] such that we can use the construction in our setting where all global states are final (see the proof of Theorem 3.62 where the safety property is essential).<sup>2</sup> Pighizzini’s approach is complemented by determinization procedures for asynchronous automata [KMS94, Mus96] (showing also that a blowup of the state space is unavoidable by determinization).

In an attempt to fight the state space explosion problem for (deterministic) asynchronous automata, we propose a heuristic based on unfoldings in Section 6.3.3. Yet another possibility is to try to give alternative (more efficient) constructions in special cases. We study this in the next section.

### 5.2.3 Alternative Constructions for Special Cases

Since the construction of asynchronous automata proved to be ‘a hard nut to crack’ in the general case, some research effort was directed to special cases. Thus, Duboc [Dub86] constructed asynchronous automata for a finite trace or the star closure of a connected trace<sup>3</sup>. Then, Métivier [Mét87] gave a polynomial<sup>4</sup> construction for *acyclic* dependence graphs of actions. His construction was generalized in [DM96] to the case

<sup>1</sup>I.e., a regular expression taking concurrency into account when defining the iteration – see [Pig93b].

<sup>2</sup>Just recently, an alternative construction to Pighizzini’s construction was proposed in [BM06]. The authors construct a *nondeterministic* asynchronous automaton that is polynomial in the size of the original transition system (and double exponential in the size of the distribution). However, the generated asynchronous automaton is not *safe* (in the sense of [Zie87]), so it cannot be really used in our setting where all states are final.

<sup>3</sup>A trace  $[t]$  with  $t \in \Sigma$  is called *connected* if the subgraph of the dependence graph  $(\Sigma, \parallel)$  generated by the actions appearing in  $t$  (i.e.,  $\Sigma(t)$ ) is connected.

<sup>4</sup>The construction is polynomial when starting with a monoid homomorphism; when starting with a transition system instead of a monoid homomorphism, the construction is exponential.

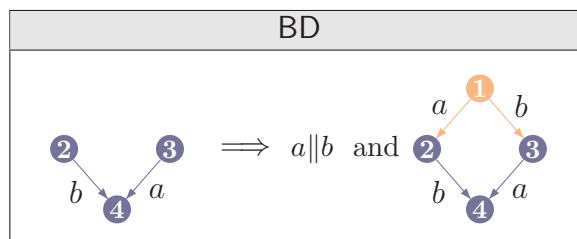


Figure 5.2: The backward diamond property

of triangulated<sup>1</sup> dependence graphs. A possible problem with last approach is that it is not flexible regarding the distribution. The input to the procedure is a transition system together with a *concurrent alphabet*  $(\Sigma, \parallel)$ , from which a *very specific* distribution complying with  $(\Sigma, \parallel)$  is constructed. This distribution seems not very useful in practice (e.g., we could not use it with our examples).

The constructions given in this subsection will make use of the characterizations modulo isomorphism of the global states of asynchronous automata, i.e., Theorems 3.30 and 3.32. We will show that in particular cases (i.e., finite specifications, conflict-free specifications, or transitive dependence graphs) the tests provided by the above theorems are positive and we can immediately construct equivalent asynchronous automata.

### Asynchronous automata for finite languages

The finite specifications (given by *acyclic* transition systems) are the first special cases for which we give an alternative construction to Zielonka's method.

The main ingredient is the following *backward* diamond rule<sup>2</sup> defined as:

$$\text{BD: } q_2 \xrightarrow{b} q_4 \wedge q_3 \xrightarrow{a} q_4 \Rightarrow a \parallel b \wedge q_2 \neq q_3 \wedge \exists q_1 \in Q : q_1 \xrightarrow{a} q_2 \wedge q_1 \xrightarrow{b} q_3.$$

(The presence of condition  $q_2 \neq q_3$  in BD is informally justified by the similar inequality in the second part of Remark 3.31.)

The technical result supporting the construction for acyclic specifications is the following:

**Theorem 5.10** *Let  $\Delta = (\Sigma, Proc, \Delta)$  be a distribution. Then, any acyclic deterministic transition system  $TS$  satisfying ID, FD, and BD is isomorphic to an asynchronous automaton over  $\Delta$ .*

**Proof.** Let  $TS$  be a deterministic transition system satisfying the hypothesis. We show using Theorem 3.32 (which is the specialization of the general Theorem 3.30 to deterministic transition systems) that  $TS$  is indeed isomorphic to an asynchronous automaton over  $\Delta$ .

Let  $(\equiv_k)_{k \in Proc}$  be the least family of equivalences satisfying DAA<sub>1</sub> and DAA<sub>2</sub> from Theorem 3.32. We denote

<sup>1</sup>A graph is *triangulated* if and only if all the chord-less cycles are of length three.

<sup>2</sup>This was suggested by Javier Esparza in one of the discussions we had.

- $equiv(q_1, q_2) := \{p \in Proc \mid q \equiv_p q'\}$ ,  
i.e., the maximal subset of  $K \subseteq Proc$  such that  $q_1 \equiv_K q_2$ , and
- $\overline{dom}(a) := Proc \setminus dom(a)$ ,  
i.e., the complement of the domain.

For a transition system  $TS$  satisfying the hypothesis hold:

- There could be at most one transition between  $q_1$  and  $q_2$ :  
We cannot have  $q_1 \xrightarrow{a} q_2$  and  $q_1 \xrightarrow{b} q_2$  because of BD, and we cannot have  $q_1 \xrightarrow{a} q_2$  and  $q_2 \xrightarrow{b} q_1$  because of acyclicity.
- If from a state  $q_1$  we find two directed paths to another state  $q_2$ , then the set of actions appearing on the two paths are equal:

Proof by induction on the sum of lengths of the two paths (again using BD and the acyclicity condition).

Using the above remarks we can show that for any two distinct states  $q_1, q_2 \in Q$  there exists a set of actions  $A(q_1, q_2) := \{a_1, \dots, a_n\} \subseteq \Sigma$  such that

$$equiv(q_1, q_2) = \bigcap_{i=1}^n \overline{dom}(a_i) \quad (5.9)$$

The set  $A(q_1, q_2) = \{a_1, \dots, a_n\}$  above consists of the actions on the *shortest undirected* path (i.e., a path in  $TS$  where we ignore the direction of transitions) connecting  $q_1$  and  $q_2$ . The equation (5.9) follows from the following observations:

1. Using DAA<sub>1</sub> applied to the actions  $a_i$  and the transitivity property of the equivalences  $(\equiv_k)_{k \in Proc}$  we have that  $q_1 \equiv_{\bigcap_{i=1}^n \overline{dom}(a_i)} q_2$ , so we obtain ‘ $\supseteq$ ’ of (5.9).
2. Using the hypotheses and the previous remarks, one can prove that any undirected path connecting  $q_1$  and  $q_2$  will contain all the actions  $A(q_1, q_2) = \{a_1, \dots, a_n\}$ . Thus, using only DAA<sub>1</sub>, we cannot have  $q_1 \equiv_p q_2$  with  $p \notin \bigcap_{i=1}^n \overline{dom}(a_i)$  (\*).
3. Moreover, using DAA<sub>2</sub> we cannot find any  $p \in Proc$  such that  $q_1 \equiv_p q_2$  and  $p \notin \bigcap_{i=1}^n \overline{dom}(a_i)$ : Let  $q_1 \xrightarrow{a} q'_1$  and  $q_2 \xrightarrow{a} q'_2$  such that  $q_1 \equiv_{dom(a)} q_2$ . From (\*), we have that  $dom(a) \subseteq \bigcap_{i=1}^n \overline{dom}(a_i)$  for  $\{a_1, \dots, a_n\} = A(q_1, q_2)$ , so  $dom(a) \cap dom(a_i) = \emptyset$  for every  $a_i \in A(q_1, q_2)$ , which implies  $a \parallel a_i$ , for all  $i \in [1..n]$ . Applying ID and FD for the path connecting  $q_1$  and  $q_2$  with the actions of  $A(q_1, q_2)$ , we have that there exists also an undirected path containing all actions of  $A(q_1, q_2)$  between  $q'_1$  and  $q'_2$ , so  $q'_1 \equiv_{\bigcap_{i=1}^n \overline{dom}(a_i)} q'_2$  by applying DAA<sub>1</sub> (and transitivity). This means that indeed the application of DAA<sub>2</sub> does not provide any new equivalence between  $q'_1$  and  $q'_2$  that was not already present after exhausting the application of DAA<sub>1</sub>.
4. From the last two items, we obtain also ‘ $\subseteq$ ’ of (5.9).

Having now (5.9), it is easy to prove that DAA<sub>3</sub> and DAA<sub>4</sub> from Theorem 3.32 hold (and thus finishing the proof).

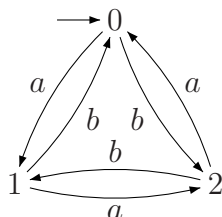


Figure 5.3: A *cyclic* transition system satisfying ID, FD, and BD that is not isomorphic to an asynchronous automaton

**DAA<sub>3</sub>** : By contradiction, assume that there exist  $q_1 \neq q_2$  such that  $\text{equiv}(q_1, q_2) = \text{Proc}$ . Then, from (5.9), there exists  $\{a_1, \dots, a_n\} \subseteq \Sigma$  such that  $\bigcap_{i=1}^n \overline{\text{dom}}(a_i) = \text{equiv}(q_1, q_2) = \text{Proc}$ , which implies that  $\overline{\text{dom}}(a_i) = \text{Proc}$  for any  $a_i$ . This is a contradiction with the fact that for any  $a \in \Sigma$ ,  $\text{dom}(a) \neq \emptyset$ .

**DAA<sub>4</sub>** : Proof similar to the item 3. above (i.e., ID and FD are applied).  $\square$

**Example 5.11** The acyclicity condition is essential in Theorem 5.10 as showed by the following (counter)example. We choose  $\Sigma = \{a, b\}$ , a distribution such that  $a \parallel b$ , and the transition system with three states  $\{0, 1, 2\}$  and transitions:

$$0 \xrightarrow{a} 1 \xrightarrow{a} 2 \xrightarrow{a} 0 \xrightarrow{b} 2 \xrightarrow{b} 1 \xrightarrow{b} 0$$

(see Figure 5.3). Then, the transition system is cyclic, deterministic, and fulfills the diamond properties ID, FD and BD, but there is no asynchronous automaton isomorphic to it (proof similar to the one for Example 3.23 – see also Remark 3.31).

We can use now Theorem 5.10 to construct a deterministic asynchronous automaton accepting a given *finite* prefix-closed forward-closed trace-closed language  $L$  (cf. Corollary 3.64). To do this, we easily construct a transition system  $TS$  accepting  $L$  such that  $TS$  satisfies ID, FD, and BD. This is realized by starting in an initial state and constructing a tree according to the words in  $L$  and identifying nodes of the tree only for reasons of determinism or diamond rules. More formally,  $TS$  will have as states the elements of  $[L] = \{[u] \mid u \in L\}$  with  $[\varepsilon]$  as initial state and  $[u] \xrightarrow{a} [ua]$  as transitions (for  $u \in L$  and  $a \in \Sigma$  such that  $u, ua \in L$ ) – cf. [DR95].

According to Theorem 5.10,  $TS$  is isomorphic to an asynchronous automaton which can be then constructed as mentioned in Section 5.2.1.

Since the new diamond rule **BD** is a sufficient condition, but not a necessary one, the size of the constructed  $TS$  is not necessarily optimal. An unfavorable situation is when there is not much independence in the system (**BD** does not allow transitions labeled by dependent actions to have the same outgoing state). So the method in this section is suitable for systems with a higher degree of concurrency.

### Asynchronous automata for conflict-free specifications

Another special case that we consider are the *conflict-free* specifications, i.e., transition systems in which the enabled actions in each state are pairwise independent:

**Definition 5.12 (Conflict-freeness)**

Given a distribution  $(\Sigma, Proc, \Delta)$ , a transition system  $TS = (Q, \Sigma, \rightarrow, I)$  is said to be *conflict-free* if, for all  $q \in Q$ ,  $a, b \in \Sigma$ :

$$q' \xleftarrow{a} q \xrightarrow{b} q'' \wedge a \neq b \Rightarrow a \parallel b.$$

Assuming conflict-freeness, FD allows the following generalization:

**Lemma 5.13** *Let  $TS$  be a conflict-free deterministic transition system satisfying FD. Then, if  $q_1, q_2, q_3$  are three states of  $TS$ ,  $b \in \Sigma$  an action, and  $w \in \Sigma^*$  a word such that  $q_2 \xleftarrow{w} q_1 \xrightarrow{b} q_3$  and  $b \notin \Sigma(w)$ , then there exists a state  $q_4$  such that  $q_2 \xrightarrow{b} q_4 \xleftarrow{w} q_3$  and  $\forall a \in \Sigma(w) : a \parallel b$ .*

**Proof.** Easy by induction on the length of  $w$  using the conflict-freeness together with the FD property.  $\square$

Using Remark 5.1, we can show that from conflict-free deterministic specification we can synthesize (modulo language equivalence) asynchronous automata that are *linear* in the size of the specification:

**Theorem 5.14** *For a distribution  $(\Sigma, Proc, \Delta)$ , any transition system  $TS$  that is deterministic, conflict-free, and satisfies ID and FD is language-equivalent to an asynchronous automaton over  $\Delta$  linear in the size of the specification.*

**Proof.** Let  $TS = (Q, \Sigma, \rightarrow, \{q_0\})$  be a transition system satisfying the hypothesis. We show that the least family of equivalences  $(\equiv_k)_{k \in Proc}$  over the states of  $TS$  satisfying DAA<sub>1</sub> and DAA<sub>2</sub>, also satisfies DAA<sub>4</sub>. Then, according to Remark 5.1, there exists an asynchronous automaton  $\mathcal{AA}$  linear in the size of  $TS$  and  $\Delta$  such that  $L(\mathcal{AA}) = L(TS)$ .

Let  $(\equiv_k)_{k \in Proc}$  be the least family of equivalences over  $Q$  satisfying *only* DAA<sub>1</sub>. We will show that:

- a) the family  $(\equiv_k)_{k \in Proc}$  is equal to the least family of equivalences satisfying *both* DAA<sub>1</sub> and DAA<sub>2</sub>, and
- b) DAA<sub>4</sub> also holds.

The proof is split over several steps, fixing two states  $q \neq q'$  and an action  $a \in \Sigma$ .

To ease presentation, we denote  $(\leftarrow \cup \rightarrow)^*$  by  $\leftrightarrow^*$ . More precisely, for two states  $q, q'$  and a word  $w = b_1 b_2 \dots b_n \in \Sigma^*$  with  $b_i \in \Sigma$  for  $i \in [1..n]$ , we have

$q \xleftrightarrow{w} q'$  iff there exist the states  $q_1, \dots, q_n$  such that  $q = q_1$ ,  $q_n = q'$ , and for each  $i \in [1..n - 1]$ , either  $q_i \xrightarrow{b_i} q_{i+1}$  or  $q_i \xleftarrow{b_i} q_{i+1}$ .

**Step 1 :** Since  $(\equiv_k)_{k \in Proc}$  was generated using DAA<sub>1</sub> (and transitivity), it is easy to see that the set of processes on which  $q$  and  $q'$  are equivalent is obtained by accumulating the processes on the paths connecting  $q$  and  $q'$  in the following way:

$$equiv(q, q') = \bigcup_{q \xleftrightarrow{w} q'} \bigcap_{b \in \Sigma(w)} \overline{dom}(b) \quad (5.10)$$

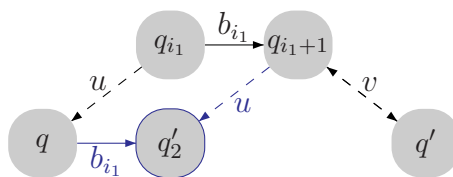


Figure 5.4: Details for Step 2 of the proof of Theorem 5.14

Consequently, if  $q \equiv_{\text{dom}(a)} q'$ , then for each  $k \in \text{dom}(a)$ , there exists a word  $w_k \in \Sigma^*$  such that  $q \overset{w_k}{\rightsquigarrow} q'$  and  $k \in \bigcap_{b \in \Sigma(w_k)} \overline{\text{dom}(b)}$ .

We choose each  $w_k$  to be a word  $w$  of minimal length with the property  $q \overset{w}{\rightsquigarrow} q'$  and  $k \in \bigcap_{b \in \Sigma(w)} \overline{\text{dom}(b)}$ . We remark that none of letters of  $w_k$  is equal to  $a$ , because  $k \notin \overline{\text{dom}(a)}$ . Moreover, the intermediary states in the sequence  $q \overset{w_k}{\rightsquigarrow} q'$  are pairwise distinct, otherwise  $w_k$  would not be of minimal length.

**Step 2 :** For each  $k \in \text{dom}(a)$ , we prove that there exists a state  $p_k$  such that  $q \xrightarrow{w_{kr}} p_k \xleftarrow{w_{kl}} q'$  where  $w_{kr} = b_{ki_1} \dots b_{ki_{n_r}}$  and  $w_{kl} = b_{kj_1} \dots b_{kj_{n_l}}$  are the words composed of the letters labeling the ‘right’  $\rightarrow$ , respectively ‘left’  $\leftarrow$  transitions in the sequence  $q \overset{w_k}{\rightsquigarrow} q'$  (preserving the ordering on letters). (For example, if  $q \overset{w_k}{\rightsquigarrow} q'$  is  $q = q_1 \xrightarrow{b_1} q_2 \xleftarrow{b_2} q_3 \xrightarrow{b_3} q_4 \xrightarrow{b_4} q_5 \xleftarrow{b_5} q_6 = q'$ , then we can prove that there exists a state  $p_k$  such that  $q \xrightarrow{w_{kr}} p_k \xleftarrow{w_{kl}} q'$  where  $w_{kr} = b_1 b_3 b_4$  and  $w_{kl} = b_2 b_5$ .)

The proof goes by *induction* on the length of  $w_{kr}$ :

*Base case :* If  $|w_{kr}| = 0$ , we just take  $p_k := q$  and  $w_{kl} := w_k$  and we have  $q = p_k \xleftarrow{w_k} q'$ .

*Induction step :* We assume that the property holds for all  $q \overset{w_k}{\rightsquigarrow} q'$  with  $|w_{kr}| = n$  and prove that it holds also for  $|w_{kr}| = n + 1$ .

Let  $b_{i_1}$  be the first letter of  $w_k$  that is the label of a ‘right’  $\rightarrow$  transition. That means that there exist two words  $u, v \in \Sigma^*$  and a state  $q_{i_1}$  such that  $w_k = u b_{i_1} v$  and  $q = q_1 \xleftarrow{u} q_{i_1} \xrightarrow{b_{i_1}} q_{i_1+1} \overset{v}{\rightsquigarrow} q'$ . Then, we use Lemma 5.13 to show that there exists a state  $q'_2$  such that  $q \xrightarrow{b_{i_1}} q'_2 \xleftarrow{u} q_{i_1+1} \overset{v}{\rightsquigarrow} q'$  (see Figure 5.4).

In order to use Lemma 5.13, we must have that  $b_{i_1} \notin \Sigma(u)$ . By contradiction, suppose there exist  $u', u'' \in \Sigma^*$  and two states  $s, s'$  such that  $u = u' b_{i_1} u''$  with  $b_{i_1} \notin \Sigma(u')$  and  $q \xleftarrow{u''} s' \xleftarrow{b_{i_1}} s \xleftarrow{u'} q_{i_1}$ . (Since  $TS$  is deterministic, necessarily  $|u'| \neq 0$ !) Applying Lemma 5.13 to  $s \xleftarrow{u'} q_{i_1} \xrightarrow{b_{i_1}} q_{i_1+1}$ , there exists a state  $s''$  such that  $s \xrightarrow{b_{i_1}} s'' \xleftarrow{u'} q_{i_1+1}$ . Since  $TS$  is deterministic and we have  $s \xrightarrow{b_{i_1}} s'$ , we deduce that  $s'' = s'$  and therefore  $s' \xleftarrow{u'} q_{i_1+1}$ . So we obtain a sequence of transitions connecting  $q$  and  $q'$  as  $q \xleftarrow{u''} s' \xleftarrow{u'} q_{i_1+1} \overset{v}{\rightsquigarrow} q'$  with  $|u'' u' v| < |u'' b_{i_1} u' b_{i_1} v| = |w_k|$  and this is a contradiction with the minimality of  $w_k$ .



We also remark that set of states in the sequences  $q'_2 \xleftarrow{u} q_{i_1+1}$  and  $q_{i_1+1} \xleftarrow{v} q'$  have only  $q_{i_1+1}$  in common, otherwise the minimality of  $w_k$  is not fulfilled.

Finally, we apply the induction hypothesis to the states  $q'_2$ ,  $q'$  and the word  $w'_k := uv$  as a word of minimal length with the property that  $q'_2 \xleftarrow{w'_k} q'$  and  $k \in \bigcap_{b \in \Sigma(w'_k)} \overline{\text{dom}}(b)$ . So there exists a state  $p'_k$  such that  $q'_2 \xrightarrow{w'_{kr}} p'_k \xleftarrow{w'_{kl}} q'$ . We finish the induction step by choosing  $p_k := p'_k$ ,  $w_{kr} := b_{i_1} w'_{kr}$ , and  $w_{kl} := w'_{kl}$ .

**Step 3 :** Suppose now we have a situation where for a state  $p$  and each  $k \in \text{dom}(a)$ , there is a path  $p \xrightarrow{u_k} p_k \xrightarrow{a} p'_k$  such that  $k \in \bigcap_{b \in \Sigma(u_k)} \overline{\text{dom}}(b)$ . We prove that there exists one process  $k \in \text{dom}(a)$  such that  $\forall b \in \Sigma(u_k) : a \parallel b$ .

The proof will be by induction on the measure  $M := \sum_{k \in \text{dom}(a)} |u_k|$ .

*Base case :* If  $M = 0$ , there is nothing to prove.

*Induction step :* Supposing that the property holds for all  $M \leq n$  (with  $n$  a natural number), we want to prove that it holds also for  $M = n + 1$ .

First, if there exists  $k_0 \in \text{dom}(a)$  such that  $|u_{k_0}| = 0$ , then obviously  $\forall b \in \Sigma(u_{k_0}) : a \parallel b$  is true, so  $k_0$  is the  $k$  needed. Hence, from now on we assume that  $|u_k| > 0$ , for all  $k \in \text{dom}(a)$ .

Let us fix  $k_0 \in \text{dom}(a)$  and let  $b$  be the first letter of  $u_{k_0}$ , so there exist  $u'_{k_0} \in \Sigma^*$  and a state  $p'$  such that  $u_{k_0} = bu'_{k_0}$  and  $p \xrightarrow{b} p' \xrightarrow{u'_{k_0}} p_k \xrightarrow{a} p'_k$ .

We will prove that  $b \parallel a$  and then we prove that we can apply the induction hypothesis for the node  $p'$ . First, for each  $k \in \text{dom}(a) \setminus \{k_0\}$ , we have that either  $b \in \Sigma(u_k)$  or  $b \notin \Sigma(u_k)$  (see Figure 5.5):

- If  $b \in \Sigma(u_k)$ , there exist the states  $q_k, q'_k$  and  $u'_k, u''_k \in \Sigma^*$  such that  $u_k = u'_k b u''_k$  with  $b \notin \Sigma(u'_k)$  and  $p \xrightarrow{u'_k} q_k \xrightarrow{b} q'_k \xrightarrow{u''_k} p_k \xrightarrow{a} p'_k$ . We can then apply Lemma 5.13 to  $p' \xleftarrow{b} p \xrightarrow{u'_k} q_k$  and we obtain that there exists a state  $s$  such that  $p' \xrightarrow{u'_k} s \xleftarrow{b} q_k$ . Since  $TS$  is deterministic and we have both  $q_k \xrightarrow{b} q'_k$  and  $q_k \xrightarrow{b} s$ , then  $s = q'_k$ , so  $p' \xrightarrow{u'_k} q'_k$  and we have a path  $p' \xrightarrow{u'_k} q'_k \xrightarrow{u''_k} p_k$ .
- If  $b \notin \Sigma(u_k)$ , then we can apply Lemma 5.13 to  $p' \xleftarrow{b} p \xrightarrow{u_k} p_k$  and we have that there exists a state  $q_k$  such that  $p' \xrightarrow{u_k} q_k \xleftarrow{b} p_k$ . From  $b \in \Sigma(u_{k_0})$ , we have  $k_0 \in \overline{\text{dom}}(b)$ , so  $a \neq b$  (otherwise  $k_0 \in \overline{\text{dom}}(b) = \overline{\text{dom}}(a)$  and also  $k_0 \in \text{dom}(a)$ , which is a contradiction). Then, from  $q_k \xleftarrow{b} p_k \xrightarrow{a} p'_k$  and  $a \neq b$ , applying the conflict-freeness of  $TS$ , we have that  $a \parallel b$ . Next, applying the FD rule, there exists a state  $q'_k$  such that  $q_k \xrightarrow{a} q'_k \xleftarrow{b} p'_k$ .

We can show now that  $b \parallel a$ : If there exists  $k \in \text{dom}(a)$  such that  $b \notin \Sigma(u_k)$ , then we already showed in the corresponding case above that  $b \parallel a$ . Otherwise, we have that  $\forall k \in \text{dom}(a) : b \in \Sigma(u_k)$ . That implies that  $\forall k \in \text{dom}(a) : k \in \overline{\text{dom}}(b)$ , so  $\text{dom}(a) \subseteq \overline{\text{dom}}(b)$  and this is equivalent to  $a \parallel b$  (by definition, because  $\text{dom}(a) \cap \text{dom}(b) = \emptyset$ ). In any case,  $b \parallel a$ .



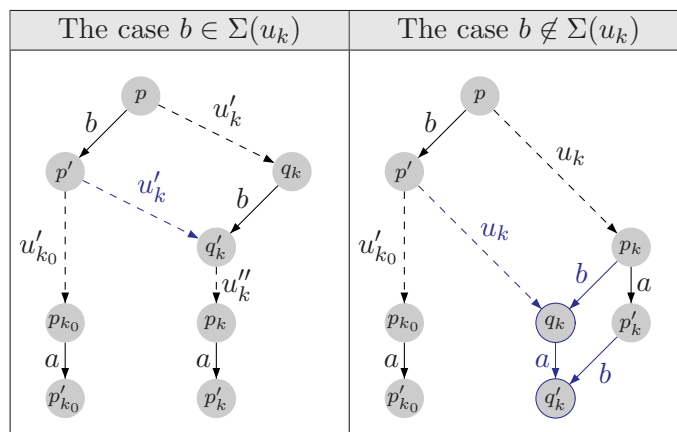


Figure 5.5: Details for Step 3 of the proof of Theorem 5.14

We prove now that we can apply the induction hypothesis to  $p'$ : First, the state  $p'$  has the property that for each  $k \in \text{dom}(a)$ , there exists a path of the form  $p' \xrightarrow{v_k} s_k \xrightarrow{a} s'_k$  such that  $k \in \bigcap_{b \in \Sigma(v_k)} \overline{\text{dom}(b)}$ . If  $k = k_0$ , we choose  $s_k := p_{k_0}$ ,  $s'_k := p'_{k_0}$ , and  $v_k := u'_{k_0}$ . If  $k \neq k_0$ , we have the two cases (see again Figure 5.5): If  $b \in \Sigma(u_k)$ , we choose  $s_k := p_k$ ,  $s'_k := p'_k$ , and  $v_k := u'_k u''_k$ , otherwise we choose  $s_k := q_k$ ,  $s'_k := q'_k$ , and  $v_k := u_k$ . We notice that  $|v_{k_0}| = |u'_{k_0}| < |u_{k_0}|$  and for  $k \neq k_0$ ,  $|v_k| \leq |u_k|$  and also that  $k \in \bigcap_{b \in \Sigma(v_k)} \overline{\text{dom}(b)}$  (by construction).

So we can apply the induction hypothesis to the state  $p'$  and deduce that there exists  $k \in \text{dom}(a)$  such that  $\forall b \in \Sigma(v_k) : a \parallel b$ . Using also the fact that  $b \parallel a$  proved above, we see that indeed  $\forall b \in \Sigma(u_k) : a \parallel b$ .

**Step 4** : We are able now to prove that:

(\*) if  $q \equiv_{\text{dom}(a)} q'$  and  $q \xrightarrow{a} q''$ , then there exists a sequence of transitions  $\rightarrow$  or  $\leftarrow$  connecting  $q$  and  $q'$  labeled with actions independent to  $a$ .

Formally, (\*) says that there exists  $w \in \Sigma^*$  such that  $q \xleftrightarrow{w} q'$  and  $\forall b \in \Sigma(w) : a \parallel b$ .

First, Step 1 chose a minimal  $w_k$  for each  $k \in \text{dom}(a)$  such that  $q \xleftrightarrow{w_k} q'$  and  $k \in \bigcap_{b \in \Sigma(w_k)} \overline{\text{dom}(b)}$ . Step 2 proved then that for each  $k \in \text{dom}(a)$  there exists a state  $p_k$  such that  $q \xrightarrow{w_{kr}} p_k \xleftarrow{w_{kl}} q'$  and  $k \in \bigcap_{b \in \Sigma(w_{kr})} \overline{\text{dom}(b)} \cap \bigcap_{b \in \Sigma(w_{kl})} \overline{\text{dom}(b)}$ . Using Lemma 5.13 (this is possible, since it can be easily shown that  $a \notin \Sigma(w_{kr})$ ), we have that there exists  $p'_k$  such that  $q'' \xrightarrow{w_{kr}} p'_k \xleftarrow{a} p_k$  and  $\forall b \in \Sigma(w_{kr}) : a \parallel b$ . We are then able to apply Step 3, with  $p := q'$  and  $p_k := p_k$ ,  $p'_k := p'_k$ , and  $u_k := w_{kl}$ , so we have that there exists a process  $k_0 \in \text{dom}(a)$  such that  $\forall b \in \Sigma(w_{k_0l}) : a \parallel b$ . Thus there exists  $w \in \Sigma^*$ , namely  $w := w_{k_0r} w_{k_0l}$ , such that  $q \xleftrightarrow{w} q'$  and  $\forall b \in \Sigma(w) : a \parallel b$ .

Furthermore, since  $p'_{k_0} \xleftarrow{a} p_{k_0} \xleftarrow{w_{k_0l}} q'$  and  $\forall b \in \Sigma(w_{k_0l}) : a \parallel b$ , it is easy to show by induction using property ID that there exists  $q'''$  such that  $p'_{k_0} \xleftarrow{w_{k_0l}} q''' \xleftarrow{a} q'$ .

**Step 5** : Let  $(\equiv_k)_{k \in \text{Proc}}$  be the least family of equivalences over the states of  $TS$  satisfying

DAA<sub>1</sub>. We prove that  $(\equiv_k)_{k \in Proc}$  is equal to the least family of equivalences satisfying both DAA<sub>1</sub> and DAA<sub>2</sub>.

Let  $q, q'$  be two states and  $a \in \Sigma$  an action such that  $q \equiv_{dom(a)} q'$  and there exist  $q'', q'''$  such that  $q \xrightarrow{a} q''$  and  $q' \xrightarrow{a} q'''$  (as in DAA<sub>2</sub>). Using the result of Step 4 for  $q \equiv_{dom(a)} q'$  and  $q \xrightarrow{a} q''$ , there exists  $w \in \Sigma^*$  such that  $q \xrightarrow{w} q'$  and  $\forall b \in \Sigma(w) : a \parallel b$ . In fact, we proved something more at Step 4, namely that there exists also a state  $p$  such that  $q' \xrightarrow{a} p$  and  $q'' \xrightarrow{w} p$ . Because  $TS$  is deterministic, we have that  $p = q'''$ , so  $q'' \xrightarrow{w} q'''$ . Now since  $q'' \xrightarrow{w} q'''$  and  $\forall b \in \Sigma(w) : a \parallel b$ , we have that  $dom(a) \subseteq \bigcap_{b \in \Sigma(w)} \overline{dom(b)}$  which implies  $q'' \equiv_{dom(a)} q'''$ , so DAA<sub>2</sub> is satisfied.

**Step 6** : Finally we prove that  $(\equiv_k)_{k \in Proc}$  satisfies also DAA<sub>4</sub>. Let  $q, q'$  be two states and  $a \in \Sigma$  an action such that  $q \equiv_{dom(a)} q'$  and there exists  $q''$  with  $q \xrightarrow{a} q''$ . Just similar to the proof in the previous step, there exists  $p$  such that  $q' \xrightarrow{a} p$  and this means exactly that DAA<sub>4</sub> is satisfied.  $\square$

The conflict-freeness can be translated into formal languages terms and use the above theorem in the following way. We say that a language  $L$  is *conflict-free* if:  $wa \in L$  and  $wb \in L$  and  $a \neq b$  implies  $a \parallel b$ . It is clear then that the minimal deterministic transition system  $TS$  accepting a conflict-free language  $L$  is conflict-free. Then, applying Theorem 5.14 to the minimal deterministic transition system  $TS$  accepting a conflict-free prefix-, forward-, and trace-closed language  $L$ , we can construct an asynchronous automaton  $\mathcal{AA}$  (linear in the size of  $TS$ ) such that  $L(\mathcal{AA}) = L$ .

In fact, we stumbled upon the conflict-free case in one of our attempts of finding alternatives to Zielonka's construction. The conflict-free case was supposedly one of the basic steps involved in the construction. Unfortunately, we failed in the general case. A somehow similar 'story' appears in [NT02], where the authors present a conjecture (in concurrency theory) that they were able to prove it only in the conflict-free case, which (similar to Theorem 5.14) does involve rather complicated proofs.

### An idea based on graph unfoldings

We make a short digression to present one attempt we had for an alternative to Zielonka's construction. The idea was to use Morin's characterization result for deterministic case, i.e., Theorem 3.32, to 'guide' a transformation of the initial (global) specification into a distributable one. At the end of our endeavor, we were only able to use this idea for the case of distributions that generate transitive dependence graph for actions (presented in the next subsection). Even if we were not successful, we give below a few hints on our attempt. We note that this idea inspired the heuristic presented in Section 6.3.3.

We sketch below how the intended algorithm looked like:

1. Given a distribution and a prefix-, forward-, and trace-closed regular specification  $L$ , construct the minimal deterministic transition system  $TS$  accepting  $L$ . According to Proposition 3.36,  $TS$  satisfies ID and FD.
2. Apply the test provided by Theorem 3.32 whether  $TS$  is isomorphic to an asynchronous automaton.

The distribution	The transition system	A distributable unfolding
$\Sigma = \{a, b, c\}$ $Proc = \{1, 2, 3\}$ $dom(a) = \{1, 2\}$ $dom(b) = \{1, 3\}$ $dom(c) = \{2, 3\}$		
$\Sigma = \{a, b\}$ $Proc = \{1, 2\}$ $dom(a) = \{1\}$ $dom(b) = \{2\}$ (so, $a \parallel b$ )		

Figure 5.6: Exemplifying unfolding using copies

3. If the result is positive, synthesize the asynchronous automaton and stop.
4. Otherwise, identify transitions involved in violations of the conditions of the characterization test and unfold *some* of them<sup>1</sup> in such a way that *going back* to step 2, the number of violations have strictly decreased (thus forcing the termination of the algorithm).

The correctness of the above algorithm is given by step 3 (and the fact that an unfolding preserves the accepted language), whereas the termination is given by the decreasing number of violations envisaged in the unfolding from the last step. As you may have guessed, the problem is to find a systematic unfolding that ‘solves’ violations eventually leading to a transition system that can be decomposed as an asynchronous automaton.

If a transition system does not pass the test of Theorem 3.32 (i.e., is not isomorphic to an asynchronous automaton), a list of pairs of states violating  $DAA_3$  or  $DAA_4$  is provided. Intuitively, the reason a transition system is not distributable is that there are *too many* connections or paths between the nodes and that implies that the equivalence classes w.r.t.  $(\equiv_k)_{k \in Proc}$  constructed by  $DAA_1$  and  $DAA_2$  are too large (causing the failure of  $DAA_3$  or  $DAA_4$ ). The idea of unfoldings is intended to decrease the number of states that are ‘too connected’ by *making copies* (two or more if necessary) of the states as showed by the two following examples.

The first example in Figure 5.6 (top), the initial transition system is *not* isomorphic to an asynchronous automaton: From  $DAA_1$  applied to the transitions  $0 \xrightarrow{b} 2$  and  $2 \xrightarrow{c} 0$ , we have that  $0 \equiv_{1,2} 2$ , so  $0 \equiv_{dom(a)} 2$ . This together with the existence of transition  $0 \xrightarrow{a} 1$  violates  $DAA_4$ , because there is no state  $q$  such that  $2 \xrightarrow{a} q$ . The cause of problem is that the states 0 and 2 are linked by two transitions labelled  $b$ , respectively  $c$ . A possible solution arises by unfolding the transition  $2 \xrightarrow{c} 0$  as follows: We make a

<sup>1</sup>In a similar way that a graph is unfolded into a tree.

copy of all states and transitions and we ‘cross’ the copies of the transitions  $2 \xrightarrow{c} 0$  and  $2' \xrightarrow{c} 0'$  as in Figure 5.6. The new transition system obtained by ‘unfolding’ the original transition  $0 \xrightarrow{a} 2$  is now isomorphic to an asynchronous automaton. (Now the states 0 and 2 – as well as their twins  $0'$  and  $2'$  – are equivalent on process 2 because of the  $b$ -labeled transitions, but not on process 1, thus avoiding the violation of  $\text{DAA}_4$ ).

In the second example in Figure 5.6 (bottom), the initial transition system is *not* isomorphic to an asynchronous automaton because  $\text{DAA}_3$  is violated: The states 0 and 1 are equivalent on both available processes, using  $\text{DAA}_1$  (see also Remark 3.31). Since (the global state space of) any asynchronous automaton satisfies ID and FD, we must consistently apply the unfolding (implemented by making copies) such that the diamond rules are preserved. Since  $a$  and  $b$  are independent, we simultaneously ‘cross’ all the  $b$ -labeled in the process of duplication. The result, which will be isomorphic to an asynchronous automaton, is showed in Figure 5.6.

Unfortunately, we were not able to make the above idea (or a modification of it) feasible in practice (the main problems were created by the attempt to systematically preserve the diamonds). The intended correctness proof of such construction would have been of course by induction on the number of duplications/unfoldings involved together with the observation that at each step the language is preserved. A sanity check shows that the size of the transition system obtained by the method is not *too small*<sup>1</sup>: The size is proportional to  $|Q| \cdot 2^{|T|}$  where  $|Q|$  the number of states of the initial transition system and  $|T|$  the number of transitions to be unfolded (every unfolding step doubles the number of states).

However, we managed to work out a proof in the special case that we present next.

### Asynchronous automata for transitive dependence relations

We give a new construction for the synthesis of asynchronous automata in case the distribution from the specification generates a *transitive* dependence relation between actions (i.e.,  $\forall a, b, c \in \Sigma : a \parallel b \wedge b \parallel c \Rightarrow a \parallel c$ ). First we give a construction for the case where all actions are dependent (i.e.,  $\forall a, b \in \Sigma : a \parallel b$ ) and use this in the more general case of *transitive* dependence relations.

**Complete unfolding** As the main ingredient of this subsection, we define the ‘complete unfolding’ of a transition system  $TS := (Q, \Sigma, \rightarrow, I)$  as the transition system obtained by iteratively duplicating the state space of  $TS$  for each of its (non self-loop) transitions.

Formally, let  $T_d$  be the set of transition with *distinct* ‘in’ and ‘out’ states, i.e.,

$$T_d := \{q_1 \xrightarrow{a} q_2 \mid q_1 \neq q_2\}.$$

Then, we construct a new transition system  $Unf(TS) = (Q_{unf}, \Sigma, \rightarrow_{unf}, I_{unf})$ , called the *complete unfolding* of  $TS$ , as follows:

- $Q_{unf} := \{q^{(loc)} \mid q \in Q \wedge loc \in \{0, 1\}^{T_d}\}$

(The states of  $Unf(TS)$  are obtained indexing the states of  $Q$  by *location* functions, generating two copies (0/1) of a state  $q$  for each transition of  $T_d$ .)

<sup>1</sup>See a worst-case analysis on the size of a *deterministic* asynchronous automaton in [KMS94].

$$\blacksquare \rightarrow_{unf} := \left\{ \begin{array}{l} \left. \begin{array}{l} q_1^{(loc_1)} \xrightarrow{a} q_2^{(loc_2)} \\ \cup \left\{ q^{(loc)} \xrightarrow{a} q^{(loc)} \right\} \end{array} \right| \begin{array}{l} q_1 \xrightarrow{a} q_2 \in T_d \wedge \\ loc_2(t) = \begin{cases} loc_1(t), & \text{if } t \in T_d \setminus \{q_1 \xrightarrow{a} q_2\} \\ 1 - loc_1(t), & \text{if } t = q_1 \xrightarrow{a} q_2 \end{cases} \\ q \xrightarrow{a} q \in T \setminus T_d \wedge loc \in \{0, 1\}^{T_d} \end{array} \right\}$$

(Each transition of  $T_d$  will generate copies in  $\rightarrow_{unf}$  such that the locations of the ‘in’ and ‘out’ states will differ only on the position corresponding to the given transition.)

$$\blacksquare I_{unf} := \{q^{(loc_0)} \mid q \in I \wedge \forall t \in T_d : loc_0(t) = 0\}$$

(The initial states of the unfolding are the initial states in the ‘copy’ of the state space corresponding to the nil location function  $loc_0$ .)

The state space of  $Unf(TS)$  is linear in the state space of  $TS$  and exponential in the number of non self-loop transitions of  $TS$ , i.e.,  $|Q_{unf}| = |Q| \cdot 2^{|T_d|}$ . Note also that if  $TS$  is deterministic, then  $Unf(TS)$  is deterministic too.

The main result is that the complete unfolding  $Unf(TS)$  is isomorphic to an asynchronous automaton over distributions generating empty independence relations between actions. The proof is based on the characterization of Theorem 3.30.

**Theorem 5.15** *Given a finite transition system  $TS$ , we have that  $Unf(TS)$  is bisimilar to  $TS$  and therefore accepting the same language.*

*Moreover, for any distribution  $(\Sigma, Proc, \Delta)$  generating an empty independence relation (i.e.,  $\parallel = \emptyset$ ),  $Unf(TS)$  is isomorphic to an asynchronous automaton over  $\Delta$ .*

**Proof.** First, it is easy to check that  $R \subseteq Q_{unf} \times Q$  defined by

$$R := \{ (q^{(loc)}, q) \mid q \in Q \wedge loc \in \{0, 1\}^{T_d} \}$$

is a bisimulation between  $Unf(TS)$  and  $TS$ . From bisimilarity, we deduce that  $Unf(TS)$  and  $TS$  accept the same language (cf. Lemma 4.34).

We use now Theorem 3.30 to prove that  $Unf(TS)$  is isomorphic to an asynchronous automaton over  $\Delta$ . For that, it is enough to construct a family of equivalences satisfying  $AA_1$ – $AA_3$  from Theorem 3.30.

We choose a family of binary relations  $(\equiv_k)_{k \in Proc}$  with  $\equiv_k \subseteq Q_{unf} \times Q_{unf}$  as follows. First, let

$$\equiv_k := \left\{ \left( q_1^{(loc_1)}, q_2^{(loc_2)} \right) \mid \exists a \in \Sigma : q_1^{(loc_1)} \xrightarrow{a} q_2^{(loc_2)} \in \rightarrow_{unf} \wedge k \in \overline{dom}(a) \right\},$$

where (similar to the previous subsections),  $\overline{dom}(a) := Proc \setminus dom(a)$ . We take the reflexive and transitive closures of  $(\equiv_k)_{k \in Proc}$  and we denote them, without danger of confusion, also by  $(\equiv_k)_{k \in Proc}$ . We prove that the family of equivalences  $(\equiv_k)_{k \in Proc}$  satisfies  $AA_1$ – $AA_3$ .

For  $q, q' \in Q_{unf}$  we denote by  $equiv(q, q')$  the maximal subset  $K \subseteq Proc$  such that  $q \equiv_K q'$ , i.e.,

$$equiv(q, q') := \{k \in Proc \mid q \equiv_k q'\}.$$

We start making some observations on  $equiv$ :

- First, for any  $q, q' \in Q_{unf}$  it is not hard to prove that:

$$equiv(q, q') = \bigcup_{q \overset{w}{\leftrightarrow} q'} \bigcap_{a \in \Sigma(w)} \overline{dom}(a) \quad (5.11)$$

where the notation  $q \overset{w}{\leftrightarrow} q'$  is given on page 142.

- For two states in different locations, i.e.,  $q_1^{(loc_1)}, q_2^{(loc_2)} \in Q_{unf}$  with  $loc_1 \neq loc_2$ , let  $t := q \xrightarrow{a} q' \in T_d$  such that  $loc_1(t) \neq loc_2(t)$ . Using (5.11) and the construction of  $\rightarrow_{unf}$  (more precisely, the fact that a transition labeled by  $a$  will appear on any path from  $q_1^{(loc_1)}$  to  $q_2^{(loc_2)}$ ) we have that:

$$equiv(q_1^{(loc_1)}, q_2^{(loc_2)}) \subseteq \overline{dom}(a) \quad (5.12)$$

- For two distinct states in the same location, i.e.,  $q_1^{(loc)}, q_2^{(loc)} \in Q_{unf}$  with  $q_1 \neq q_2$ , it is not difficult to see that there is no path connecting them. Then, using (5.11), we have that:

$$equiv(q_1^{(loc)}, q_2^{(loc)}) = \emptyset \quad (5.13)$$

Using the above observations, we are now able to prove that indeed the equivalences  $(\equiv_k)_{k \in Proc}$  satisfy the three conditions of Theorem 3.30:

**AA<sub>1</sub>** : Directly from the way  $(\equiv_k)_{k \in Proc}$  were defined.

**AA<sub>2</sub>** : It is enough to show that  $\forall q_1, q_2 \in Q_{unf} : q_1 \neq q_2 \Rightarrow equiv(q_1, q_2) \neq Proc$  (cf. Remark 4.6). We have two cases. First, if  $q_1$  and  $q_2$  have the same location, then by (5.13) we have that  $equiv(q_1, q_2) = \emptyset \neq Proc$ . Second, if  $q_1$  and  $q_2$  have different locations, then by (5.12) there exists an action  $a \in \Sigma$  such that  $equiv(q_1, q_2) \subseteq \overline{dom}(a)$ . Because  $dom(a) \neq \emptyset$  (by definition), then  $\overline{dom}(a) \neq Proc$  and hence  $equiv(q_1, q_2) \neq Proc$ .

**AA<sub>3</sub>** : For  $q_1 = q_2$  there is nothing to prove. If  $q_1 \neq q_2$ , as before, we have two cases. First, if  $q_1$  and  $q_2$  have the same location, then by (5.13) we have that  $equiv(q_1, q_2) = \emptyset$  and therefore the hypothesis that  $q_1 \equiv_{dom(a)} q_2$  is false (because  $dom(a) \neq \emptyset$ ), so **AA<sub>3</sub>** is true. Second, if  $q_1$  and  $q_2$  have different locations, then by (5.12) there exists an action  $b \in \Sigma$  such that  $equiv(q_1, q_2) \subseteq \overline{dom}(b)$ . The hypothesis that  $q_1 \equiv_{dom(a)} q_2$  is false again, otherwise  $dom(a) \subseteq equiv(q_1, q_2) \subseteq \overline{dom}(b)$ , so  $dom(a) \cap dom(b) = \emptyset$  which implies  $a \parallel b$ , but this is not possible since  $\parallel = \emptyset$ . Since in either case the hypothesis is false, the implication of **AA<sub>3</sub>** is true.  $\square$

**Corollary 5.16** *Given a distribution  $(\Sigma, Proc, \Delta)$  generating a transitive dependence relation and a prefix-closed trace-closed language  $L \subseteq \Sigma^*$ , we can construct using the ‘complete unfolding’ idea an asynchronous automaton over  $\Delta$  accepting  $L$ .*

**Proof.** If the dependence relation  $\parallel \subseteq \Sigma \times \Sigma$  generated by  $\Delta$  is transitive, then the maximally connected components  $(\Sigma_i)_{i \in I}$  of the dependence graph  $(\Sigma, \parallel)$  form a partition of  $\Sigma$  with the property that



- $\forall i \in I, a, a' \in \Sigma_i : a \not\parallel a'$  (i.e., each  $\Sigma_i$  is a clique) and
- $\forall i \neq j \in I, a_i \in \Sigma_i, a_j \in \Sigma_j : a_i \parallel a_j$  (there are no ‘connections’ in  $(\Sigma, \parallel)$  between the components  $\Sigma_i$  because of the transitivity property).

The first step in our construction is to take the projections  $(L_i)_{i \in I}$  of  $L$  on the partition  $(\Sigma_i)_{i \in I}$ . I.e., for each  $i \in I$ ,  $L_i := L \upharpoonright_{\Sigma_i}$ . Using Propositions 2.1 and 3.7, the projections  $L_i$  are also prefix-closed trace-closed languages and we can construct a family of finite transition systems  $(TS_i)_{i \in I}$  such that  $\forall i \in I : L(TS_i) = L_i$ .

Further, we construct the ‘complete unfoldings’  $Unf(TS_i)$  for each  $TS_i$ . According to the first part of Theorem 5.15,  $L(Unf(TS_i)) = L(TS_i) = L_i$ . Moreover, since  $\forall i \in I, a, a' \in \Sigma_i : a \not\parallel a'$ , we have that the *projection* of the distribution  $\Delta$  on  $\Sigma_i$ , denoted by  $\Delta_i$ , generates an empty independence relation between actions, so we can apply Theorem 5.15 and show that for each  $i \in I$ ,  $Unf(TS_i)$  is isomorphic to an asynchronous automaton  $\mathcal{AA}_i$  over the projection  $\Delta_i$ . The asynchronous automaton  $\mathcal{AA}_i$  is constructed as mentioned in Section 5.2.1.

Finally, we combine all the asynchronous automata  $(\mathcal{AA}_i)_{i \in I}$  over  $\Delta_i$  into a single one, denoted  $\mathcal{AA}$ , over the distribution  $\Delta$ , putting them in parallel. This is possible (and correct) because the dependence graph  $(\Sigma, \parallel)$  is transitive, so according to an observation at the beginning of the proof, actions belonging to different partitions  $\Sigma_i$  and  $\Sigma_j$  are independent, which implies that there is *no common process* for any pair of distinct projection distributions  $\Delta_i$  and  $\Delta_j$ . It is easy to see that  $L(\mathcal{AA}) = L$ .

We remark that the above approach can also be used to synthesize *deterministic* asynchronous automata. The transition systems  $(TS_i)_{i \in I}$  can be the minimal deterministic transition systems accepting the languages  $(L_i)_{i \in I}$ . Since the *Unf*-transformation preserves the determinism, we have that all  $\mathcal{AA}_i$  constructed are deterministic, and this further implies that the final  $\mathcal{AA}$  is also deterministic.  $\square$

The construction we presented is not necessarily an optimal one. It was rather used to exemplify a possible implementation. Alternatives to Zielonka’s construction for transitive dependence relations can be obtained by combining approaches from the literature. For instance, since a transitive dependence relation is a particular case of triangulated dependence relation, [DM96] (generalizing [Mét87]) provides a simpler (more precisely, polynomial-time) solution than Zielonka’s construction. Moreover, minimum deterministic asynchronous automata were synthesized in [BPS94] for transitive dependence relations. However, in the above approaches only a *specific* distribution generating the given transitive dependence relation is considered (recall that, to a given relation of (in)dependence between actions, one can associate usually more than one compatible distribution – see Section 2.1.4): The distribution is given in [DM96] by a ‘perfect vertex elimination scheme’, while in [BPS94] by the maximal cliques of the dependence relation. In our construction, we obtain in a rather neat way an asynchronous automaton over *any* distribution compatible with the dependence graph.<sup>1</sup>

<sup>1</sup>The approaches of [DM96] and [BPS94] can be complemented by the construction from [CSLR88] that transform an asynchronous automaton over a given distribution  $\Delta$  to an asynchronous automaton over *any* other distribution  $\Delta'$  generating the same independence relation. The translation of [CSLR88] may involve an exponential (in the number of processes of  $\Delta'$ ) blowup.

## Discussion

In this chapter we presented synthesis algorithms of distributed transition system from implementable specifications. Whereas the synthesis of synchronous products poses no difficulties [CMT99, Muk02], a lot of effort have been put over the last almost twenty years in the synthesis of (deterministic) asynchronous automata [Dub86, Zie87, CSLR88, Pig93a, Pig93b, BPS94, KMS94, MS94, DR95, DM96, Mor98, SEM03]. Unfortunately, Zielonka's expensive (and intricate) approach is the only available method at present in the general case. In Section 5.2.2, we present the ingredients of the construction together with a modification to accommodate the prefix-closed case (i.e., all states of the implementation are accepting). In Section 5.2.3, we give alternatives to Zielonka's construction for special cases (finite specifications, conflict-free specifications, distributions generating transitive dependence relations).

In the next chapter, we present implementations for some of the algorithms of this chapter together with a couple of case studies and heuristics against the state space explosion problem.

During the preparation of the final version of this thesis, we found out about the (yet) unpublished manuscript [GM06] showing an *improvement* of Zielonka's construction for *deterministic* asynchronous automata reducing the double-exponential in the number of processes to a single exponential (more precisely, the size of the synthesized deterministic asynchronous automaton is  $2^{O(|\mathcal{A}|^2 \times |Proc|^3)}$ , where  $\mathcal{A}$  is the automaton describing the global specification and  $Proc$  the set of processes in the distribution). The authors of [GM06] construct asynchronous automata equipped with global *final* states, but they mention that their idea can be modified to work also in the framework of this thesis where no final states are considered. Although we have not experimented yet with this new approach, we note that the *heuristics* to obtain smaller asynchronous automata that we propose in the next chapter (namely, Section 6.3.3) can be used in conjunction with [GM06].





---

Beware of bugs in the above code;  
I have only proved it correct, not  
tried it.

*Donald Knuth*

Beware of bugs in my implementation  
code; I have only tried it, not proved  
it correct.

*The author of this thesis*

## CHAPTER 6

# IMPLEMENTATIONS AND CASE STUDIES

---

IN this chapter we reach the final destination on the path from theory to practice, going in the direction pointed at the end of [KV01]: “we believe that the real challenge that synthesis algorithms and tools face in the coming years is mostly not that dealing with computational complexity, but rather that of making automatically synthesized systems more practically useful”. Although Zielonka’s asynchronous automata [Zie87] will soon celebrate a venerable age of twenty years and a large body of literature, we are not aware of any implementations for them (this must be due to the complexity of their synthesis procedure). In this chapter we report on a couple of (heuristic) implementations that synthesize asynchronous automata for specifications larger than Zielonka’s algorithm could tackle.

In our thesis, we address the problem of automatically synthesizing a finite state, closed distributed system from a given specification. Seminal papers in this area are [CE82, MW84], where synthesis algorithms from temporal logic specifications are developed. The algorithms are based on tableau procedures for the satisfiability problem of CTL (respectively LTL). These approaches suffer from the limitation that the synthesis algorithms produce a sequential process  $P$ , and not a distributed implementation. The solution suggested in these works is to first synthesize the sequential solution, and then decompose it. However, since distribution aspects like concurrency and independency of actions are not part of the CTL or LTL specification, the solution may be impossible to distribute while keeping the intended concurrency. As showed in Section 6.1.1, this is in fact what happens with the solution of [CE82] to the mutual exclusion problem, the first problem considered in this chapter.<sup>1</sup> A couple of (theoretical) improvements to [CE82] were reported by Attie and Emerson [AE98, AE01], but the specification still does not incorporate concurrency, and no implementations are available for the moment.

A better approach to the problem consists of formally specifying not only the properties the system should satisfy, but also its architecture (how many components, and how they communicate). This approach was studied in [PR89] for *open* systems, in which the environment is an *adversary* of the system components, and the question is whether the system has a strategy that guarantees the specification against all possible behaviors of the environment. The realization problem (given the properties and the architecture, decide if

---

<sup>1</sup>The solution of [MW84] is even less satisfactory.

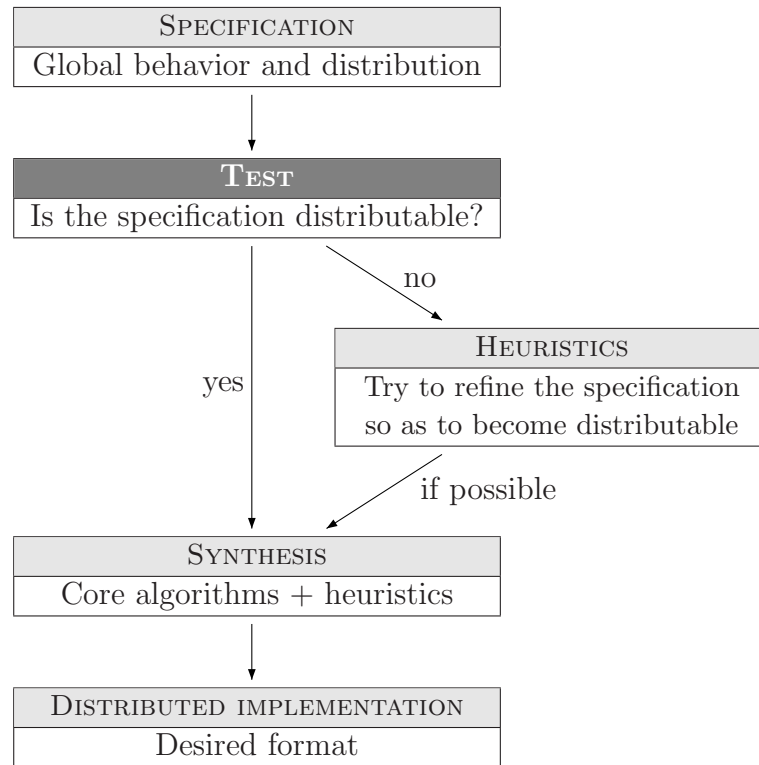


Figure 6.1: The synthesis flow followed in this thesis

there exists an implementation) was shown to be undecidable for arbitrary architectures, and decidable but non-elementary for hierarchical architectures vs. LTL specifications. Recent work [KV01] extends the decidability result (and the upper bound) to CTL\* specifications and linear architectures. To the best of our knowledge the synthesis procedures have not been implemented or tested on small examples. The introduction of [PR89] already pointed out the limitation of [CE82, MW84] mentioned above, namely the fact that the solution may be impossible to distribute with the intended concurrency, and stated that this was “particularly embarrassing when the problem is meaningful only in a distributed context, such as the mutual exclusion problem”. Unfortunately, the results of [PR89], being for open systems and very generally applicable, did not provide any better automatically generated solutions to the mutual exclusion problem. Recent papers on automatic synthesis of distributed controllers might also be applicable to this problem [MT02], but we are not aware of any paper claiming to have done so.

In this thesis we study the synthesis problem for the simpler case of *closed* systems, the original class of systems considered in [CE82, MW84] using synchronous products and asynchronous automata as implementation models. In our approach, a specification consists of two parts: (1) a distribution  $(\Sigma, Proc, \Delta)$  and (2) a regular specification  $Spec$  over the alphabet  $\Sigma$  of actions, containing all the finite executions that the synthesized system should be able to execute. The synthesis problem consists of constructing a distributed transition system over  $\Delta$  accepting  $Spec$ . In the introduction of this thesis (Chapter 1), we have seen a small example of synthesis (Section 1.2). In this chapter we show how to

---

automatize the procedure. Our approach is sketched in Figure 6.1 and can be summarized as follows:

1. We start with a global specification given as a transition system (respectively, a regular expression) and a distribution.
2. We choose the implementation model (i.e., either (deterministic) synchronous products or asynchronous automata) and the equivalence used to govern the ‘correctness’ of the distributed implementation (i.e., graph isomorphism, language equivalence, or bisimulation) – cf. Section 3.5.
3. We test whether the global specification is ‘distributable’ modulo the chosen equivalence (the computational complexity of this test was studied in Chapter 4).
4. If the implementability test is negative, we may relax the problem and heuristically look for a ‘smaller’ specification that can be distributed (recall Section 4.5).
5. If the implementability test is positive, we apply the synthesis procedures described in Chapter 5 to obtain a distributed transition system.
6. The synthesized distributed transition system may be further implemented at a lower level (e.g. as a distributed algorithm).

We implemented the synthesis procedure as described above except the last mentioned step and in this chapter we present a couple of algorithms and heuristics.<sup>1</sup> Since usually in practice *deterministic* systems are required, our implementations focused on the synthesis of **deterministic** synchronous products and asynchronous automata. Moreover, we will focus on synthesis **modulo language equivalence** (more specifications are distributable modulo language equivalence than modulo isomorphism). Nevertheless, tests of implementability modulo isomorphism can help the synthesis modulo language equivalence and were also considered.

We automatically synthesize several (maybe not ‘elegant’ but) new and far more realistic solutions to the *Mutex* problem than those of [CE82, MW84]<sup>2</sup>. We also apply our approach to the dining philosophers problem, where we are able to automatically synthesize variants of the left-handed/right-handed strategy for guaranteeing deadlock-freedom and absence of starvation.

The chapter is structured as follows: We introduce the benchmarks (mutual exclusion and dining philosophers problems) in Section 6.1, starting with the solutions in the literature, followed by our solutions with details for the simple case of two processes. Section 6.2 presents the implementation for the synthesis of deterministic synchronous products (modulo language equivalence). Then, Section 6.3 gives implementations of (heuristic) algorithms for the synthesis of asynchronous automata. Each implementation description is followed by experimental results (for the benchmarks of Section 6.1). We close the chapter with a discussion.

---

<sup>1</sup>Some of the results of this chapter were published in [SEM03, HS04].

<sup>2</sup>What is a ‘real’ solution to the mutual exclusion problem is a matter of discussion, and so we do not dare to claim that our solutions are ‘real’.

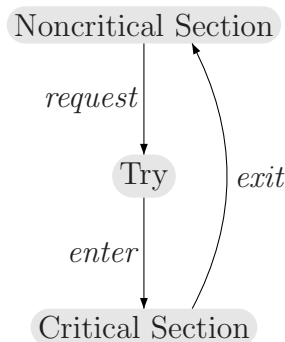


Figure 6.2: The life-cycle of a local process involved in mutual exclusion

The prototype implementations (together with the benchmarks) described in this chapter are available online at:

<http://www.fmi.uni-stuttgart.de/szs/people/stefanescu/thesis/>

The experimental results reported in this chapter were performed on a Linux machine with a 3 GHz dual Intel Pentium 4 processor and 1 GB of RAM. (The reported times are given in seconds.)

## 6.1 Motivating Example: Mutual Exclusion

In this section we describe the mutual exclusion, a basic problem for distributed systems. After a short excursion in the literature, we show how we can synthesize a solution to the problem using techniques introduced in this thesis.

The mutual exclusion problem was first presented by Dijkstra in [Dij65]. Given its importance, many mutual exclusion algorithms have been published (see for instance the surveys [Ray86, AKH03]).

A *mutual exclusion* (or *Mutex* for short) situation appears when two or more processes (a.k.a. agents)<sup>1</sup> are trying to access for ‘private’ use a common resource. Each concurrent process competing for the resource exhibits a general cyclic behavior as sketched in Figure 6.2: After being in a noncritical section, a process tries to enter the critical section (i.e., tries to have exclusive access to a shared resource). After getting permission, the process enters the critical section, which is eventually exited to go back to a noncritical state.

A ‘good’ distributed solution to the *Mutex* problem is a collection of programs, one for each process, such that their concurrent execution satisfies the properties:

- *mutual exclusion*: It is never the case that two (or more) processes have simultaneous access to the resource (i.e., they are not simultaneously in their critical sections).

<sup>1</sup>The name ‘process’ in the context of mutual exclusion algorithms should not be confounded with the processes from the set *Proc* from distributions.

- *absence of starvation*: If a process requests access to the resource, the request is eventually granted (i.e., a pending process is not put on hold forever).
- *deadlock freedom*: There is always progress possible in the system.

For exposition purposes, in this section we will consider the problem for only *two* processes. The actions of the system will be:

$$\Sigma := \{req_1, enter_1, exit_1, req_2, enter_2, exit_2\}$$

with the intended meanings of ‘requesting access to’, ‘entering’, respectively ‘exiting the critical section’. The indices 1 and 2 specify the process that executes the action.

### 6.1.1 A Classical Solution for Mutual Exclusion

In a seminal paper [CE82], Clarke and Emerson proposed two formal approaches using temporal logic for specification. The first one was a verification procedure to check the transition system semantics of a system against a temporal logic specification, this idea later breeding the prolific area of ‘model checking’. The second novel approach of [CE82] was a procedure to automatically *synthesize* a transition system from a temporal formula written in the branching-time temporal logic CTL. A similar approach was adopted by Manna and Wolper in [MW84] using linear temporal logic. Both papers exemplify their method synthesizing a mutual exclusion algorithm. We give below the idea presented in [CE82] followed by a short discussion.

The specification of [CE82] is given as a conjunction of CTL formulae for the three properties of a mutual exclusion algorithm (mutual exclusion, absence of starvation, deadlock freedom) plus a number of properties for local behavior. Using a tableau technique, the global transition system on top-left of Figure 6.3 is automatically derived<sup>1</sup>. In a second step, the solution is distributed to yield two ‘synchronization skeletons’, one for each process. Figure 6.3 shows at top-right the local behavior of the first process, the second process being dually constructed (the extraction of the local processes uses the information from the tableaux construction). Boolean conditions followed by question marks denote guards on the transitions. (We see that process 1 always needs to inspect the control state of the second process.)

A possible implementation generated by the synthesized ‘synchronization skeletons’ is given at the bottom of Figure 6.3. The communication is realized by shared variables:  $s_i$  is the control-state variable taking the values {N,T,S} with the meaning ‘process  $i$  is in its noncritical, try, respectively critical section’. The variable  $turn$  (in the range {1,2}) is used to schedule the access to the shared resource. The given pseudo-code uses the bracket notation  $\langle set\ of\ commands \rangle$  to describe that the *set of commands* must be executed in one single *atomic* step.

As possible problematic issues for the solution from [CE82] (Figure 6.3) we can enumerate:

<sup>1</sup>In fact, in [CE82] a Kripke structure is derived, but the translation to a transition system is immediate. (A *Kripke structure* is a transition system with labels attached to states rather than transitions.)

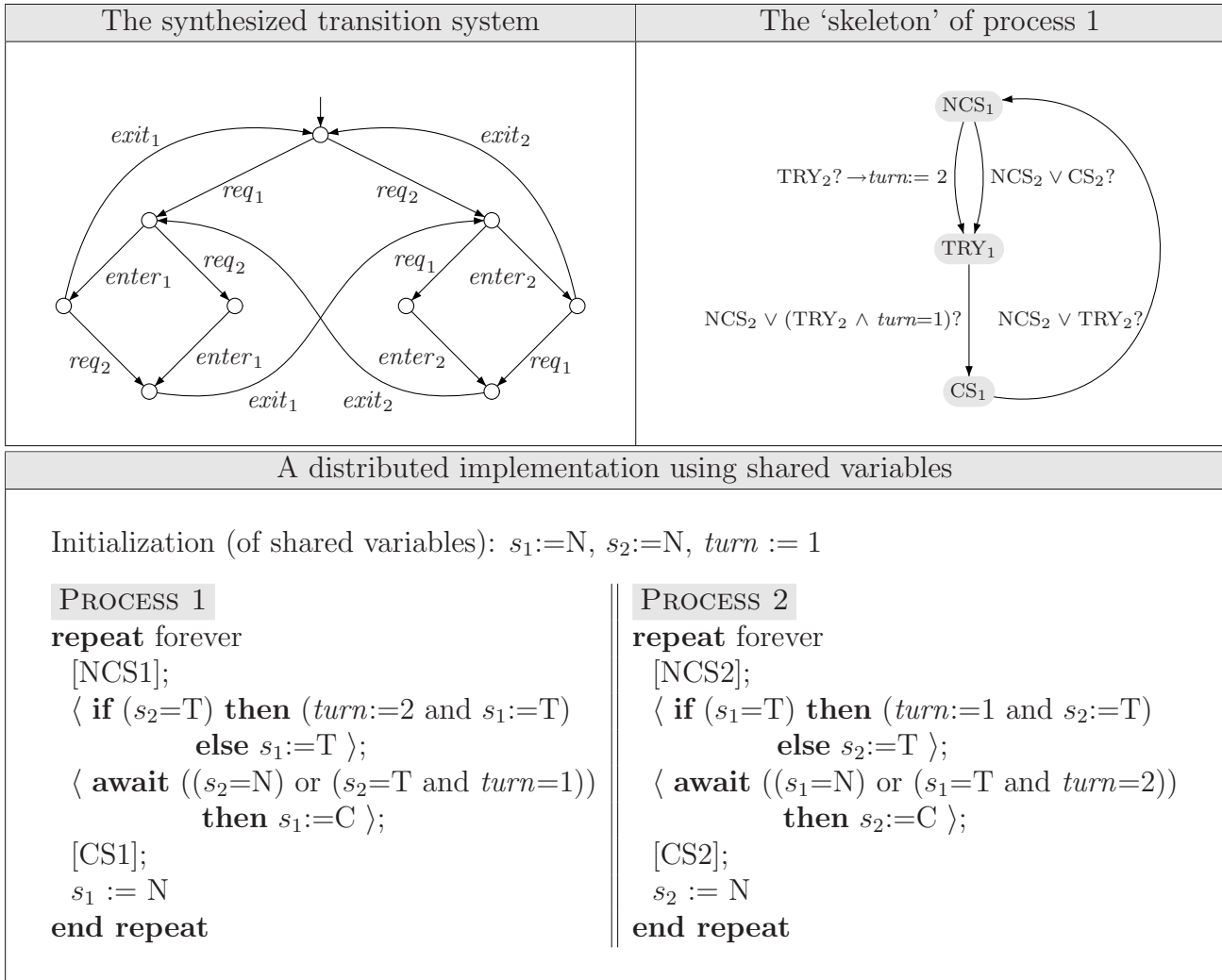


Figure 6.3: A classical synthesized solution to mutual exclusion problem [CE82]

- (a) *The test-and-set atomic operations are complicated, e.g., to move from the noncritical section to a trying state, process 1 must in one atomic instruction to: (a) test if the other process tries to enter the critical section, (b) update the value of variable  $turn$  (if process 2 is in its ‘trying’ state), and (c) move to the next state (i.e., ‘trying’ state). Thus, the grain of atomicity is rather coarse in [CE82]. Moreover, at every moment each process must be knowledgeable about the control state of the other processes.*

To tackle these issues, Attie and Emerson propose in [AE01] a systematic way to decrease a high atomicity to a lower one. The idea is to refine each large atomic instruction into smaller ones. Since this step may introduce unwanted behavior, a second step is needed to delete refinements that do not comply with the original specification. This procedure involves particular heuristics that turn the approach incomplete. Moreover, a state space explosion is susceptible for this approach.

- (b) *The synthesis procedure explicitly constructs the global state space of the distrib-*

*uted algorithm*, which can lead to the ‘classic’ state space explosion problem in a parametrized version of mutual exclusion.

This will be also a disadvantage in our approach where we start with a transition system specifying the global behavior. In fact most of the synthesis approaches in the literature face this problem (e.g., [MW84, PR89, AM94, CMT99, KV01, BCD02]). As a possible solution, [AE98, Att99] directly synthesize local pairwise-communication between processes, which are afterwards consistently composed to generate the final distributed algorithm. This proposal suffers unfortunately from the high-atomicity problem mentioned in the previous item: Each process needs to *atomically* inspect the state of all its neighbors (i.e., all processes with which it is composed in some pair-program) in a single transition.

- (c) *The two actions request actions  $req_1$  and  $req_2$  cannot be independent* following the description of Section 3.1. The reason is that the diamond properties (ID and FD) are violated for the transition system in Figure 6.3: The executions  $req_1 req_2$  and  $req_2 req_1$  lead to two *different* states. Therefore, the specification cannot be implementable over any distribution having  $req_1$  and  $req_2$  independent. On the other hand, it seems natural to ask that the two processes freely request access to enter their critical section without any coordination with the other process.

This ‘anomaly’ present in the global transition system synthesized in [CE82] can be best explained by the fact that the (CTL) temporal logic specification *does not take concurrency into account* in the first place. So, the classical temporal logics are not really adequate to specify distributed systems. Recently, *local* temporal logics (interpreted over Mazurkiewicz traces) were developed [TH98, Wal98, AS02, GM02, Wal02] but, although their complexity is at most PSPACE, the current proposals seem not very suited to be used in practice and a good candidate able to easily express natural properties of distributed systems and allowing a neat synthesis procedure is still to be discovered.

In the rest of the section, we explore the possibility of synthesizing a mutual exclusion algorithm from a global transition system using synchronous products, respectively asynchronous automata.

### 6.1.2 Mutual Exclusion Modeled in Our Framework

We work out now a solution for the mutual exclusion problem within the setting of this thesis. More precisely, we start with a distribution and a global regular specification (i.e., a transition system), and try to synthesize a synchronous product or an asynchronous automaton that is *language-equivalent*<sup>1</sup> to the specification.

The techniques used in this section may seem a bit ad-hoc, but we will see how to perform them automatically in the following sections (this section giving then a quick glimpse of the ideas used by the implementations).

<sup>1</sup>We use language equivalence rather than graph isomorphism as consistency relation between specification and distributed implementation, in order to have more specifications that are implementable (note though that the implementability test is theoretically harder for language equivalence, cf. Chapter 4).



**Distribution** According to Section 3.5, we will define first a *distribution*  $(\Sigma, Proc, \Delta)$ . Since we already have the alphabet,

$$\Sigma := \{req_1, enter_1, exit_1, req_2, enter_2, exit_2\},$$

we have to choose the set of processes  $Proc$ , and the local alphabets for each process  $p \in Proc$ . One way to come up with a distribution is to directly construct it or to decide on a concurrency relation  $(\Sigma, \parallel)$  and then to construct a compliant distribution as described in Section 3.2.1. As mentioned at the end of the previous subsection, it is desirable to have  $req_1 \parallel req_2$  (both processes *independently* choose when to request access to the critical section). On the other hand, it is clear that  $req_1$  and  $enter_1$  should be dependent (because  $req_1$  always precedes  $enter_1$ ), so a natural choice is to have the following independences:

$$\{req_1 \parallel req_2, exit_1 \parallel exit_2, req_1 \parallel exit_2, req_2 \parallel exit_1\}.$$

The dependence graph generated by the above independence relation is depicted in the top-left corner of Figure 6.4. We distribute then the actions over a number of processes such that independent actions are executed by disjoint sets of processes. The distribution is constructed from a covering by cliques of the dependence graph. A covering by maximal cliques is given by the following two sets

$$\{req_1, enter_1, exit_1, enter_2\} \text{ and } \{req_2, enter_2, exit_2, enter_1\}.$$

So, we choose two processes, say  $V_1$  and  $V_2$ , with the above as local alphabets of actions. To ease the translation of the synthesized distributed transition system into a distributed algorithm, we choose also the following cliques for the local control of the two processes/agents involved in the mutual exclusion problem:

$$\{req_1, enter_1, exit_1\} \text{ and } \{req_2, enter_2, exit_2\}.$$

Thus, we have two more processes  $P_1$  and  $P_2$  having the above sets as local alphabets. Hence, the set of processes is

$$Proc := \{P_1, V_1, V_2, P_2\},$$

with the corresponding local alphabets given in Figure 6.4.

Note that from the information on the local alphabets we can extract also the domains of the actions of  $\Sigma$ :

$$\begin{aligned} dom(req_1) &= \{P_1, V_1\}, & dom(enter_1) &= \{P_1, V_1, V_2\}, & dom(exit_1) &= \{P_1, V_1\}, \\ dom(req_2) &= \{V_2, P_2\}, & dom(enter_2) &= \{V_1, V_2, P_2\}, & dom(exit_2) &= \{V_2, P_2\}. \end{aligned}$$

**Global specification** Once we have a distribution, we give the second part of the specification by defining the desired behavior of the mutual exclusion algorithm. For that, we define a regular language,  $Mutex_1 \subseteq \Sigma^*$  capturing the properties mentioned at the beginning of Section 6.1. More precisely, we want  $Mutex_1$  to be a prefix-closed regular language satisfying the following conditions:



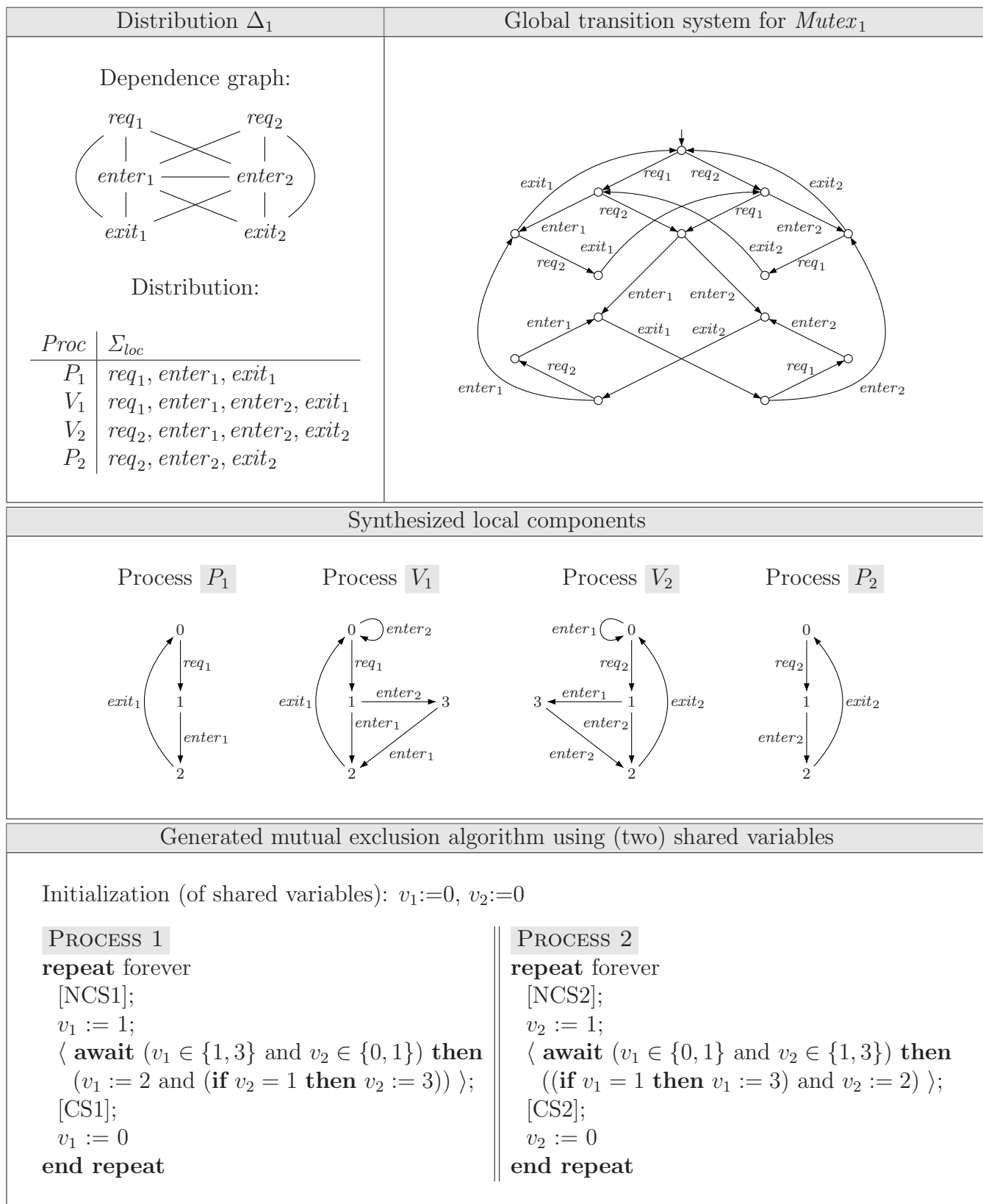


Figure 6.4: Synthesis cascade for the mutual exclusion problem  $Mutex_1$

1.  $Mutex_1$  is included in the shuffle of the prefix-closures of  $(req_1 enter_1 exit_1)^*$  and  $(req_2 enter_2 exit_2)^*$ :

$$Mutex_1 \subseteq \text{Shuffle}(\text{Prefix}((req_1 enter_1 exit_1)^*), \text{Prefix}((req_2 enter_2 exit_2)^*)).$$

I.e., each of the two processes executes  $req_i enter_i exit_i$  in cyclic order (see Figure 6.2).

2.  $Mutex_1 \subseteq \Sigma^* \setminus (\Sigma^* enter_1 (\Sigma \setminus exit_1)^* enter_2 \Sigma^*)$  and its dual version.

I.e., a process must exit before the other one can enter (we model this by *forbidding* runs that have two different enters without a corresponding exit in-between). This guarantees *mutual exclusion*.

3.  $Mutex_1 \subseteq \Sigma^* \setminus (\Sigma^* req_1 (\Sigma \setminus enter_1)^* enter_2 (\Sigma \setminus enter_1)^* enter_2 \Sigma^*)$  and its dual version.

I.e., after a request by one process, the other process can enter the critical section *at most once* (we model this by forbidding runs that after a request of one process, the other one enters twice without any entering of the first process). This is a sufficient condition to guarantee *absence of starvation*.

4. For any  $w \in Mutex_1$ , there exists an action  $a \in \Sigma$  such that  $wa \in Mutex_1$ .

I.e., each run has a possible continuation. This guarantees *deadlock freedom*.

Condition 3 needs to be discussed. In our current framework we cannot deal with ‘proper’ liveness properties, like: ‘if a process requests access to the critical section, then the access will *eventually* be granted’. This is certainly a shortcoming of our current framework. In this example, we enforce absence of starvation by putting a *concrete bound* on the number of times a process can enter the critical section after a request by the other process (here we take the bound equal to one).

Since we want our specification to be as rich as possible, we try to construct the ‘largest’ possible language  $Mutex_1$  satisfying conditions 1–4. Since last condition is not constructive, we take first into account only the first three conditions, i.e., *we intersect the languages given in 1–3*. First, this intersection is a *regular* language, using the closure properties of regular languages (cf. Proposition 2.5). Second, the intersection is a *prefix-closed* language, using the closure properties of prefix-closed languages (cf. Proposition 2.1) and the fact that the languages from 1–3 are prefix-closed (the language from constraint 1 is a shuffle of prefix-closed languages, so a prefix-closed language, while the languages of constraints 2 and 3 are also prefix-closed because any language of the form  $\Sigma^* \setminus (E\Sigma^*)$  is prefix-closed, for any regular expression  $E$  over  $\Sigma$ ).

Since the intersection of languages of 1–3 is a prefix-closed regular language, we can construct according to Corollary 2.19, a *minimal deterministic* transition system  $TS$  accepting this intersection. The result is depicted in Figure 6.4 (top-right). Coming back to the last constraint regarding the deadlock freedom, we notice that  $TS$  is already deadlock-free (cf. Definition 4.51), so the accepted language is also deadlock-free. (If the transition system had been not deadlock-free, we would have had to iteratively remove all deadlock states from  $TS$ .)

In conclusion, the transition system in Figure 6.4 (top-right) accepts the ‘largest’ language satisfying conditions 1–4, and thus specifies a mutual exclusion scenario. (Note that our specification is ‘richer’ than that from Figure 6.3.)

At this point we have a distribution and a transition system, and we want to check whether there exists a (deterministic) distributed transition system accepting the same language.

**Synthesis of synchronous products** First, we try to synthesize a *deterministic* synchronous product of transition systems. The procedure is described in Section 5.1.2, and is based on the projections on local alphabets from Algorithm 4.4 (that decides whether the language of a transition system is a product language). The algorithm is simple: We project the transition system on each of the local alphabets (of the processes) and test whether the language of the transition system includes the language of the synchronization of the projections.

For our specification in Figure 6.4, we projected the transition system  $TS$  at the top-right on the local alphabets of the four processes  $\{P_1, V_1, V_2, P_2\}$  and the result is displayed in the middle of Figure 6.4. It is easy to verify that the language of the synchronization of the local projections (having  $(0, 0, 0, 0)$  as global initial state) is indeed included in the language of  $TS$ . Hence, the specification is distributable and the projections are exactly the local components of the synthesized synchronous product.

The constructed synchronous product can be further translated into a distributed algorithm as shown in Figure 6.4 (bottom)<sup>1</sup>. The synchronizations on common actions are ‘implemented’ using two shared variables corresponding to processes  $V_1$  and  $V_2$ . According to the local transition systems associated with the variables, they both have their range given by the local state space, that is:  $\{0, 1, 2, 3\}$ . The local transition systems associated with the processes  $P_1$  and  $P_2$  are isomorphic to the generic cycle from Figure 6.2. The actions  $req$ ,  $enter$ , respectively  $exit$  will change the values of the variables accordingly: E.g.,  $req_1$  changes the value of  $v_1$  from 0 to 1, while  $enter_2$  *simultaneously* changes the values of  $v_1$  and  $v_2$  for certain pairs  $((v_1, v_2) \in \{0, 1\} \times \{1, 3\})$ .

By construction, the synthesized algorithm will have the characteristics of the mutual exclusion paradigm described at the beginning of Section 6.1. Comparing it with the solution in the literature presented in Section 6.1.1, we can make the following observations with respect to the problems mentioned at the end of that section (items (a)–(c)). First, our solution still suffers from complicated atomic test-and-set commands (cf. the implementation of the  $enter$  actions). Moreover, the global transition system of the specification is explicitly constructed. On the other hand, in our case the requests are independent of each other (this requirement is present in the distribution we choose). While there is nothing we can do about the explicit construction in the specification, we try to reduce the atomicity of the commands in the following subsection by refining the distribution.<sup>2</sup>

**Synthesis of asynchronous automata** Since a synchronous product is a particular case of asynchronous automaton over a fixed distribution (Remark 3.19), once we synthesize a synchronous product, we have also an isomorphic asynchronous automaton, so we obtain the same distributed algorithm. However, to practice a bit the synthesis of

<sup>1</sup>The pseudo-code was derived by hand, but the procedure can be automatized.

<sup>2</sup>Alternatively, one could try to reduce the high atomicity of our synthesized solution using the method proposed in [AE01] or another refinement techniques from the literature (see references in [AE01]).

asynchronous automata, we assume that we have not tested the implementability as a synchronous product and we try to directly synthesize an asynchronous automaton for our problem. (The synthesis modulo language equivalence for *deterministic* asynchronous automata is covered by Section 5.2.2.)

According to Theorem 3.62, the class of languages accepted by deterministic asynchronous automata coincides with the class of prefix-, forward-, trace-closed regular languages. Using Proposition 3.36, the implementability test for the language of a *deterministic* transition system  $TS$  can be done in polynomial time as follows:

- minimize the deterministic transition system  $TS$  (if necessary) and
- test the minimized  $TS$  if it satisfies the ID and FD rules.

If the ID and FD rules hold, we know that there exists an asynchronous automaton  $\mathcal{AA}$  language equivalent to  $TS$ . In general,  $\mathcal{AA}$  will be constructed using Zielonka's procedure (see Section 5.2.2).

In our case, the global transition system  $TS$  from Figure 6.4 is minimal and satisfies ID and FD. This allows us to apply Zielonka's construction, that yields a (reachable deterministic) asynchronous automaton with 34 *global* states. Yet, using the implementability test *modulo isomorphism* from Section 5.2.1 (based on Theorem 3.32), we see that  $TS$ , having only 14 *global* states, is already isomorphic to an asynchronous automaton! The families of local states and transitions can be constructed as in Section 5.2.1 and we obtain 3 local states for each of the processes  $P_1$  and  $P_2$ , and 7 local states for each of  $V_1$  and  $V_2$ .

We can now ask if the solution can be simplified, i.e., if there is a smaller family of local states making the transition system of Figure 6.4 asynchronous. This amounts to finding a *larger* family  $(\equiv_k)_{k \in Proc}$  of equivalences satisfying the properties of Theorem 3.32. This can be done by heuristically trying to merge equivalence classes, and checking if the resulting equivalences still satisfy conditions  $DAA_3$  and  $DAA_4$  from Theorem 3.32. It turns out that in the end we are able to obtain an optimal solution in which  $V_1$  and  $V_2$  have only 4 local states each and this solution is the same as the one obtained for synchronous products presented in Figure 6.4 (this is no surprise, given the fact that the class of asynchronous automata subsumes one of the synchronous products).

We also remark that the above specification is not implementable in the setting of *unlabeled* Petri nets [BCD02]. (Asynchronous automata are more expressive than unlabeled Petri nets: The latter can be seen as subclasses of the former with the additional condition that, for each action  $a$ , the relation  $\rightarrow_a$  contains at most one element.)

### 6.1.3 Mutual Exclusion Revisited

One of the disadvantages of the solution synthesized in the previous subsection is that of high atomicity: The *enter* actions should simultaneously update the variables  $v_1$  and  $v_2$  and this is difficult to implement. We observe that the problem lies in the distribution we have chosen. More precisely, there is no way we can cover the dependence graph of Figure 6.4 by cliques such that the domains of the *enter* actions contain at most one variable (there are too many dependence relations to be covered). As a possible solution, we remove the dependence  $enter_1 \parallel enter_2$  (equivalently, we add  $enter_1 \parallel enter_2$  to the

independence relation). The new dependence graph is depicted at top-left of Figure 6.5 and a covering by maximal cliques gives the distribution depicted below the dependence graph. Moreover, the new domains of the actions extracted from the new distribution are:

$$\begin{aligned} \text{dom}(req_1) &= \{P_1, V_1\}, & \text{dom}(enter_1) &= \{P_1, V_2\}, & \text{dom}(exit_1) &= \{P_1, V_1\}, \\ \text{dom}(req_2) &= \{V_2, P_2\}, & \text{dom}(enter_2) &= \{V_1, P_2\}, & \text{dom}(exit_2) &= \{V_2, P_2\}. \end{aligned}$$

Looking at the above domains we see that the *enter* actions will involve now only one of the variables, and not both.

Unfortunately, the specification  $Mutex_1$  is *not implementable* anymore under the new distribution. The reason is that all languages accepted by *deterministic* distributed transition systems are necessarily *forward-closed* (Corollary 3.35), while the language of  $TS$  from Figure 6.4 is not forward-closed:  $req_1 req_2 enter_1$ ,  $req_1 req_2 enter_2$  belong to  $L(TS)$  and  $enter_1 || enter_2$ , but  $req_1 req_2 enter_1 enter_2$  does *not* belong to  $L(TS)$ . In fact, the problem is that  $TS$  does not satisfy FD: In the state reached by executing the sequence  $req_1 req_2$ , we can execute both (independent actions)  $enter_1$  and  $enter_2$ , but there is no converging state to ‘close’ a diamond as required by the forward diamond rule FD (Figure 3.5).

Since the specification in this form is not implementable, we try to change the transition system in order to obtain an implementable one. For that, we observe that the specification has been obtained by complementing undesirable behaviors. In other words, we are requesting the system to execute ‘all behaviors which are not bad’. We can try to synthesize another, more modest solution, which executes less behaviors (cf. Section 4.5). A simple way to do so is to ‘solve’ the FD violation by removing either of the  $enter_1$  or  $enter_2$  labeled transition involved in the problematic FD (cf. Section 4.5.2). Without loss of generality, suppose we delete the  $enter_2$  transition. Doing so, we are intuitively requiring that, if the two processes make a request independently of each other, then process 1 has *priority* over process 2. The new transition system  $TS'$  is depicted at top of Figure 6.5. If we denote the language accepted by  $TS'$  by  $Mutex_2$ , then  $Mutex_2 \subset Mutex_1$ , so  $Mutex_2$  is still compliant with the formal requirements from the previous subsection (the new transition system is also deadlock-free). Moreover, the new transition system satisfies ID and FD, so we can try to apply the synthesis procedure.

**Synthesis of synchronous products** As mentioned in the previous subsection, we have to check that  $Mutex_2$  is a *product language* (cf. Definition 3.40) using projections on the local alphabets (Proposition 3.43). Unfortunately, this is not the case, i.e., the synchronization of languages of the projections of  $Mutex_2$  on local alphabets strictly includes  $Mutex_2$ . Take for instance the following execution<sup>1</sup>:

$$w := req_2 enter_2 req_1 enter_1.$$

On one hand, it is easy to see that  $w \notin Mutex_2$  (simulate  $w$  in  $TS'$  or notice that  $w$  violates the mutual exclusion property). On the other hand, we show that  $w$  belongs to the synchronization of the projections of  $Mutex_2$ , showing that  $w \upharpoonright_{\Sigma_{loc}(p)} \in Mutex_2 \upharpoonright_{\Sigma_{loc}(p)}$  for each  $p \in Proc$ :

<sup>1</sup>In Section 6.2, we give implementation details on how to automatically find such counterexamples.

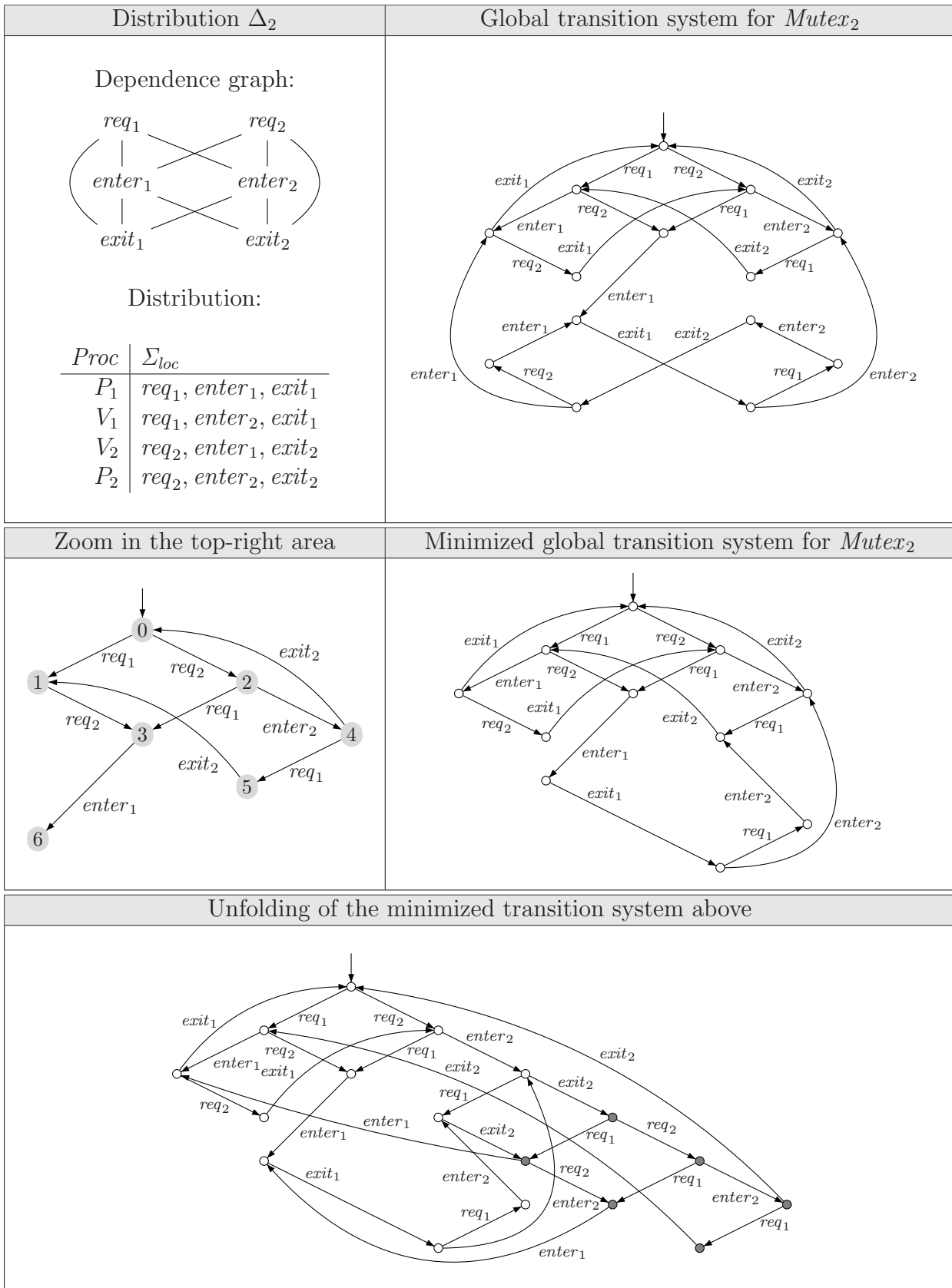


Figure 6.5: Synthesis cascade for the revisited mutual exclusion problem  $Mutex_2$

Initialization (of shared variables):  $v_1:=0, v_2:=0$

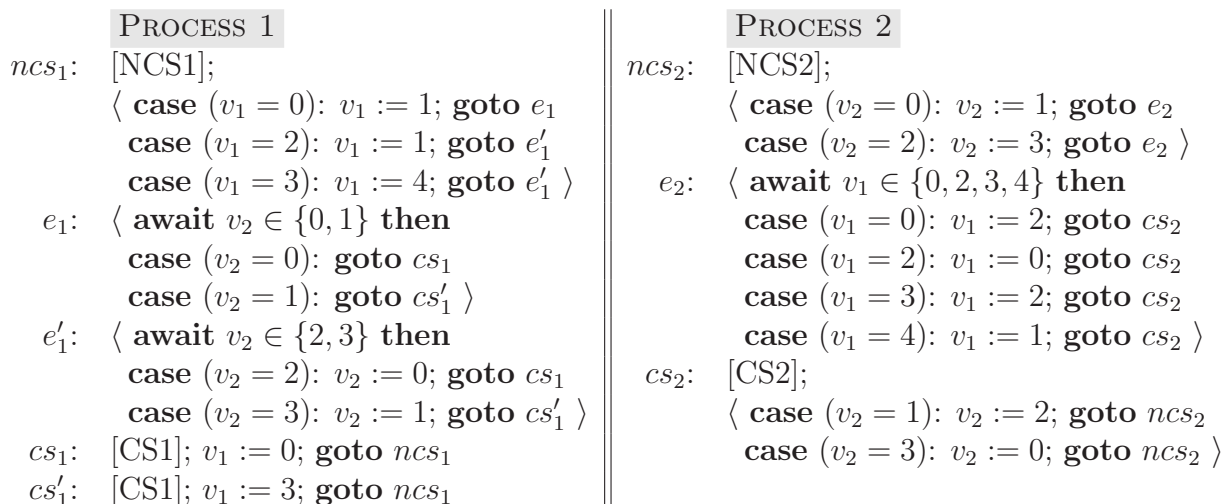


Figure 6.6: Synthesized mutual exclusion algorithm for  $Mutex_2$  using (two) shared variables

- For  $p := P_1$ , we have  $\Sigma_{loc}(P_1) = \{req_1, enter_1, exit_1\}$ , so  $w \upharpoonright_{\Sigma_{loc}(P_1)} = req_1 enter_1$ . Choosing  $u := req_1 enter_1 \in Mutex_2$ , we have that indeed  $w \upharpoonright_{\Sigma_{loc}(P_1)} = u \upharpoonright_{\Sigma_{loc}(P_1)} \in Mutex_2 \upharpoonright_{\Sigma_{loc}(P_1)}$ .
- For  $p := V_1$ , we have  $\Sigma_{loc}(V_1) = \{req_1, enter_2, exit_1\}$ , so  $w \upharpoonright_{\Sigma_{loc}(V_1)} = enter_2 req_1$ . Choosing  $u := req_2 enter_2 req_1 \in Mutex_2$ , we have that indeed  $w \upharpoonright_{\Sigma_{loc}(V_1)} = u \upharpoonright_{\Sigma_{loc}(V_1)} \in Mutex_2 \upharpoonright_{\Sigma_{loc}(V_1)}$ .
- For  $p := V_2$ , we have  $\Sigma_{loc}(V_2) = \{req_2, enter_1, exit_2\}$ , so  $w \upharpoonright_{\Sigma_{loc}(V_2)} = req_2 enter_1$ . Choosing  $u := req_2 req_1 enter_1 \in Mutex_2$ , we have that indeed  $w \upharpoonright_{\Sigma_{loc}(V_2)} = u \upharpoonright_{\Sigma_{loc}(V_2)} \in Mutex_2 \upharpoonright_{\Sigma_{loc}(V_2)}$ .
- For  $p := P_2$ , we have  $\Sigma_{loc}(P_2) = \{req_2, enter_2, exit_2\}$ , so  $w \upharpoonright_{\Sigma_{loc}(P_2)} = req_2 enter_2$ . Choosing  $u := req_2 enter_2 \in Mutex_2$ , we have that indeed  $w \upharpoonright_{\Sigma_{loc}(P_2)} = u \upharpoonright_{\Sigma_{loc}(P_2)} \in Mutex_2 \upharpoonright_{\Sigma_{loc}(P_2)}$ .

Therefore, according to Proposition 3.43,  $Mutex_2$  is not a product language, so there exists *no* deterministic synchronous product accepting  $Mutex_2$  (Corollary 3.45).

Of course one could try further to remove further behaviors and test if we can obtain eventually a non-trivial product language, but there is no systematic way to successfully perform this operation (the relaxed implementability problem for synchronous products is proved undecidable in Corollary 4.45).

Moreover, the given specification is not implementable as an unlabeled Petri net either (following [BCD02]).

**Synthesis of asynchronous automata** Even we cannot synthesize a solution for  $Mutex_2$  as a synchronous product, we can construct a solution as an asynchronous au-



tomaton, because  $Mutex_2$  is a prefix-, forward-, trace-closed regular language ( $TS'$  at top of Figure 6.5 satisfies ID and FD). Thus, following Section 5.2.2, we apply Zielonka's construction and generate a (reachable deterministic) asynchronous automaton with 4799 global states (the implementation used will be described in Section 6.3.3). Since this is unacceptable for a mutual exclusion algorithm with only two parties, the question is whether we can synthesize something smaller.

Similar to the previous case, we can check if the given transition system is already isomorphic to an asynchronous automaton. Unfortunately, this is not the case. We use for that the test given by Theorem 3.32 (see also Section 4.2.2). We explain in detail why the transition system is not distributable.

We emphasized in Figure 6.5 (middle-left) the 'problematic' area of the transition system that leads to the failure of implementability modulo isomorphism via Theorem 3.32. Let  $(\equiv_p)_{p \in Proc}$  be the least family of equivalences satisfying  $DAA_1$  and  $DAA_2$  (from Theorem 3.32). The transitions emphasized in Figure 6.5 contribute to the violation of  $DAA_4$  for the states 3 and 5, and transition  $3 - enter_1 \rightarrow 6$  as explained below:

*technically* : Using  $DAA_1$  (and transitivity) for the transitions  $3 \leftarrow req_2 - 1$  and  $1 \leftarrow exit_2 - 5$ , we have that  $3 \equiv_{P_1, V_1} 5$ . Similarly, Using  $DAA_1$  (and transitivity) for the transitions  $3 \leftarrow req_1 - 2$ ,  $2 - enter_2 \rightarrow 4$ , and  $4 - req_1 \rightarrow 5$  we have that  $3 \equiv_{V_2} 5$ . So,  $3 \equiv_{P_1, V_1, V_2} 5$ . Since  $dom(enter_1) = \{P_1, V_2\}$  is included in  $\{P_1, V_1, V_2\}$ , the states 3 and 5 are equivalent on the domain of  $enter_1$ . The violation of  $DAA_4$  follows now from the fact that although  $3 \equiv_{dom(enter_1)} 5$ , state 3 can execute an  $enter_1$  transition (reaching state 6), while 5 cannot.

*intuitively* : In the state 3, both processes already requested access to the critical section and it is possible for the first process  $P_1$  to proceed with  $enter_1$ . In the state 5,  $P_2$  requested and entered the critical section and after that  $P_1$  made the request, but it cannot go further with  $enter_1$  because of mutual exclusion. Roughly, this global scenario cannot be distributed because in case both processes make a request, the first process  $P_1$  is not able to 'notice' if  $P_2$  enters the critical section (this possibly leading to a violation of mutual exclusion), the main reason being that  $enter_1$  and  $enter_2$  are independent (i.e., they are not 'aware' of each other). The situation is even more subtle because the transition  $exit_2$  from 5 to 1 contributes to the troubles, obstructing  $req_1$  to let  $P_1$  in state 5 know whether  $enter_2$  had happened or not.

The problem persists after minimizing  $TS'$  (we minimize  $TS'$  to have a 'canonical' representation of the language accepted by  $TS'$ ). The reasons are the same as above, because the 'problematic' zone showed in Figure 6.5 is not modified during minimization, as one can see looking at the minimization of  $TS'$  that is also depicted in Figure 6.5 (middle-right).

A possible solution to this situation is to modify the transition system such that  $DAA_4$  is not violated anymore, *while preserving the language*. This can be done for instance by **unfolding** the  $exit_2$ -labeled transitions from 5 to 1 and 4 to 0<sup>1</sup>.

<sup>1</sup>The idea is to unfold  $5 - exit_2 \rightarrow 1$  involved in the failure of  $DAA_4$ , but since  $exit_2 \parallel req_1$ , we unfold also  $4 - exit_2 \rightarrow 0$  in order to preserve the diamond formed by states  $\{5, 4, 0, 1\}$ .



Unfolding a transition  $q_1 \xrightarrow{a} q_2$  is done by creating a new state  $q_{new}$ , replacing transition  $q_1 \xrightarrow{a} q_2$  by a new one  $q_1 \xrightarrow{a} q_{new}$ , and adding new transitions starting in  $q_{new}$  such that the behavior of the new transition system is the same. The preservation of the language of the transition system after the unfolding can be obtained by choosing the new transitions from  $q_{new}$  to have the same labels as the transitions from  $q_2$  and such that the destination states have respectively the same behavior<sup>1</sup> as the corresponding destination states reached from  $q_2$ . There are several ways of performing the unfolding depending on the choice for the destinations of the new edges going out of  $q_{new}$ .

After unfolding the two *exit*<sub>2</sub> transitions, we test again if the new transition system is isomorphic to an asynchronous automaton. As this is not the case, we take a look again where ‘problematic’ areas could be and try to solve the violations again by unfolding. Such procedure is worth only in case it terminates. In Section 6.3.3 we show how to use Zielonka’s equivalence to force termination. The hope is that this approach will construct smaller asynchronous automata compared with Zielonka’s approach.

Fortunately, applying the above heuristic to our problem, we obtain a transition system isomorphic to an asynchronous automaton with 17 states (compare this with the 4799 global states for Zielonka’s construction!). The transition system is depicted at the bottom of Figure 6.5, where the new states are distinguished by a gray tint.

Once distributed over the four processes of the specification (and merging local states if possible), we obtain the distributed algorithm shown in Figure 6.6. The variables  $v_1$  and  $v_2$  range over  $[0, 1, 2, 3, 4]$ , respectively  $[0, 1, 2, 3]$ . The labels associated with the commands are rather suggestive. The command in brackets  $\langle \dots \rangle$  is executed atomically and the program pointers for the two components advance only as a result of a **goto** command.

Entering the critical section is guarded by the values of the variables. However, the *request* and the *exit* actions are not hindered in any way. This fact may not be so obvious, so we elaborate a bit. The request action *req*<sub>1</sub> (first component after executing the noncritical section NCS<sub>1</sub>) seems to be incomplete:  $v_1$  has only can only choose from 0, 2, and 3. It can be proved that indeed  $v_1$  can have only these three values when first process reaches that point. To sketch a proof, we know that  $v_1$  can only be updated by *exit*<sub>1</sub>, *enter*<sub>2</sub>, or *req*<sub>1</sub> itself. The action *exit*<sub>1</sub> (i.e., the assignment after the critical section CS<sub>1</sub>) will change  $v_1$  to 0 or 3, which is fine. On the other hand, *enter*<sub>2</sub> (i.e., the atomic command labeled by  $e_2$  in the second component) may change  $v_1$  to 0, 1 or 2. The only debatable case is when *enter*<sub>2</sub> changes  $v_1$  from 4 to 1. But  $v_1$  is changed to 4 only by *req*<sub>1</sub> itself, so *enter*<sub>2</sub> will not block *req*<sub>1</sub> because the next action to update  $v_1$  is *exit*<sub>1</sub> (if  $v_1$  is 1, *enter*<sub>1</sub> cannot change it), which changes it from 1 to 0 or 2 which is safe for *req*<sub>1</sub>. A similar analysis can be carried out for *req*<sub>2</sub> and *exit*<sub>2</sub>. We mention that the above properties can be informally verified using the global transition system at bottom of Figure 6.5 noting that, for example, a *req*<sub>1</sub> action is really enabled everywhere after the occurrence of *exit*<sub>1</sub>.

To conclude, both components can independently request to enter, respectively exit their critical sections. Moreover, each command will change only one variable. However,

<sup>1</sup>Two states have ‘the same behavior’ if the languages of the possible runs starting in the two states are equal. Pairs of states with the same behavior can be identified with a procedure similar to the minimization of a deterministic transition system (Corollary 2.19). For efficiency, after each unfolding step, we can tag the new state  $q_{new}$  as having the same behavior as  $q_2$ .

the two generated components are now asymmetric, due to the fact that removing the  $enter_2$  transition in the initial transition system broke the symmetry of  $Mutex_1$  giving priority to first process in case there are requests from both parties. Yet, according to the specification, the algorithm is starvation-free, i.e., if the second process request access to the critical section, it will receive it as soon as possible.

### 6.1.4 Parametrized Mutual Exclusion

The mutual exclusion problem description can be parametrized to allow more than only two processes and will be used as benchmark for our implementations. We try to generalize the two specifications for mutual exclusion from Sections 6.1.2 and 6.1.3.

First, it is natural for  $n \geq 2$  processes to have the alphabet:

$$\Sigma := \bigcup_{i \in [1..n]} \{req_i, enter_i, exit_i\}.$$

There are several possible distributions of the above actions such that the request actions are independent (this requirement is a natural one). We choose two similar to the ones chosen in Sections 6.1.2 and 6.1.3, but now over  $2n$  processes:

$$Proc := \{P_1, \dots, P_n, V_1, \dots, V_n\},$$

with local alphabets as below:

- first distribution  $\Delta_1$ :  
 $\Sigma_{loc}(V_i) := \{req_i, exit_i, enter_1, \dots, enter_n\}$  and  
 $\Sigma_{loc}(P_i) := \{req_i, enter_i, exit_i\}$ .
- second distribution  $\Delta_2$ :  
 $\Sigma_{loc}(V_i) := \{req_i, exit_i, enter_1, \dots, enter_{i-1}, enter_{i+1}, \dots, enter_n\}$  and  
 $\Sigma_{loc}(P_i) := \{req_i, enter_i, exit_i\}$ .

(The difference between  $\Delta_1$  and  $\Delta_2$  is that  $enter_i \notin \Sigma_{loc}(V_i)$ .)

The above distribution will generate (if possible) distributed implementations that share  $n$  variables<sup>1</sup>.

We also generalize the regular specification from Section 6.1.2 to  $n$  processes as follows:

1.  $Mutex \subseteq \text{Shuffle}_{i \in [1..n]} \text{Prefix}((req_i enter_i exit_i)^*)$   
 (any global run is a shuffle of local runs)
2.  $Mutex \subseteq \Sigma^* \setminus \bigcup_{i \neq j} (\Sigma^* enter_i (\Sigma \setminus exit_i)^* enter_j \Sigma^*)$   
 (mutual exclusion)

<sup>1</sup>If one wants to have variables that are ‘more local’ (i.e., shared by a smaller number of processes), we could have chosen a distribution like:  $Proc := \{P_i \mid i \in [1..n]\} \cup \{V_{ij} \mid i \neq j\}$  and  $\Sigma_{loc}(P_i) := \{req_i, enter_i, exit_i\}$ ,  $\Sigma_{loc}(V_{ij}) = \{req_i, exit_i, enter_j\}$ , for  $i \neq j$ . However, the distributed algorithm over such distribution would not have a lower atomicity than for the previous distributions and the number of variables will increase by a factor.

Table 6.1: The (sizes of the) benchmarks used in this chapter

Problem	$\Delta$		$TS$	
	$ \Sigma $	$ Proc $	$ Q $	$ \rightarrow $
<i>Mutex</i> (2)	6	4	14	22
<i>Mutex</i> (3)	9	6	107	210
<i>Mutex</i> (4)	12	8	1340	3040
<i>Mutex</i> (5)	15	10	25537	63990
<i>Phil</i> (2)	6	4	7	10
<i>Phil</i> (3)	9	6	25	57
<i>Phil</i> (4)	12	8	79	244
<i>Phil</i> (5)	15	10	241	935

3.  $Mutex \subseteq \Sigma^* \setminus \bigcup_{i \neq j} (\Sigma^* req_i (\Sigma \setminus enter_i)^* enter_j (\Sigma \setminus enter_i)^* enter_j \Sigma^*)$   
 (strong absence of starvation)

4. For any  $w \in Mutex$ , there exists an action  $a \in \Sigma$  such that  $wa \in Mutex$ .  
 (deadlock freedom)

We choose *Mutex* as the intersection of the prefix-closed regular languages from 1–3 above.

Using AMoRE [MMP<sup>+</sup>95], a tool for manipulating finite automata and regular expressions, we are able to generate transition systems for up to  $n = 5$  processes (for  $n = 6$ , AMoRE runs out of resources). The generated transition systems were determinized (in fact, since the specifications involved many complementation operation, the output of AMoRE was already deterministic) and minimized. Moreover, the generated transition systems do not contain any *deadlock state*, i.e., a state where no action is enabled, so condition 4 above holds as well.

The sizes of the parametrized specification for mutual exclusion (including the distributions) is given in the first half of Table 6.1. (Since the format of representing finite transition system in AMoRE was different from the format used in our implementations, we had to write scripts to translate the former format into the latter.)

### 6.1.5 Dining Philosophers

Another classic example of resource-allocation paradigm for distributed systems is the *dining philosophers* problem: There are  $n$  philosophers (users) seated around a table. Between each pair of neighbor philosophers there is a single fork (resource). They are usually thinking, but from time to time they are getting hungry (request resources). Therefore they are trying to make exclusive use of the (two) forks next to them, if possible, then take them, eat and finally release them. The problem is to find an algorithm describing how the philosophers may use and release the forks such that they all have a pleasant dinner. The usual requirements for this problem are: The eating process does not stop (deadlock freedom), every fork is owned by at most one philosopher at any time (mutual exclusion), and if a philosopher gets hungry, then her/his neighbors will eat at most once

before the philosopher get the chance to eat (strong absence of starvation). Since the specification for dining philosophers resembles somehow the mutual exclusion case study, we will not provide as many details for it.

The actions for  $n$  dining philosophers are:

$$\Sigma := \bigcup_{i \in [1..n]} \{h_i, l_i, r_i, rel_i\}$$

with the intended meaning that each philosopher is getting hungry ( $h_i$ ), takes her/his left fork ( $l_i$ ), take her/his right fork ( $r_i$ ), and release both forks ( $rel_i$ ). The set of processes will consist of one process for each philosopher and one process for each fork<sup>1</sup>:

$$Proc := \{P_i \mid i \in [1..n]\} \cup \{F_i \mid i \in [1..n]\},$$

with the local alphabets:

$$\Sigma_{loc}(P_i) = \{h_i, l_i, r_i, rel_i\} \text{ and } \Sigma_{loc}(F_i) = \{r_i, l_{i+1}, rel_i, h_i, h_{i+1}, rel_{i+1}\},$$

where we assume that fork  $i$  is shared by philosophers  $i$  and  $i + 1$  (in fact  $i$  has its range in the ring  $\mathbb{Z}/n\mathbb{Z}$ , so the successor of  $n$  is 1).

The global behavior  $Phil$  is a prefix-closed regular language such that:

1.  $Phil \subseteq \text{Shuffle}_{i \in [1..n]} \text{Prefix}((h_i l_i r_i rel_i + h_i r_i l_i rel_i)^*)$   
(any global run is a shuffle of local runs)
2.  $Phil \subseteq \Sigma^* \setminus \bigcup_{i \in [1..n]} (\Sigma^* (r_i (\Sigma \setminus rel_i)^* l_{i+1} + l_{i+1} (\Sigma \setminus rel_{i+1})^* r_i) \Sigma^*)$   
(no fork can be simultaneously used by two philosophers)
3.  $Phil \subseteq \Sigma^* \setminus \bigcup_{i \in [1..n]} (\Sigma^* (h_i (\Sigma \setminus rel_i)^* rel_{i-1} (\Sigma \setminus rel_i)^* rel_{i-1} + h_i (\Sigma \setminus rel_i)^* rel_{i+1} (\Sigma \setminus rel_i)^* rel_{i+1}) \Sigma^*)$   
(strong absence of starvation)
4. For any word  $w \in Phil$ , there exists an action  $a \in \Sigma$  such that  $wa \in Phil$ .  
(deadlock freedom)

We sketch below the experience with two philosophers (i.e.,  $n = 2$ ). As for mutual exclusion case study, we construct a transition system accepting the intersection of the languages from 1–3). In this case the (minimal deterministic) transition system has 24 states, of which 3 are deadlock states. To conform with requirement 4 above, we remove all deadlock states. We verify then the ID and FD and see that FD are violated in the state where both philosophers are hungry. We solve the conflicts by cutting two transitions<sup>2</sup> (and removing states that become unreachable because of that). Zielonka's construction can be now applied and outputs a (reachable) deterministic asynchronous

<sup>1</sup>Thus, each fork will have associated an internal variable used to represent e.g., the status of the fork.

<sup>2</sup>In fact, by cutting these transitions we enforce in fact a strategy for the philosophers of choosing one hand before the other (so, we automatically rediscover a classic strategy used to solve the dining philosophers problem).

automaton with 2169 global states. However, testing implementability modulo isomorphism and unfolding where needed, we construct a much smaller asynchronous automaton with only 28 global states. Once more, the solution has the problem that  $h_1$ ,  $h_2$ ,  $rel_1$ , and  $rel_1$  are implemented as rather complicated atomic actions. Again, the specification is not distributable using synchronous products or unlabeled Petri nets.

Trying to work with a greater number of philosophers, we have a scalability problem: The proposed specification seems rather complex, as we were not able to construct a (minimal deterministic) transition system accepting the regular specification for a number of processes greater than three using the finite automata tool AMoRE [MMP<sup>+</sup>95]. Thus, we do not even have the chance to apply the synthesis procedure. In order to simplify the specification, we leave out the strong absence of starvation property (requirement 3). Therefore, we consider a simpler version of the problem without the ‘getting hungry’ action. The new specification will be obtained by removing condition the  $h_i$  actions all over the place. We still have the property of exclusive access to the forks and deadlock freedom, so the problem remain interesting, while we can generate a transition system from the specification (using AMoRE) for up to 5 philosophers.

After generating (using AMoRE) the transition systems accepting the new parametrized specifications (i.e., the intersection of the regular expressions from 1–3 without  $h_i$  actions), we can easily check also the *deadlock freedom* property (given by condition 4). We see that each of the generated transition systems will contain a deadlock state from which no action is executable. The paths leading the deadlock state gives the scenarios for the deadlock: *A deadlock occurs when each philosopher takes her/his left fork* (dually, the same happens when each philosophers takes her/his right fork). In order to construct implementations that are deadlock-free, we proceed to delete the deadlock state (together with the transitions leading to it). We check again for deadlock states, but in our case there is none found anymore. Also, we minimize the transition system after we remove the deadlock state.

The benchmarks based on the dining philosophers problem in the rest of the chapter will use this deadlock-free version without the strong absence of starvation condition. The sizes of the parametrized specifications for dining philosophers (including the distributions) are given in the second half of Table 6.1.

## 6.2 Implementation for Synchronous Products of Transition Systems

In this section we describe our implementation for the synthesis of deterministic synchronous products of transition systems (modulo language equivalence).

The procedure is described in Section 5.1.2, and is based on the projections on local alphabets from Algorithm 4.4 (that decides whether the language of a transition system is a product language [CMT99]). The algorithm is sketched below:

1. *Project* the transition system on each of the local alphabets (of the processes).
2. Test whether the language of the transition system *includes* the language of the synchronization of the projections. (The other inclusion always holds.)

3. If the inclusion holds, the synthesis problem has a solution, i.e., there exists a *deterministic* synchronous product accepting the same language as the specification. The components of the synchronous product are given by the *determinization* (and optionally minimization) of the projections.
4. If the inclusion does not hold, the synthesis problem does not have a solution. In this case the algorithm should return a *counterexample* in form of a run that is executable by the synchronization of the projections, but is not present in the original specification.<sup>1</sup>

We implement the above test using a *reduction to the (non)reachability problem for 1-safe Petri nets*<sup>2</sup>. The difficult part of the above algorithm is the language inclusion test. Following a classic pattern in model checking theory, testing the *inclusion* of the behavior of the implementation into that of a specification, i.e.,  $L(Impl) \subseteq L(Spec)$ , is reduced to checking the *emptiness* of the intersection of the implementation and the *complement* of the specification, i.e., testing  $L(Impl) \cap \overline{L(Spec)} = \emptyset$ . In a similar vein, we will test the emptiness of the intersection of the synchronization of projections and the complement of the original specification. Since the intersection of two languages relies on the product construction, this fits well with our framework based on synchronizations on common actions. The idea is to synchronize the projections and a transition system for the complement, to describe the result of the synchronization as a *1-safe Petri net*, and to test the reachability of certain markings. If any of these markings is reachable, we conclude that the intersection is not empty, which further implies that the synthesis problem does not have a solution. The details of this procedure are provided in Algorithm 6.1.

**Proposition 6.1** *The reduction given by Algorithm 6.1 is correct (in the sense specified at the bottom of the algorithm).*<sup>3</sup>

**Proof.** (Sketch) The synthesis problem does not have a solution if and only if  $L(TS)$  is *not* a product language (Proposition 3.43) if and only if there exists a run executable by the synchronization of the projections of  $TS$ , but not by  $TS$  if and only if there exists a run executable by the synchronization of the projections of  $TS$  and a transition system accepting the complement of  $L(TS)$  if and only if there exists a path executable in the net  $N$  constructed by Algorithm 6.1 starting in  $M$  and reaching a marking having a token in the place corresponding to  $\perp$ . □

To verify the reachability for the generated 1-safe Petri net, we use already available reachability tests<sup>4</sup> implemented in tools like **Prod** [VHL97], **PEP** [PEP], **Mcsmodels** [Hel99], **Spin** [Hol04], or **LoLa** [Sch00]. A very convenient way to access all the reachability checkers

<sup>1</sup>In principle, this counterexample may be used to guide the refinement of the specification until it becomes distributable, but this is a topic of future research.

<sup>2</sup>This reduction was suggested by Keijo Heljanko.

<sup>3</sup>Algorithm 6.1 obviously terminates since it involves no loops.

<sup>4</sup>Another possibility would be to use the net unfolders specialized to nets obtained from synchronous products described in [ER99] and modify it to exit in case a special transition associated with  $\perp$  is encountered. Unfortunately, an implementation for this procedure is not available at the moment (however, it is currently proposed as a small student project).



Algorithm 6.1: The reduction of the synthesis problem for deterministic synchronous products to the non-reachability of 1-safe Petri nets

<b>Input</b>	a distribution $(\Sigma, Proc, \Delta)$ and a transition system $TS$
	<ol style="list-style-type: none"> <li>1: Compute the projections of <math>TS</math> on <math>\Sigma_{loc}(p)</math>, <math>p \in Proc</math>.</li> <li>2: Complement the language of transition system <math>TS</math>: To do this, we apply the determinization procedure for transition systems described in (the proof of) Corollary 2.19 and do not delete the sink state <math>\perp</math> (as proposed at that point). The complement of <math>L(TS)</math> is accepted by the constructed deterministic transition system (minimized, as optimization) with a single <i>final</i> state given by <math>\perp</math>.<sup>a</sup></li> <li>3: Construct a 1-safe Petri net <math>\langle N, M_0 \rangle</math> from the projections from step 1 and the finite automaton (with <math>\perp</math> as unique final state) from step 2 as follows: <ul style="list-style-type: none"> <li>▪ The <i>places</i> of the net <math>N</math> are the all states of the local transition systems constructed at steps 1 and 2.</li> <li>▪ The <i>transitions</i> of <math>N</math> are obtained by the synchronization on common labels of the local transitions of the constructed components.</li> <li>▪ The <i>initial marking</i> <math>M_0</math> is obtained placing one token on each of the places corresponding to the initial states of the transition systems.</li> </ul> </li> <li>4: Choose a set of ‘goal’ markings, denoted by <math>\mathcal{M}_\perp</math>, consisting of all the markings (of the net constructed at step 3), having one token on the place corresponding to the sink state <math>\perp</math>.</li> </ol>
<b>Output</b>	a 1-safe Petri net $\langle N, M_0 \rangle$ and a set of markings $\mathcal{M}_\perp$ , such that: The synthesis problem (via deterministic synchronous products) for $\Delta$ and $TS$ has a solution if and only if none of the markings from $\mathcal{M}_\perp$ is reachable from the initial marking $M_0$ in $N$ .

<sup>a</sup>Since the class of prefix-closed languages is not closed under complementation (Proposition 2.1), we move from the class of transition systems to that of finite automata that allow also final states (Definition 2.13).

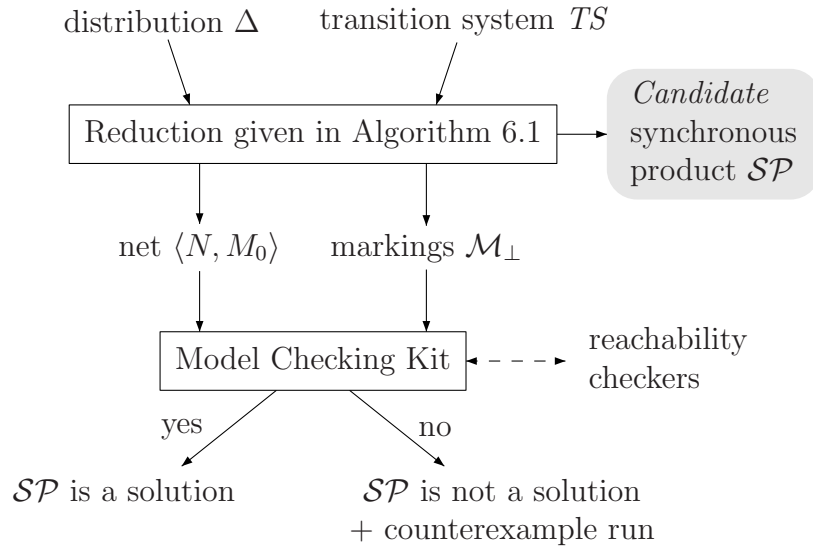


Figure 6.7: Implementation for the synthesis of deterministic synchronous products

mentioned before is to use the interface offered by the **Model Checking Kit**<sup>1</sup> (version 1.4) [SSE03]. We implemented<sup>2</sup> a compiler that takes as input a distribution and a transition system and constructs according to Algorithm 6.1 a net and a set of markings in one of the input formats recognized by the **Model Checking Kit**. Then, the **Model Checking Kit** calls the integrated reachability checkers (taking care of the particularities of each of them) and returns (if feasible) a positive answer or a negative one together with a counterexample. The above implementation flow is depicted in Figure 6.7.

We end with a couple of remarks (regarding possible optimizations):

**Remark 6.2** In the translation of Algorithm 6.1 (step 3), we construct the transitions of the net as *all possible* synchronizations on common labels of the local transitions. A way to reduce the number of local transitions (and therefore the number of their synchronizations) can be obtained by *determinizing* and then *minimizing* the local components (i.e., projections) before synchronizing their transitions. The overload of this procedure will be not very high in practice and it leads to very substantial reductions (of a couple of orders of magnitude) in the generated net.

**Remark 6.3** Another optimization is based on Corollary 3.35 showing that a necessary condition for a language  $L$  to be implementable as a deterministic synchronous product is to be forward- and trace-closed. According to Proposition 3.36, a prefix-closed regular

<sup>1</sup>A short description of the **Model Checking Kit** from its own (online) documentation: ‘The Model-Checking Kit is a collection of programs which allow to model a finite-state system using a variety of modeling languages, and verify it using a variety of checkers, including deadlock-checkers, *reachability-checkers*, and model-checkers for the temporal logics CTL and LTL. The most interesting feature of the Kit is that: Independently of the description language chosen by the user, (almost) all checkers can be applied to the same model. The counterexamples produced by the checker are presented to the user in terms of the description language used to model the system.’

<sup>2</sup>The programming of the reduction was done in C.



Table 6.2: Synthesis of deterministic synchronous products (modulo language equivalence)

Problem	Reduction to a 1-safe net			Solution	
	places	transitions	time	answer	time
$Mutex(2, \Delta_1)$	23	180	<0.01s	YES	<0.01s
$Mutex(2, \Delta_2)$	FD violated		<0.01s	NO	–
$Mutex(3, \Delta_1)$	135	12312	0.15s	YES	1.36s
$Mutex(3, \Delta_2)$	132	4536	0.06s	NO	0.19s
$Mutex(4, \Delta_1)$	out of memory			?	–
$Mutex(4, \Delta_2)$	1389	718776	37.34s	?	timeout
$Mutex(5, \Delta_1)$	out of memory			?	–
$Mutex(5, \Delta_2)$	out of memory			?	–
$Phil(2)$	FD violated		<0.01s	NO	–
$Phil(3)$	FD violated		<0.01s	NO	–
$Phil(4)$	FD violated		<0.01s	NO	–
$Phil(5)$	FD violated		<0.01s	NO	–

language  $L$  is forward- and trace-closed if and only if the minimal deterministic transition system accepting  $L$  satisfies ID and FD. Since in step 2 of Algorithm 6.1, we already construct the minimal deterministic transition system accepting  $L(TS)$ , we can test in *linear* time if the local properties ID and FD hold. If this is not the case, we simply stop with a negative answer, i.e., the given specification is not distributable as a deterministic synchronous product (and similarly, also not distributable as a deterministic asynchronous automaton). The above test may provide a fast negative answer (modulo of course the determinization procedure).

**Remark 6.4** Since synchronous products admit a description as asynchronous automata (Remark 3.19), in case of a *positive answer*, we obtain also an asynchronous automaton implementing the specification, so one does not have to go through the expensive Zielonka’s construction.

## Experimental Results

We apply the previous approach to the mutual exclusion and dining philosophers benchmarks. The experimental results are presented in Table 6.2. The first column gives the problem under consideration (see the dimensions of specifications in Table 6.1). In the following columns, we provide the results of the implementation of the reduction from Algorithm 6.1 (including the optimizations mentioned in Remarks 6.2 and 6.3). I.e., we give the number of places and transitions of the generated 1-safe Petri net and the time needed by the translation procedure. In the last two columns we report the output of the implementability test using the Model Checking Kit according to Figure 6.7. The given times (if available) were obtained using the reachability checker of *Mcsmodels* [Hel99], which proved to perform best on our examples (compared with the other reachability checkers offered by Model Checking Kit).

A YES entry in the answer column means that the synthesis for the given specification has a solution. As we can see this is the case only for *Mutex* with  $n = 2, 3$  and the first distribution  $\Delta_1$ . The synthesized deterministic synchronous product for  $n = 2$  was already described in Section 6.1.2 (see the local components in Figure 6.4). For  $n = 3$ , the local components will have at most 6 states each. According to Remark 6.4, the generated synchronous product can also be described as asynchronous automata.

A NO entry in the answer column means that the synthesis for the given specification has no solutions. In most cases, this was decided by the cheap test for the diamond rules ID and FD following Remark 6.3. We see that all benchmarks for philosophers fail the FD test, so we conclude with a negative answer (and we do not need to generate the 1-safe net). The only different case (when we have a negative answer) is *Mutex* for  $n = 3$  and the second distribution  $\Delta_2$ . In this case, the reachability checker returns also a counterexample (i.e., a sequence of actions executed by the synchronization of projections, that is not in the original specification, see the beginning of the section) of length 4:

$$\text{counterexample} := req_1 \text{ enter}_1 req_2 \text{ enter}_2. \quad (6.1)$$

The counterexample is similar to the one explained in detail in Section 6.1.3 (page 165).

A ‘?’ entry in the answer column means that state space explosion prevented us to have a definite answer to the problem (either running out of memory during translation part or ‘running out’ of time during the reachability test). This is the case for *Mutex* with  $n = 4, 5$ . However, the above counterexample for *Mutex*(3,  $\Delta_2$ ) (6.1) is still a counterexample for *Mutex*(4,  $\Delta_2$ ) and *Mutex*(5,  $\Delta_2$ ). This can be easily verified by hand (given the very short length of the counterexample). So, we should have **NO** entries also in the rows for *Mutex*(4,  $\Delta_2$ ) and *Mutex*(5,  $\Delta_2$ ) (however, we do not add them to Table 6.2, because these NO answers were not automatically computed).

## 6.3 Implementations for Asynchronous Automata

In this section we discuss a couple of (heuristic) implementations for asynchronous automata, first for the synthesis modulo isomorphism, then modulo language equivalence.

### 6.3.1 Synthesis modulo Isomorphism

The synthesis modulo isomorphism (Section 5.2) has its roots in the theory of regions and is solved using Theorems 3.30 (for the general, i.e., nondeterministic, specifications) and 3.32 (for deterministic ones). We describe our implementations for both cases, starting with the deterministic case.<sup>1</sup>

<sup>1</sup>For synthesis modulo isomorphism, we have no implementations yet for *synchronous products*. (Note that the implementability test modulo isomorphism for *asynchronous automata* has a lower complexity compared to synchronous products (compare condition  $SP_3$  of Theorem 3.26 to condition  $AA_3$  of Theorem 3.30) and the test will be positive for more specifications (asynchronous automata are more expressive, cf. Remark 3.19.)

### Deterministic case

The synthesis modulo isomorphism for *deterministic* asynchronous automata is based on Theorem 3.32. The input transition system must be *deterministic*, otherwise the problem has no solution, so a preliminary (linear time) test checks whether the input specification is indeed deterministic. If this is the case, we follow Algorithm 4.3 to construct (in polynomial time) the least set of local equivalences satisfying  $DAA_1$  and  $DAA_2$  (see Theorem 3.32). Then, we can synthesize an asynchronous automaton isomorphic to the given input if and only if conditions  $DAA_3$  and  $DAA_4$  hold (the test is performed again in polynomial time). The asynchronous automaton is constructed from the local equivalences as described in Section 5.2.1.

Our implementation of the above procedure was done in **C** and the experimental results are described at the end of the subsection (second and third column of Table 6.3).

### Nondeterministic case

Comparing the deterministic and nondeterministic cases, dealing with deterministic specifications or implementations has a computational advantage (P vs. NP, see Table 4.2). Nevertheless, it is worth considering also the nondeterministic case: In case the original specification is given as a regular expression, a nondeterministic transition system exhibiting the given behavior may be exponentially more succinct than a deterministic one with the same behavior. Moreover, the nondeterministic asynchronous automata are more expressive than their deterministic counterparts (see Figure 3.13). This encouraged us to consider also the implementability test and synthesis modulo isomorphism for *nondeterministic* transition system.<sup>1</sup>

The existing tools implementing the notion of regions for the synthesis of a distributed system (**Synet** [Cai97, BCD02], **Petrify** [CKK<sup>+</sup>97], and **Synasync** [SEM03]) do not directly handle nondeterministic specifications. In this sense, we can say that an implementation for the nondeterministic case widens the class of systems tackled. Of course, one can always determinize the specification and the behavior is preserved, but then the tests of *isomorphism* to a distributed system are incomparable w.r.t. the two inputs because the *structure* is changed in the process of determinization.

Our implementation was done in a constraint-based logic programming framework called **Smodels** (<http://www.tcs.hut.fi/Software/smodels/>). It consists of **smodels**, an efficient implementation of the stable model semantics for normal logic programs, and **lparse**, a grounder front-end that transforms normal logic programs (with variables) to ground logic programs (without variables). We translate the synthesis problem into the problem of finding a stable model of a logic program. The program itself is written in the input syntax of **lparse**. The synthesis solutions (if any!) are given as stable models of the input program. A more detailed description of the **Smodels** system can be found in the **lparse 1.0 User's Manual** (<http://www.tcs.hut.fi/Software/smodels/lparse.ps>) and a general tutorial on the stable model semantics in [SNS02].

We chose to use logic programs with stable model semantics over the more widely used SAT for the main reason that in the stable model semantics the notion of a least

<sup>1</sup>This implementation is joint work with Keijo Heljanko and appeared in [HS04].

fixpoint is trivial to express (we need this e.g., for testing that the specification is indeed reachable). However, in a SAT encoding we would have to encode the least fixpoint computation procedure itself (e.g., as a series of approximations) which would have resulted in additional blow-up of the translation not to mention implementation effort. An additional advantage of using **Smodels** is that the code is very concise. The actual implementation of synthesis without any optimizations (and comments) has no more than **20 lines of code** and it is given in the Appendix A.1. A couple of hints regarding the synthesis flow are given below.

We translate first the specification, i.e., the distribution and the transition, system into a format accepted by **lparse**. E.g., a (trivial) specification consisting of one action  $a$ , one process, and a transition system with one transition is translated to the following set of *facts*:

```
dom(a,1).  initialstate(0).  trans(0,a,1).
```

The logic program implementing the synthesis is a set of (normal) *rules* of the form

$$a :- b_1, \dots, b_n, \text{not } c_1, \dots, \text{not } c_m.$$

with the intuition that a (stable) model satisfying the rule, either contains  $a$  or does not contain one element from the right of the rule. As a simple example, a reachability predicate **reach(Q)** (used to guarantee that the transition system is indeed reachable) is implemented as follows:

```
reach(Q)   :-  initialstate(Q).
reach(Q2)  :-  reach(Q1), trans(Q1,A,Q2), neq(Q1,Q2).
```

The first rule in conjunction with the fact **initialstate(0)** forces **reach(0)** to appear in a model. The second rule, as expected, recursively derives **reach(Q2)** if there exists another state **Q1** that is reachable and  $Q1 \xrightarrow{A} Q2$  and (optionally, as optimization)  $Q1 \neq Q2$ .

The implementation of the test provided by Theorem 3.30 (that also generates the local states of the synthesized asynchronous automaton, in case there is one) is given in Appendix A.1. The implementation starts with the rules capturing the reachability and deadlock-free properties (that a specification must specify). Then, it guesses a set of local equivalences and test if they satisfy the **AA<sub>1</sub>–AA<sub>3</sub>** conditions of Theorem 3.30.

The prototype logic implementation presented here can be used as a starting point for more optimized synthesis procedures. Namely, the memory overheads involved in the prototype implementation are (polynomial but still) very significant in larger instances. A special purpose synthesis procedure (i.e., a special purpose NP-solver) could potentially eliminate quite a lot of this memory overhead. Also, it would be interesting to see how a recently introduced new translation of the stable model semantics to SAT [Jan03] performs on our examples.

## Experimental Results

We apply the previous approach to the mutual exclusion and dining philosophers benchmarks. The experimental results are presented in Table 6.3 that we explain below.

Table 6.3: Synthesis of asynchronous automata modulo isomorphism

Problem	Solution	Imperative Impl. (Det.)	Logic Impl.	
			Nondet.	Det.
<i>Mutex</i> (2, $\Delta_1$ )	YES	<0.01s	0.15s	0.13s
<i>Mutex</i> (2, $\Delta_2$ )	NO	<0.01s	0.28s	0.23s
<i>Mutex</i> (3, $\Delta_1$ )	YES	0.02s	528.64s	87.33s
<i>Mutex</i> (3, $\Delta_2$ )	NO	0.01s	25.03s	85.78s
<i>Mutex</i> (4, $\Delta_1$ )	YES	1.37s	–	–
<i>Mutex</i> (4, $\Delta_2$ )	NO	0.01s	–	–
<i>Mutex</i> (5, $\Delta_1$ )	YES	800.99s	–	–
<i>Mutex</i> (5, $\Delta_2$ )	NO	163.33s	–	–
<i>Phil</i> (2)	NO	0.01s	0.08s	0.05s
<i>Phil</i> (3)	NO	0.01s	1.34s	2.47s
<i>Phil</i> (4)	NO	0.01s	16.49s	102.59s
<i>Phil</i> (5)	NO	0.04s	195.88s	–

The first column gives the problem under consideration (see the dimensions of specifications in Table 6.1), while the second one provides the results of the implementability test modulo isomorphism for asynchronous automata. We see that the only positive cases are for *Mutex* problem with the first distribution  $\Delta_1$ . (In fact, we know already that the dining philosophers problem has no solutions from Table 6.2: We showed there that the input transition system violates FD, so according to Proposition 3.22 it cannot be the global state space of an asynchronous automaton. Anyway, we give the running times of the implementability test supposing we do not check ID and FD beforehand.)

The third column presents the running times (in seconds) of the imperative implementation done in  $\mathcal{C}$  described at the beginning of the subsection. The test is polynomial and is based on Theorem 3.32 (which works for *deterministic* specifications – as it is the case with our benchmarks).

The fourth column presents the running times (in seconds) of the logic implementation (using *Smodels*) described before (we optimized the code to accommodate various symmetry reductions). The implementation is based on the NP-complete (Theorem 4.11) test of Theorem 3.30 devised for general specifications. Despite our efforts to obtain smaller nondeterministic<sup>1</sup> input transition systems for our benchmarks, we ended up with deterministic transition systems (mainly due to complementation operations in the specifications).

Compared with the imperative implementation (column 3), the logic implementation performs much worse: *Mutex* 4 and 5 instances are not decided (an ‘–’ entry in the table stands for an ‘out of memory’ premature termination). However, there is a gain. The synthesis for the imperative implementation based on Theorem 3.32 produces asynchronous automata with the local state spaces as big as possible (because we construct the least local equivalences), which is not desirable. On the other hand, the logic program (based on

<sup>1</sup>I.e., where nondeterministic choice really occurs.

Theorem 3.30) involves a constant `max_local_state` which gives the maximum number of local states for each process. Tuning this constant, we can limit the logic program to look for synthesis solutions which are systems composed out of ‘small’ components. Moreover, we can very easily modify the logic implementation to search for (under)approximated solutions in case the specification is not distributable (see next section).

In the last column we give experimental results for a logic implementation based on the Theorem 3.32 (tailored to *deterministic* specifications). With the exception of *Mutex*(3,  $\Delta_1$ ), it is slightly slower than the logic implementation for general (nondeterministic) specifications. This is because the rules introduced by Theorem 3.32 will make `lparse` to produce larger grounded programs, `Smodels` having then to deal with a larger input problem.

### 6.3.2 Heuristics to Construct Under-Approximations

In this section we provide some heuristics for the unfortunate case when the specification is not distributable. Experience has shown (e.g., Tables 6.2 and 6.3, see also [Cai97, Yak98]) that most of the time this is indeed the case. Some approaches to tackle this problem have been proposed in the literature (most of them are at least NP-hard and at the level of heuristics): label-splitting, introduction of silent events (both not really suitable for our framework<sup>1</sup>), changing the distribution, changing the transition system by cutting edges that create problems. We opt for the last type of heuristic.

A necessary condition for a transition system (that was determinized and minimized) to be distributable is to satisfy the ID and FD diamond rules (Definition 3.21). The violation of any of them leads to ‘no solutions’ for the distributed synthesis problem (cf. Table 6.2). Our idea is to trim problematic transitions such that we end up with a *sub*-transition system (i.e., an isomorphic embedding, cf. Definition 4.46) satisfying ID and FD. We obtain thus an under-approximation of the original specification (we do not consider also over-approximations, i.e., adding edges, for reasons given at the beginning of Section 4.5).

In this section, we provide ways to construct isomorphic ID and FD embeddings for non-implementable specifications. At the end of the section, we also give a heuristic that looks for isomorphic embeddings that are already asynchronous automata.

Before going into details, we impose two important restrictions on the way we cut the transitions in order to have reasonable solutions:

- First, **all actions** of  $\Sigma$  should still be present in the under-approximation (i.e., the transition system obtained after cutting edges). E.g., it makes no sense to come up with an under-approximation for *Phil*(5) that has only, say, 3 philosophers specified, or as an extreme situation, to come up with the empty under-approximation. Moreover, we want the under-approximation to be *reachable* and *deadlock-free*.
- Second, for our benchmarks (Table 6.1) it is not advisable to cut *any* transition:

<sup>1</sup>The label splitting cannot solve the conflicts without changing the distribution and the silent events must have the whole set *Proc* as domain, and so we force a global synchronization, which is not something that we would like to do in a concurrent setting.



- For the *Mutex* problem we can cut  $enter_i$  actions (and thus to prevent at some point process  $i$  to enter its critical section). On the other hand, it is not all right to cut  $req_i$ , respectively  $exit_i$  actions, because the processes competing for a common resource should not be blocked in ‘showing interest in’, respectively ‘releasing’ a resource.
- For the *Phil* problem we can cut  $l_i$ , respectively  $r_i$  actions (and thus preventing at some point philosopher  $i$  to pick up her/his left, respectively right, fork). On the other hand, there are no reasons to cut  $rel_i$  actions, thus preventing a philosopher to release (i.e., put down) the forks s/he has at a certain time.

### Isomorphic ID-FD embedding

Below we give a couple of ideas to construct for a transition system  $TS$  an isomorphic embedding  $E \sqsubset TS$  satisfying ID and FD. Recall that finding an embedding  $E \sqsubset TS$  that is reachable, deadlock-free, and satisfies ID and FD, is an NP-complete problem (Corollaries 4.50 and 4.52). Starting with a deterministic transition system, an isomorphic ID-FD embedding is also deterministic. Moreover, minimizing a deterministic transition system satisfying ID and FD, still satisfies the diamond rules.

**Logic implementation (complete search)** We search for an isomorphic ID-FD embedding of a specification using a logic implementation based on **Smodels**. The idea is simple: We ‘guess’ a subset  $E$  of edges of the input transition system  $TS^1$ , then test if the transition system generated by  $E$  is reachable, deadlock-free, and satisfies ID and FD. **Smodels** will perform a *complete* search in the state space of solutions, returning the first one (or all of them, if we activate this option).

In the following, we give two heuristics that do not search the whole possible isomorphic embeddings; however, if lucky, they may find a solution very fast (the algorithms are *linear* in the size of the specification).

**Heuristic 1 (incomplete search)** The first heuristic works in a **top-down** manner trying to delete edges until the properties are satisfied. More precisely, starting with the initial transition system  $TS$ , we iteratively remove edges that prevent the properties ID and FD to hold. E.g., we have a conflict w.r.t. FD (i.e.,  $\exists q_1 \xrightarrow{a} q_2$  and  $\exists q_1 \xrightarrow{b} q_3$  with  $a \parallel b$ , but there exists no state  $q_4$  such that  $q_2 \xrightarrow{b} q_4$  and  $q_3 \xrightarrow{a} q_4$ ), we (nondeterministically)<sup>2</sup> remove one of the edges involved in the conflict (e.g., removing  $q_1 \xrightarrow{b} q_3$  will solve the conflict). A similar solution is applied to ID. Since we want also to have reachability and deadlock-freedom, after each step we remove all unreachable, respectively deadlock states (i.e., state with no outgoing edges). Moreover, since we also want to preserve the initial

<sup>1</sup>In the syntax of **lparse**, ‘guessing’ a subset of edges is simply realized by one line of code: `{subtrans(Q1,A,Q2)} :- trans(Q1,A,Q2)`, with the semantics that `subtrans(Q1,A,Q2)` may or may not be present in a stable model when `trans(Q1,A,Q2)` holds.

<sup>2</sup>In the current implementation we choose which of the two edges to be removed using an input file provided by the user containing pairs of (independent) actions specifying which of the two has *priority* over the other.



Table 6.4: Construction of under-approximations satisfying ID and FD

Problem	Logic impl.	Heuristic 1	Heuristic 2
$Mutex(2, \Delta_1)$	<0.01s	<0.01s	<0.01s
$Mutex(2, \Delta_2)$	<0.01s	<0.01s	<0.01s
$Mutex(3, \Delta_1)$	0.06s	<0.01s	<0.01s
$Mutex(3, \Delta_2)$	0.06s	<0.01s	<0.01s
$Mutex(4, \Delta_1)$	7.80s	<0.01s	<0.01s
$Mutex(4, \Delta_2)$	7.77s	<0.01s	<0.01s
$Mutex(5, \Delta_1)$	timeout	0.01s	0.05s
$Mutex(5, \Delta_2)$	timeout	0.01s	0.05s
$Phil(2)$	<0.01s	0.01s	0.01s
$Phil(3)$	0.02s	no sol. found	no sol. found
$Phil(4)$	21.24s	no sol. found	no sol. found
$Phil(5)$	timeout	no sol. found	no sol. found

alphabet of actions  $\Sigma$ , at each step we remove an edge labeled by  $a$  only if there exists at least another edge labeled by  $a$ . The algorithm stops either if all conditions are satisfied (and outputs the current under-approximation) or the conditions are still not satisfied and removing any other edge (involved in a violation) would lead to the eradication of an action (against the rule ‘all actions of  $\Sigma$  must be present’). In the latter case we *do not backtrack*, but report a failure of the heuristic (e.g., ‘no solution found during a heuristic *incomplete* search’).

**Heuristic 2 (incomplete search)** The second heuristic works in a **bottom-up** manner starting from scratch and adding edges until all properties are satisfied (if possible). More precisely, starting with only one initial state of initial transition system  $TS$ , we iteratively add edges in a (breadth-first) traversal of  $TS$  together with a ‘greedy’ strategy of selecting edges labeled by actions of  $\Sigma$  that were not chosen yet. At the same time, we avoid adding edges leading to violations of ID and FD rules and we add edges such no state is a deadlock (we assume  $TS$  is deadlock-free). The algorithm stops either if all conditions are satisfied (and outputs the current under-approximation) or the conditions are still not satisfied and adding any other edge would lead to a violation. In the latter case we *do not backtrack*, but report a failure of the heuristic (e.g., ‘no solution found during a heuristic *incomplete* search’).

**Experimental results** The running times of implementations for the three ideas presented above applied to our benchmarks are presented in Table 6.4.

The second column of Table 6.4 gives the running times for the logic implementation using **Smodels**. There was an **ID-FD embedding found** in all cases (this is not specifically mentioned in the table), except for  $Mutex(5)$  and  $Phil(5)$ , when we stopped without any answer after running the program for one hour. **Smodels** allows also to compute under-approximations with *maximal*, respectively *minimal* number of states (paying the price

of an increased complexity), but we have not switched on this option for the results in Table 6.4.

The last two columns of Table 6.4 give the running times for imperative implementations in  $\mathbf{C}$  for the two heuristics. They perform very well for *Mutex*, obtaining solutions also for *Mutex*(5) that is not solved by the logic implementation. In this case, the original specification already satisfies ID and FD and this helps the heuristics (which do not have much to do), whereas the logic implementation does not provide any results because it looks in the full space of solutions, which is huge. On the other hand, we do not obtain *any* solution for dining philosophers except for the easy *Phil*(2). This shows the weakness of an *incomplete* search of the space of solutions.

Regarding the size of the generated transition system, the first heuristic will produce larger solutions than the second one. On the one hand, larger solutions represent more behaviors, so richer implementations. On the other hand, larger isomorphic embeddings may be a hassle for Zielonka's procedure (that takes the embeddings as inputs) leading to a state space explosion (the experimental results in Table 6.6 witness this fact). That is why in general the second heuristic is preferred over the first one.

**Closure under independence** We mention now another attempt to generate specifications closed under ID and FD in special cases. Examining our benchmarks, we see that the pattern of the properties is to express *forbidden* behaviors and take their complement. At the end, we intersect all these complements and a base language (that is, the shuffle of local behaviors). An idea to obtain under-approximation that are trace- and forward-closed is to close the forbidden behaviors under independence. This is sound because if  $ab$  is undesirable and  $a||b$ , then we are forced to consider  $ba$  as undesirable too, since the system cannot execute the one without the other.<sup>1</sup> Thus, adding forbidden runs, we obtain a smaller specification (by complementation). Moreover, since the class of trace-closed languages is closed under complementation and intersection (Proposition 3.7), the new specification will be trace-closed.

Unfortunately, the trace-closure of a regular language is not always regular (Proposition 3.8). There are though a couple of results that might help. In [DR95, Chapter 12], sufficient conditions are given for a regular language to remain regular after closing it under an independence relation. Yet more practical seems to be [BMT01], where a specific class of regular languages, called *Alphabetic Pattern Constraints*, APCs, is proved to have the property that their closure under *any* independence relation is still regular, and can be effectively computed. The APCs have the form

$$\Sigma_0^* a_1 \Sigma_1^* a_2 \dots a_n \Sigma_n^*,$$

and capture the properties of our case studies. Unfortunately, the closure procedure in [BMT01] may lead to an exponential blowup in the size of the regular expression of the specification, which further makes difficult the translation to a transition system. Implementing the closure algorithm, we faced indeed this scalability problem, which rendered our attempt useless.

---

<sup>1</sup>Of course, an alternative would be to make  $a$  and  $b$  dependent, but we try to leave the distribution untouched.

Table 6.5: Constructing an under-approximation that is isomorphic to an asynchronous automaton

Problem	Logic Impl.	
	Nondet.	Det.
$Mutex(2, \Delta_1)$	0.19s	0.16s
$Mutex(2, \Delta_2)$	0.44s	0.28s
$Mutex(3, \Delta_1)$	601.65s	126.93s
$Mutex(3, \Delta_2)$	674.75s	147.33s
$Mutex(4)$	–	–
$Mutex(5)$	–	–
$Phil(2)$	0.06s	0.06s
$Phil(3)$	15.54s	3.49s
$Phil(4)$	–	–
$Phil(5)$	–	–

### A heuristic to find embedded asynchronous automata

Until now, we looked for isomorphic embeddings satisfying ID and FD. Following Remark 4.53, we could impose even stronger conditions and have a heuristic that looks for an isomorphic embedding that is already isomorphic to an asynchronous automaton. With such heuristic, the synthesized asynchronous automata will have the global state space *smaller* than the input transition system. Of course, the chances that no solutions are found increase in this case, but in case of success, we avoid Zielonka’s procedure which may, theoretically, produce asynchronous automata super-exponentially larger than the input.

We implemented the above idea again with logic programming using `Smodels`.<sup>1</sup> We simply ‘guess’ a subset of edges of the input transition system, and tests if the generated transition system is reachable, deadlock-free, and isomorphic to an asynchronous automaton. In fact, our implementation is easily obtained by minimally modifying the logic implementation for the implementability test described in Section 6.3.1 (see also Appendix A.1 and footnote 1 on page 183). The logic program code was optimized to include some rudimentary symmetry reductions and the relaxation obtained by dropping the condition  $AA_2$  (see Corollary 4.14 and Remark 5.1). Moreover, following the remarks at the beginning of this section (page 182), we only allow edges labeled by *enter* and *take fork* actions to be cut.

**Experimental results** Table 6.5 gives the running times for our logic implementations. Following the pattern from Section 6.3.1 (see the last two columns of Table 6.3), we considered two implementations based respectively on Theorem 3.30 (column 2) and 3.32

<sup>1</sup>We tried also an imperative implementation in C (without spending too much time with optimizations though), but it performed much worse than the logic implementation, so we do not report any results here.

(column 3). We stopped when the first solution was found by `Smodels`. ‘-’ entries in the table denote that we ran out of resources before having a definite answer.

As we can see the heuristic scales up only until  $n = 3$ . This is not very impressive; however, it beats at least the original construction of Zielonka which only works for  $n = 2$  (for *Mutex*) due to state space explosion (see Table 6.6 from the next section). Moreover, the gain of this heuristic is that it makes a *complete* search in the class of ‘small’ asynchronous automata whose global transition system is isomorphically embedded in the specification (so we are guaranteed solutions not exceeding the size of the specification).

**Remark 6.5** There are alternative ways to find stable models of a logic program. The `Cmodels` system (<http://www.cs.utexas.edu/users/tag/cmodels.html>) is a viable alternative to `Smodels` (neither of the systems is consistently better than the other). Our logic implementations described in Tables 6.4 and 6.5 were used as benchmarks in papers describing `Cmodels` (in comparison with `Smodels`): [GLM04a, Table 3] and [GLM04b, Tables 3 and 8]). `Cmodels` has sometimes better running times. Moreover, it manages to solve also *Phil(4)* from Table 6.5 (not solved by `Smodels`) in about 12 minutes.

### 6.3.3 A Heuristic using Unfoldings for Zielonka’s Construction

We discuss now the synthesis modulo *language equivalence* of deterministic asynchronous automata based on Zielonka’s construction (see Section 5.2.2).

In practice, the distributed implementations are required to be small in order to be efficient. Moreover, they should usually be deterministic. In this section we present a method that heuristically tries to synthesize asynchronous automata smaller than the ones generated by Zielonka’s equivalence. One could think of several ways to achieve this:

- One idea is to construct Zielonka’s asynchronous automaton and then try to minimize it. A real danger in doing so is that one runs out of resources before completing Zielonka’s procedure (due to the state space explosion problem mentioned above). Moreover, there is no consistent approach of minimizing an asynchronous automaton ([BPS94, Pig93a, Pig94] show for instance that the category of deterministic asynchronous automata accepting a regular trace language does not admit a unique minimal element). Since an asynchronous automaton can be seen as a particular case of labeled 1-safe Petri net, one could try to apply reduction techniques for Petri nets (e.g., [AS92, SS00]), but they do not fit well with model of asynchronous automata (so far we have looked into this direction).

For small asynchronous automata, we can use a brute force trail-and-error method where we heuristically merge two different local states and check if the new asynchronous automaton accepts the same language as the original one.

- Another idea to synthesize a small asynchronous automaton for a given specification  $L$  is to fix a bound  $k \in \mathbb{N}$ , generate all the deterministic transition systems  $TS$  with the size smaller than  $k$  such that  $L(TS) = L$ , and test each generated transition system whether it is isomorphic to an asynchronous automaton (using the polynomial test of Theorem 3.32). The limitation of such approach is that of any brute force method, namely, it is time/space expensive. (In a similar vein, [BDT03]

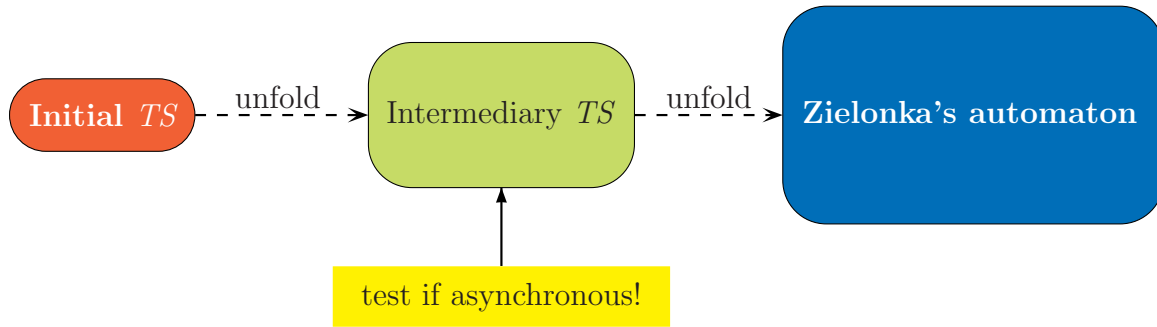


Figure 6.8: A heuristic using unfoldings and an implementability test

presents a methodology of synthesizing mutual exclusion algorithms by generating algorithms bounded by several parameters and verifying whether they implement a mutual exclusion situation.)

- The heuristic presented in this section combines in a sense the previous two ideas using an unfolding technique and an implementability test (see Figure 6.8).

We start with the minimal deterministic transition system (cf. Corollary 2.19) and unfold<sup>1</sup> it stepwise using Zielonka's equivalence, testing at each step whether the transition system is isomorphic to an asynchronous automaton (using the polynomial test of Theorem 3.32).

The above method does not bring any theoretical improvement over Zielonka's approach, in worst case the procedure generating (the big) Zielonka's automaton. The state space explosion problem is augmented by the risk of working with global transition systems (rather than local ones), but this is the price to pay for being able to use the implementability test (of Theorem 3.32) on the intermediary transition systems<sup>2</sup>. Nonetheless, by the above heuristic we avoid generating the whole Zielonka's automaton in case an intermediary transition system is already isomorphic to an asynchronous automaton, thus saving the time and space of generating Zielonka's automaton and obtaining a smaller automaton. The heuristic behaved indeed well on the examples we considered so far (see the experimental results at the end of the section).

The modified construction of Zielonka's asynchronous automaton in the Section 5.2.2 provides an algorithm to automatically derive an asynchronous automaton from a prefix-closed forward-closed trace-closed regular language  $T \subseteq \Sigma^*$ . Zielonka defines an effectively computable equivalence relation  $\approx \subseteq \Sigma^* \times \Sigma^*$  of finite index which generates the global state space given in (5.3) on page 136. Thus, we can define another 'global' equivalence

<sup>1</sup>In a similar way that a graph is unfolded into a tree.

<sup>2</sup>One could envisage another heuristic based on unfoldings where one starts with local state spaces that are stepwise unfolded towards Zielonka's asynchronous automaton testing at each step whether the *language* of the current asynchronous automaton equals the language of the specification. However, the language equivalence test is too expensive to perform it repeatedly.

$\approx_Z \subseteq \Sigma^* \times \Sigma^*$ , for  $t, t' \in \Sigma^*$  as:

$$t \approx_Z t' \text{ if and only if } \forall i \in Proc : P_i(t) \approx P_i(t').$$

Now, let  $\mathcal{T}$  be the transition system having  $T$  as set of states, and  $w \xrightarrow{a} wa$  as transitions<sup>1</sup>. Then, from (5.3) on page 136, (the global state space of) the synthesized asynchronous automaton is the quotient of  $\mathcal{T}$  under  $\approx_Z$  (note that since  $\approx$  is of finite index, also  $\approx_Z$  is of finite index).

We give a version of the synthesis algorithm based on the equivalence  $\approx_Z$ . The version is tailored so that we can easily add an implementability test to intermediary transition systems. Loosely speaking, the algorithm proceeds by *unfolding* the minimal deterministic transition system accepting  $T$  until an asynchronous automaton is obtained.

**Data structure** The algorithm maintains a deterministic transition system  $TS = (Q, \Sigma, \rightarrow, \{q_0\})$ . The transitions of  $TS$  are colored *green*, *red*, or *black*<sup>2</sup>. The algorithm keeps the following **invariants**:

1. The transition system  $TS$  accepts the specification language  $T$ .
2. **Green** transitions form a directed spanning tree of  $TS$ , i.e., a directed tree with the initial state  $q_0$  as root and containing all states of  $TS$ .

We denote by  $W(q)$  the unique word  $w$  such that there is a path  $q_0 \xrightarrow{w} q$  in the spanning-tree.

(From the definition of  $W(q)$ , it immediately follows that for any *green* transition  $q \xrightarrow{a} q'$ ,  $W(q) \cdot a = W(q')$ .)

3. A transition  $q \xrightarrow{a} q'$  is **red** if  $W(q) \cdot a \not\approx_Z W(q')$ .
4. All other transitions are **black**.

(In particular, all *black* transitions  $q \xrightarrow{a} q'$  satisfy  $W(q) \cdot a \approx_Z W(q')$ .)

Algorithm 6.2 provides an unfolding procedure that constructs Zielonka's automaton. Its steps are commented below (we show later on also a sample run of the algorithm in Figure 6.9):

**line 0**  $TS$  is initialized to the minimal deterministic transition system  $TS_0$  accepting  $T$ .  $TS$  will be gradually unfolded and each of its states  $q$  will exhibit the same behavior (i.e., the set of all possible executions starting in that state) as a certain state  $peer(q)$  from the original transition system  $TS_0$ . This correspondence will be kept by the *peer* function, also initialized at this step.

**line 1** The transitions of  $TS$  are colored according to the above data structure.

**line 2** We start a cycle that runs as long as there are still *red* transitions in  $TS$  (the body of the cycle tries to decrease the number of *red* transitions).

<sup>1</sup>Note that  $\mathcal{T}$  is infinite if and only if  $T$  is infinite.

<sup>2</sup>The presentation of the algorithm using colors was suggested by Javier Esparza.

Algorithm 6.2: An unfolding approach for the synthesis of deterministic asynchronous automata (based on Zielonka's equivalence)

<b>Input</b>	a distribution $(\Sigma, Proc, \Delta)$ and a prefix-closed forward-closed trace-closed regular language $T \subseteq \Sigma^*$
	<p>0: construct <math>TS_0 = (Q_0, \Sigma, \rightarrow_0, \{q_0\})</math> the minimal deterministic transition system accepting <math>T</math>, initialize <math>TS</math> to <math>TS_0</math>, and set <math>peer(q) := q</math>, for all <math>q \in Q_0</math></p> <p>1: color the transitions of <math>TS</math> according to the invariants given in the data structure given on page 189 (The set of <i>green</i> transitions can be computed by means of a, say, breadth-first traversal of <math>TS</math> starting from the initial state. The other colors are computed to satisfy the invariants.)</p> <p>2: <b>while</b> there are still <i>red</i> transitions in <math>TS</math> <b>do</b></p> <p>3:     choose a <i>red</i> transition <math>q \xrightarrow{a} q'</math>,</p> <p>4:     delete the transition <math>q \xrightarrow{a} q'</math>, and</p> <p>5:     <b>if</b> there is a state <math>r</math> such that <math>W(q) \cdot a \approx_Z W(r)</math> <b>then</b></p> <p>6:         add a <i>black</i> transition <math>q \xrightarrow{a} r</math>,</p> <p>7:     <b>else</b></p> <p>8:         create a new state <math>q_{new}</math>, setting <math>peer(q_{new}) := peer(q')</math>,</p> <p>9:         add a new <i>green</i> transition <math>q \xrightarrow{a} q_{new}</math>, and</p> <p>10:         <b>for</b> every transition <math>peer(q_{new}) \xrightarrow{b}_0 s</math> <b>do</b></p> <p>11:             add a new transition <math>q_{new} \xrightarrow{b} s</math> and</p> <p>12:             <b>if</b> <math>W(q_{new}) \cdot b \not\approx_Z W(s)</math> <b>then</b></p> <p>13:                 color <math>q_{new} \xrightarrow{b} s</math> <i>red</i>,</p> <p>14:             <b>else</b></p> <p>15:                 color <math>q_{new} \xrightarrow{b} s</math> <i>black</i>.</p>
<b>Output</b>	a transition system $TS$ that is isomorphic to an asynchronous automaton accepting $T$



**lines 3,4** We pick a *red* transition  $q \xrightarrow{a} q'$  and delete it. A *red* transition is a ‘persona non grata’ because it cannot appear in Zielonka’s construction (due to its characterization  $W(q) \cdot a \not\approx_Z W(q')$ ). However, in order to preserve the behavior, we need to have an outgoing transition from  $q$  labeled by  $a$ , introduced in the next lines.

**lines 5,6** We add  $q \xrightarrow{a} r$  if  $W(q) \cdot a \approx_Z W(r)$ , and we mark it as *black* according to the invariants of the data structure.

**lines 7,8,9** Otherwise, we **unfold** the *red* transition  $q \xrightarrow{a} q'$  into  $q \xrightarrow{a} q_{new}$ , where  $q_{new}$  is a fresh state.

The behavior of  $q_{new}$  should be equal to the one of  $q'$ . We mark this by the assignment  $peer(q_{new}) := peer(q')$ . This will be realized by the last for-loop.

Moreover, since we turn transition  $q \xrightarrow{a} q_{new}$  *green*, we have  $W(q_{new}) := W(q) \cdot a$  (see definition of  $W(q)$  in the data structure).

**lines 10,11** In order to preserve the language after each unfolding, we add from  $q_{new}$  one outgoing transition, for *each* outgoing transition of its peer (note that all peers belong to the initial set of states  $Q_0$ , so it is sound to use also the original transition  $\rightarrow_0$  together with the peer in the loop condition).

**lines 12,13,14,15** Finally, we color the new transitions  $q_{new} \xrightarrow{b} s$  such that the invariants described in the data structure are satisfied.

The correctness of Algorithm 6.2 is given by the following proposition.

**Proposition 6.6** *Algorithm 6.2 always terminates and its output is (the global state space of) a deterministic asynchronous automaton accepting the language  $T$ .*

**Proof.** The termination of the algorithm relies on the *finite* index of  $\approx_Z$ . On the one hand, each time the while-loop is executed, the number of *red* transitions is decremented by one in lines 3,4. On the other hand, new *red* transitions may be added only when an unfolding occurs in the else-branch of the if-conditional on line 5, which implies  $W(q) \cdot a \not\approx_Z W(r)$  for all current states  $r$ . Moreover,  $W(q_{new})$  for the new state  $q_{new}$  is equal to  $W(q) \cdot a$  (because  $q \xrightarrow{a} q_{new}$  is a *green* transition). Since the number of equivalence classes w.r.t.  $\approx_Z$  is finite, at some point only the first branch of the if-conditional on line 5 will be followed and from that point on the number of *red* transitions will strictly decrease with each while-loop, eventually leaving no *red* transitions in the system, thus terminating the algorithm.

Second, the transition system  $TS$  preserves at each step the invariants defined in the in the data structure. The initial minimal transition system satisfies the invariants by construction (see lines 0,1). Then,  $L(TS) = T$  at each step, because at each unfolding we add only the edges dictated by the minimal transition system (see line 10). The new transitions are properly colored by construction.<sup>1</sup>

<sup>1</sup>Note that  $TS$  and the minimal transition system  $TS_0$  not only accept the same language, but they are even bisimilar under the relation  $\{(q, peer(q)) \mid q \text{ state of } TS\}$  (see also Section 4.4).

Finally, at the end of the algorithm, we have only *green* and *black* edges in  $TS$ . According to the invariants, this means that all  $q \xrightarrow{a} q'$  of the output satisfy  $W(q) \cdot a \approx_Z W(q')$ . Therefore, the output  $TS$  is isomorphic to quotient under  $\approx_Z$  of the transition system having  $T \subseteq \Sigma^*$  as state space and  $w \xrightarrow{a} wa$  as transitions. But this is exactly the (deterministic) asynchronous automaton constructed in Section 5.2.2.  $\square$

**Heuristic intermediate testing** Unfortunately, as confirmed by experimental results, the Algorithm 6.2 can produce transition systems with many more states than necessary. We have implemented a heuristic (see Figure 6.8) that allows to ‘stop earlier’ if the transition system constructed so far happens to be already a solution. To achieve this, we insert at the beginning of the body of the while-loop (i.e., between lines 2 and 3) of Algorithm 6.2 the following:

**if**  $TS$  is isomorphic to an asynchronous automaton **then**  
output  $TS$  and **stop**

For the condition above, we use again Theorem 3.32 as implementability test modulo isomorphism (for deterministic specifications). In fact, we can even use the relaxation mentioned in Remark 5.1), i.e., dropping  $DAA_3$  condition.

As we will see (Table 6.6), the little trick above works pretty well. Once  $TS$  passes the test, we can derive the local structure of the asynchronous automaton as described in Section 5.2.1. Moreover, if the test fails, it can provide however guidance to select the *red* transition at line 3 as explained below.

**Which *red* transition to unfold at line 3 of Algorithm 6.2?** In Algorithm 6.2, Zielonka’s equivalence  $\approx_Z$  tells *how* (and when) to unfold the *red* transitions (lines 5,12). In case the isomorphism test (introduced in the previous paragraph) gives a *negative* answer, Theorem 3.32 (with  $DAA_3$  dropped) can tell us *where* there is need to unfold. We sketch how to do this:

- Let  $TS_{g/b}$  be the transition system obtained after selecting the *green* and *black* edges of  $TS$ . For the isomorphic embedding  $TS_{g/b}$  we compute the least family of local equivalences  $(\equiv_p^{g/b})_{p \in Proc}$  satisfying  $DAA_1$  and  $DAA_2$  (of Theorem 3.32). Then, these equivalences also satisfy  $DAA_4$ . The reason is that all the *green* and *black* transitions appear in the transition system, denoted by  $TS_Z$ , generated by Zielonka’s approach, so the equivalences  $(\equiv_p^{g/b})_{p \in Proc}$  are *included* in the least local equivalences satisfying  $DAA_1$  and  $DAA_2$  for  $TS_Z$ . If by contradiction  $(\equiv_p^{g/b})_{p \in Proc}$  violated  $DAA_4$ , this violation could be simulated also in  $TS_Z$ , but this is a contradiction, because we know that  $TS_Z$  is isomorphic to an asynchronous automaton.
- Starting with  $TS_{g/b}$ , we iteratively add *red* transitions (from  $TS$ ) and reconstruct the least family of local equivalences satisfying  $DAA_1$  and  $DAA_2$ , until  $DAA_4$  is violated (this will eventually happen, since  $TS$  fails the test).
- The last added *red* transition (that leads to a violation) is selected (as candidate for unfolding) at line 3 of Algorithm 6.2.

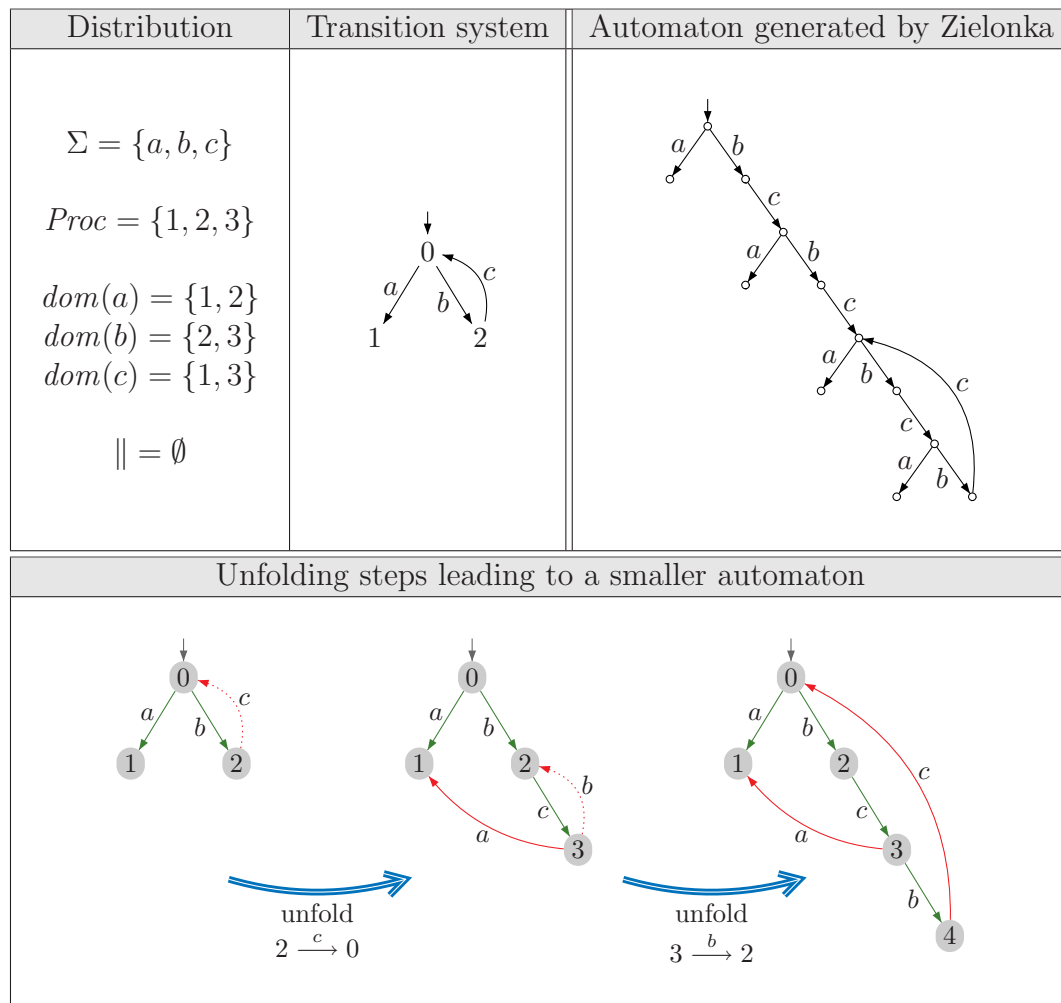


Figure 6.9: Example of the unfolding compared with Zielonka's construction

**Example 6.7** We exemplify the unfolding procedure with the specification given in Figure 6.9 (top-left). We have a distribution of three actions over three processes, with an empty independence relation, together with a transition system with three states. Applying original Zielonka's procedure implemented by Algorithm 6.2, we obtain the global transition system at the top-right of Figure 6.9 having 12 states. At the bottom of the figure, we give the first two unfolding steps:

- First we identify  $0 \xrightarrow{a} 1$  and  $0 \xrightarrow{b} 2$  as *green* edges, and  $2 \xrightarrow{c} 0$  as *red* edge. The asynchronicity test from the previous paragraph gives a negative answer (the reasons are given on page 147 – see Figure 5.6). Therefore, we proceed to unfold  $2 \xrightarrow{c} 0$ . We do this by removing  $2 \xrightarrow{c} 0$  and adding a *green* edge  $2 \xrightarrow{c} 3$  with  $peer(3) = 0$ . Moreover, we construct edges from 3 with same labels and destination nodes as from 0, i.e., we add  $3 \xrightarrow{a} 1$  and  $3 \xrightarrow{b} 2$ . According to the algorithm both of them are colored in *red*.
- We test again if the new transition system is a solution, but it is not the case. Then,

the *red* edge  $3 \xrightarrow{b} 2$  is automatically proposed for unfold (we add a new *green* edge  $3 \xrightarrow{b} 4$  with  $peer(4) = 2$ ), leading to the transition system at the bottom right of Figure 6.9.

- The test for the third transition system is now positive, so we stop (and construct the local components of the asynchronous automaton – not showed in the figure).

Thus, the above heuristic produces a solution with only 5 (global) states, which is less than half of the state space (12 states) generated by Zielonka’s approach.

**Obtaining smaller asynchronous automata** Some observations regarding the construction of smaller asynchronous automata accepting a certain language (in case we found one solution) are given already at the beginning of the subsection (see page 187). We add a couple of remarks below.

One idea is to use the isomorphism test of Theorem 3.30 whose logic implementation is described in Section 6.3.1 (see page 179). The trick is to tune the constant `max_local_states` that provides the upper bound for the (size of) local components that the program will search for. One approach is to start with a small value of constant `max_local_states` and increment it until a solution is found (if `max_local_states` is too small, the logic implementation may return a negative answer, which is correct in the sense that ‘there exist no asynchronous automata accepting the given language having all components smaller than the current `max_local_states`’).

Another heuristic is to arbitrarily *merge* local states of the components while the accepted language is preserved. This is an available option in our current implementation and showed to decrease the sizes of the local components. However, the procedure is rather time-expensive and does not really scale up.

Yet another idea is given by the following result:

**Proposition 6.8** *For a distribution  $(\Sigma, Proc, \Delta)$ , a transition system  $TS$ , and a state  $q$  from  $TS$ , we denote by  $TS(q)$  the isomorphic embedding  $TS(q) \sqsubset TS$ , having  $q$  as initial state and obtained by selecting from  $TS$  all the states and transitions reachable from  $q$ . Then, if  $TS$  is isomorphic to an asynchronous automaton over  $\Delta$ , then for any state  $q$  of  $TS$ ,  $TS(q)$  is also isomorphic to an asynchronous automaton over  $\Delta$ .<sup>1</sup>*

**Proof.** Let  $TS = (Q, \Sigma, \rightarrow, I)$  be a transition system that is isomorphic to an asynchronous automaton over  $\Delta$ . According to Theorem 3.30, there exists a set of local equivalences  $(\equiv_p)_{p \in Proc}$  satisfying  $AA_1$ – $AA_3$ , where  $\equiv_p \subseteq Q \times Q$ . We consider a set of local equivalences for  $TS(q) = (Q_q, \Sigma, \rightarrow_q, \{q\})$ , denoted by  $(\equiv_p^{(q)})_{p \in Proc}$ , obtained by restricting each equivalence  $\equiv_p$  to  $Q_q$ , i.e.,  $\equiv_p^{(q)} := \equiv_p \cap (Q_q \times Q_q)$  for all  $p \in Proc$ .

Since  $(\equiv_p)_{p \in Proc}$  satisfies the conditions  $AA_1$ – $AA_3$ , it is easy to argue that  $(\equiv_p^{(q)})_{p \in Proc}$  satisfy the same conditions (the reachability of  $TS(q)$  is used for  $AA_3$ !). Applying Theorem 3.30, we have that  $TS(q)$  is also isomorphic to an asynchronous automaton over  $\Delta$ . □

<sup>1</sup>In fact, the result can be generalized to any isomorphic embedding  $E$  of  $TS$  that is ‘closed’ under *reachability* (i.e., for any state  $q$  of  $E$ , if  $q \xrightarrow{a} q'$  in  $TS$ , then also  $q \xrightarrow{a} q'$  in  $E$ ) rather than to only  $TS(q)$ . The proof is the same.

Table 6.6: Synthesis of (deterministic) asynchronous automata

Problem	Zielonka(1)		Zielonka(2)		Heuristic(1)		Heuristic(2)	
	size	time	size	time	size	time	size	time
$Mutex(2, \Delta_1)$	34	<0.01s	23	<0.01s	14	<0.01s	10	<0.01s
$Mutex(2, \Delta_2)$	4799	4.09s	2834	2.00s	17	<0.01s	16	<0.01s
$Mutex(3, \Delta_1)$	–	–	–	–	107	<0.01	30	<0.01s
$Mutex(3, \Delta_2)$	–	–	–	–	–	–	58	0.07s
$Mutex(4, \Delta_1)$	–	–	–	–	1340	0.24s	69	<0.01s
$Mutex(4, \Delta_2)$	–	–	–	–	–	–	157	1.87s
$Mutex(5, \Delta_1)$	–	–	–	–	25337	152.91s	147	0.01s
$Mutex(5, \Delta_2)$	–	–	–	–	–	–	387	55.03s
$Phil(2)$	71 0.01s				5 <0.01s			
$Phil(3)$	–				12 <0.01s			
$Phil(4)$	–				49 <0.01s			
$Phil(5)$	n/a (no distributable specification to start with)							

**Corollary 6.9** *Let  $TS$  be the global state space of an asynchronous automaton such that  $L(TS) = T$ . Then, we may find a smaller asynchronous automata accepting  $T$  just looking for states  $q$  of  $TS$  whose reachable part  $TS(q)$  accepts the same language as  $TS$ .*

In case the asynchronous automaton  $TS$  was generated by Algorithm 6.2, we can use the available *peer* information for testing the language equality in the previous corollary. More precisely, two states  $q$  and  $q'$  have the same behavior, i.e.,  $L(TS(q)) = L(TS(q'))$ , if and only if  $peer(q) = peer(q')$ . Therefore, we can find the states  $q$  such that  $L(TS(q)) = L(TS)$  by checking if  $peer(q) = peer(q_0) = q_0$  (where  $q_0$  is the initial state of the deterministic transition system on which we run the algorithm).

The above idea is exemplified on the transition system generated by Zielonka's procedure depicted in Figure 6.9 (top-right). The state reached from the initial state after executing the sequence  $bcbc$ , denoted by  $q_{bcbc}$ , has the same behavior as the initial state (their peers are equal), i.e.,  $L(TS_Z(q_{bcbc})) = L(TS_Z)$ . According Proposition 6.8,  $TS_Z(q_{bcbc})$  is isomorphic to an asynchronous automata over the given distribution. Moreover,  $TS_Z(q_{bcbc})$  has only 6 states, which is close to the result of the heuristic (depicted at the bottom of Figure 6.9), which has 5 states. (Compare it also with the ad-hoc solution at the top of Figure 5.6).

## Experimental results

Table 6.6 compares the performances and outputs of Zielonka's construction and the heuristic introduced in this section on our benchmarks (see Table 6.1), for our (imperative) implementations programmed in C.

For each solved instance, we give the number of *global* states of the synthesized asynchronous automaton together with the time (in seconds) necessary to construct it. We imposed an upper limit of 32000 of global states, stopping after reaching this limit, and

denoting this by a ‘-’ entry in our table. We imposed this limit given: (a) the high memory consumption associated with Zielonka’s procedure (that has to keep a lot of information for further reference, e.g., up to 1 GB of RAM is used for 32000 of generated global states) and (b) the size of the specifications, i.e., a solution with 32000 global states for, say, *Mutex*(3) is impracticable.

We take first into discussion the *Mutex* problem. The second and third columns of Table 6.6 give the results after applying Zielonka’s method (implemented by Algorithm 6.2). The two versions (1) and (2) denote the fact that we used as starting specifications the output of the two heuristics described in Table 6.4 (remember that we can synthesize deterministic asynchronous automata only when the original specification satisfies ID and FD). As we can see, Zielonka’s approach is highly *inefficient* only working for  $n = 2$ .

The last two columns of Table 6.6 give the results of the heuristic presented in this subsection, that combines Zielonka’s approach with the testing of the intermediate transition system (see also Figure 6.8). (We start again with the outputs of the two heuristic described in Table 6.4, so we have two columns of results.) As we can see, the simple idea of intermediate testing performs extremely well: We solve all the instances of *Mutex* ( $n = 2..5$ ) in the last column using heuristic (2). For heuristic (1) we perform almost as good, solving all the instances for the first distribution  $\Delta_1$ , but only for  $n = 2$  with  $\Delta_2$ . The reason behind the lack of success in the latter case is the size of the input transition system: The first heuristics (1) constructs ID-FD embeddings as large as possible (cutting only edges that spoil the diamond properties), thus leading to a state space explosion. On the other hand, we still get solutions for  $\Delta_1$ , because the input specification is already isomorphic to an asynchronous automaton (see Table 6.3), so our heuristic will successfully stop after the first step (without having to unfold any transition).

For the dining philosophers problem, we cannot use the output of the two heuristics described in (the last two columns of) Table 6.4, because they cannot find any ID-FD under-approximations. In this case, we use the output of the logic implementation using *Smodels*. However, since we do not have any solution for  $n = 5$ , we cannot even start the unfolding procedure for *Phil*(5). For the other cases ( $n = 2, 3, 4$ ), we see again that the heuristic proposed in this section easily outperforms Zielonka’s approach: The latter only provides a solution for  $n = 2$ , while the former scales up for all inputs ( $n = 2..4$ ).

## Discussion

In this chapter we showed the synthesis paradigm applied to a couple of classical problems. We considered both synchronous products and asynchronous automata, with a focus on the latter. Below we make a couple of retrospective remarks:

**Scalability** Unfortunately, synthesis based on Zielonka’s procedure not only has a bad worst-case complexity, but also behaves bad in practice, as shown for instance by our experimental results (cf. Table 6.6). We proposed a couple of new heuristics that use (variants of) an implementability test modulo isomorphism to speed up the synthesis procedure.

**Running time** In case an instance is indeed distributable, a solution is usually quickly



generated (using the heuristics).

**Obtaining ‘elegant’ solutions** Looking for instance at Figure 6.6, we see that our solutions are not necessarily ‘elegant’: They use variables with larger domains than those appearing in the literature, and a human finds it difficult to understand why they are correct. Notice, however, that this is the case with virtually all computer generated outputs, whether they are HTML text, program code, or a computer generated proof of a formula in a logic. Our solutions are correct and relatively small. Notice also that our solution from Figure 6.6 to the *Mutex*(2) problem is in a sense ‘the’ optimal solution (in that context), i.e., it excludes only those behaviors strictly forbidden by mutual exclusion and absence of starvation.

**Specifying with temporal logic** Our approach is compatible with giving the global specifications as LTL temporal logic formulas over finite strings, since the language of finite words satisfying a formula is known to be regular, and a transition system recognizing this language can be effectively computed.

**Dealing with liveness properties** Currently our approach cannot deal with proper liveness properties. Loosely speaking, ‘eventually’ properties have to be transformed into properties of the form ‘before this or that happens’. Dealing with liveness properties requires to consider the theory of asynchronous automata on *infinite words*, for which not much is known yet (see [DR95, Chapter 11]). The approaches of [BCD02, CMT99] take a transition system as specification, and so do not consider liveness properties either.

**Related work** We are not aware of any other implementation for the synthesis of *asynchronous automata*, so our work is the first one to give support to the well-developed theory of this class of distributed transition systems.

Regarding the synthesis of distributed algorithms like *Mutex*, our methods scale up to 5 processes. Compared with other solutions in the literature we are doing well: The method of [CE82] implemented in [Ina84] scales up to 4 processes. The heuristic of [BDT03] deals only with 2 processes. [Cai97, BCD02] also exemplify their method for only 2 processes; moreover, their approach is not fully automatic (manual modification of the specification was needed), and they do not take any liveness into account. The approach of [MT02] may also be applicable to the problem, but there are no experimental results (implementations) available.

More promising is the recent work of Attie and Emerson [AE98, Att99, AE01] (building upon [CE82]). In [AE98, Att99], they advertize avoiding the state space explosion by constructing local machines for pairwise communication. However, similar to [CE82], the synthesized models have the disadvantage of high-atomicity (many actions performed in one atomic step). In [AE01], they offer a solution to lower the high atomicity, but unfortunately, the method only works with the method of [CE82], but not with those of [AE98, Att99] (see also the discussion at the end of Section 6.1.1).

Of course, all the above comparisons should be taken with a grain of salt, given the fact that we use different specifications, respectively models for distributed systems.



**Possible future work** Our current conviction is that it is unlikely that synthesis based on *asynchronous automata* would ever be successful on other than relatively small examples, due to the very high inherent complexity. Therefore, more work could be invested in heuristics generating small *distributable* specifications (in the line of Section 6.3.2) or small distributed implementations (in the line of Section 6.3.3).

Regarding the synthesis of *synchronous products* (modulo language equivalence), further optimizations may be explored (note that unfolding techniques as used for asynchronous automata do no work for synchronous products). In case the specification is not distributable and a counterexample is provided, we could use this counterexample to cut behaviors and then reiterate (i.e., counterexample-guided refinement of the specification).

Also, other interesting *case studies* from areas like e.g., asynchronous circuits [Yak98, CKK<sup>+</sup>00, CKK<sup>+</sup>02] or security protocols [Sai02] may be explored.



---

If I were a medical man, I should  
prescribe a holiday to any patient  
who considered his work important.

*Bertrand Russell*

## CHAPTER 7

## CONCLUSIONS

---

THIS thesis is important because it takes theory a few steps closer to practice. The main **outcomes** of this work are:

- A careful study of the languages and various properties of the classes of synchronous products and asynchronous automata (Chapter 3).
- An almost complete map of computational complexity results for the distributed implementability problem (Chapter 4). These classifications help in choosing an appropriate implementation technique for the problem at hand. E.g., since we see that the deterministic case is usually easier, working with deterministic systems is more efficient.
- A survey of synthesis methods with a couple of new algorithms for special cases (Chapter 5).
- Prototype implementations for most of the algorithms described in the thesis, together with various heuristics that proved successful on a couple of classic benchmarks from concurrency theory (Chapter 6). Note e.g., that implementations for the synthesis of asynchronous automata were not considered in the literature until now.

However, from the experimental results from the previous chapter, we conclude that due to the scalability of the core synthesis algorithms, the current method works only with rather small problems. In this light, we point a couple of possible **further** investigations:

- We could consider systems at a higher level of abstraction (e.g., communication patterns in security protocols or UML models).
- Also, more work could be invested in heuristics generating small distributable specifications (by removing behaviors, respectively collapsing actions).
- Aiming at more ‘concurrency-aware’ specifications, distributed versions of temporal logics can be considered; more research is needed in this area.





# APPENDIX A

---

## A.1 Implementability Test modulo Isomorphism for Asynchronous Automata

The logic implementation using Smodels [SNS02] of the implementability test modulo isomorphism for asynchronous automata based on Theorem 3.30 and described in Section 6.3.1 (the comments are preceded by a % symbol).

```
% Input:      a distribution (as a domain) and
%             a (possible nondeterministic) transition system TS
% Question:   is TS reachable, deadlock-free, and isomorphic
%             to an asynchronous automaton?
% Strategy:   guess a set of equivalences satisfying AA1-AA2-AA3

%===specify the specification as a database of facts. For instance:
%...the distribution
dom(a,1). dom(b,2). dom(c,1). dom(c,2).
%...the nondeterministic TS
initialstate(1). initialstate(2).
trans(1,a,3). trans(2,b,3). trans(3,c,3).

%...fix an upper bound on the number of local states
const max_local_state = 3.

%=== implementation of the begins here

%...derive the actions and processes of the distributed alphabet
action(A) :- dom(A,K).
process(K) :- dom(A,K).
%...derive the states of TS
state(Q1) :- trans(Q1,A,Q2).
state(Q2) :- trans(Q1,A,Q2).

%===reachability (least fixpoint procedure)
reach(Q) :- initialstate(Q).
reach(Q2) :- reach(Q1), trans(Q1,A,Q2), neq(Q1,Q2).
```

```

%...rule out solutions which contain unreachable states
:- state(Q), not reach(Q).

%===deadlock-freedom
live(Q1) :- trans(Q1,A,Q2).
%...rule out solutions which contain reachable deadlocks
:- state(Q), reach(Q), not live(Q).

%===choose for each state a local representative from a set of local states.
% Two states are equivalent iff they have the same local representative.
%...derive the set of local states
local_state(1..max_local_state).
%...guess exactly one k-representative lq for the each state q
%...(choice gadget)
1 {local_repr(K,Q,LQ) : local_state(LQ)} 1 :- process(K), state(Q).
%...(q1 equiv_k q2) iff they have the same local representative
equal_local(K,Q1,Q2) :- local_repr(K,Q1,LQ), local_repr(K,Q2,LQ),
    process(K), state(Q1), state(Q2), local_state(LQ).
%...equivalence of q1 and q2 on dom(a)
equiv_dom(A,Q1,Q2) :- equal_local(K,Q1,Q2):dom(A,K),
    action(A), state(Q1), state(Q2).

%===AA1: q1-a->q2 and k not in dom(a) => local_state_k(q1)=local_state_k(q2)
%...rule out solutions not satisfying AA1
:- trans(Q1,A,Q2), not dom(A,K),
    process(K), local_state(LQ1), local_state(LQ2),
    local_repr(K,Q1,LQ1), local_repr(K,Q2,LQ2), neq(LQ1,LQ2).

%===AA2: (for all k in Proc: q1 equiv_k q2) => q1 = q2
%...rule out solutions not satisfying AA2
:- neq(Q1,Q2), equal_local(K,Q1,Q2):process(K), state(Q1), state(Q2).

%===AA3: q1-a->q11 and equiv_dom(a)(q1,q2)
% => exists q22 s.t. q2-a->q22 and equiv_dom(q11,q22)
matched(A,Q1,Q11,Q2) :- trans(Q1,A,Q11), trans(Q2,A,Q22),
    equiv_dom(A,Q1,Q2), equiv_dom(A,Q11,Q22).
%...rule out solutions not satisfying AA3
:- trans(Q1,A,Q11), state(Q2), equiv_dom(A,Q1,Q2),
    not matched(A,Q1,Q11,Q2).

```

# BIBLIOGRAPHY

---

- [AE98] P.C. Attie and E.A. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):51–115, 1998. (Cited on pages 153, 159 and 197.)
- [AE01] P.C. Attie and E.A. Emerson. Synthesis of concurrent systems for an atomic read/write model of computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(2):187–242, 2001. (Cited on pages 153, 158, 163 and 197.)
- [AEY01] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In *ICALP 2001*, volume 2076 of *LNCS*, pages 797–808. Springer, 2001. (Cited on page 127.)
- [AKH03] J.H. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16(2–3):75–110, 2003. (Cited on page 156.)
- [AM94] A. Anuchitanukul and Zohar Manna. Realizability and synthesis of reactive modules. In *CAV'94*, volume 818 of *LNCS*, pages 156–168. Springer, 1994. (Cited on page 159.)
- [Arn94] A. Arnold. *Finite transition systems and semantics of communicating systems*. Prentice Hall, 1994. (Cited on pages 3, 6, 25, 32 and 72.)
- [AS92] C. Autant and Ph. Schnoebelen. Place bisimulation in Petri nets. In *ICATPN'92*, volume 616 of *LNCS*, pages 45–61. Springer, 1992. (Cited on page 187.)
- [AS02] B. Adsul and M. Sohoni. Complete and tractable local linear time temporal logics over traces. In *ICALP'02*, volume 2380 of *LNCS*, pages 926–937. Springer, 2002. (Cited on pages 1 and 159.)
- [Att99] P.C. Attie. Synthesis of large concurrent programs via pairwise composition. In *CONCUR'99*, volume 1664 of *LNCS*, pages 130–145. Springer, 1999. (Cited on pages 159 and 197.)
- [BBD95] E. Badouel, L. Bernardinello, and P. Darondeau. Polynomial algorithms for the synthesis of bounded nets. In *TAPSOFT'95*, volume 915 of *LNCS*, pages 364–378. Springer, 1995. (Cited on pages 72 and 127.)

- [BBD97] E. Badouel, L. Bernardinello, and P. Darondeau. The synthesis problem for elementary net systems is NP-complete. *TCS*, 186(1–2):107–134, 1997. (Cited on pages 72 and 127.)
- [BBL05] J. Berstel, L. Boasson, and M. Latteux. Mixed languages. *TCS*, 332(1–3):179–198, 2005. (Cited on page 111.)
- [BCD02] E. Badouel, B. Caillaud, and P. Darondeau. Distributing finite automata through Petri net synthesis. *Formal Aspects of Computing*, 13(6):447–470, 2002. (Cited on pages 3, 72, 159, 164, 167, 179 and 197.)
- [BCE<sup>+</sup>03] A. Benveniste, P. Caspi, S.E. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003. (Cited on page 55.)
- [BD98] E. Badouel and P. Darondeau. Theory of regions. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:529–586, 1998. (Cited on page 3.)
- [BDT03] Y. Bar-David and G. Taubenfeld. Automatic discovery of mutual exclusion algorithms. In *DISC’03*, volume 2848 of *LNCS*, pages 136–150. Springer, 2003. (Cited on pages 187 and 197.)
- [BM03] N. Baudru and R. Morin. Safe implementability of regular message sequence chart specifications. In *Proc. of SNPD’03*, pages 210–217. ACIS, 2003. (Cited on pages 52 and 127.)
- [BM06] N. Baudru and R. Morin. Unfolding synthesis of asynchronous automata. In *Proc. of International Computer Science Symposium in Russia (CSR’06)*, LNCS. Springer Verlag, 2006. To appear. (Cited on page 138.)
- [BMT01] A. Bouajjani, A. Muscholl, and T. Touili. Permutation rewriting and algorithmic verification. In *LICS’01*, pages 399–408. IEEE Computer Society, 2001. (Cited on page 185.)
- [BPS94] D. Bruschi, G. Pighizzini, and N. Sabadini. On the existence of minimum asynchronous automata and on the equivalence problem for unambiguous regular trace languages. *Information and Computation*, 108:262–285, 1994. (Cited on pages 151, 152 and 187.)
- [BPS01] J. Bergstra, A. Ponse, and S. Smolka, editors. *Handbook of process algebra*. Elsevier, 2001. (Cited on page 69.)
- [Cai97] B. Caillaud. SYNET: un outil de synthèse de réseaux de Petri bornés, applications. Technical Report 3155, INRIA, 1997. (Cited on pages 179, 182 and 197.)
- [CE82] E.M. Clarke and E.A. Emerson. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982. (Cited on pages vi, 1, 2, 153, 154, 155, 157, 158, 159 and 197.)



- [CKK<sup>+</sup>97] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. PETRIFY: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Trans. Information and Systems*, E80-D(3):315–325, 1997. (Cited on page 179.)
- [CKK<sup>+</sup>00] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Hardware and Petri nets: Application to asynchronous circuit design. In *ICATPN'00*, volume 1825 of *LNCS*, pages 1–15. Springer, 2000. (Cited on page 198.)
- [CKK<sup>+</sup>02] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic synthesis of asynchronous controllers and interfaces*. Springer, 2002. (Cited on pages 3 and 198.)
- [CKLY98] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving Petri nets from finite transition systems. *IEEE Transactions on Computers*, 47(8):859–882, 1998. (Cited on pages 3 and 72.)
- [CMT99] I. Castellani, M. Mukund, and P.S. Thiagarajan. Synthesizing distributed transition systems from global specifications. In *FSTTCS19*, volume 1739 of *LNCS*, pages 219–231. Springer, 1999. (Cited on pages 3, 44, 45, 47, 55, 56, 57, 72, 73, 74, 82, 92, 107, 115, 130, 131, 152, 159, 173 and 197.)
- [CMZ93] R. Cori, Y. Métivier, and W. Zielonka. Asynchronous mappings and asynchronous cellular automata. *Information and Computation*, 106(2):159–202, 1993. (Cited on page 138.)
- [CSLR88] R. Cori, E. Sopena, M. Latteux, and Y. Roos. 2-asynchronous automata. *TCS*, 61:93–102, 1988. (Cited on pages 151 and 152.)
- [Dij65] E.W. Dijkstra. Solutions of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965. (Cited on page 156.)
- [DM96] V. Diekert and A. Muscholl. A note on Métivier's construction of asynchronous automata for triangulated graphs. *Fundamenta Informaticae*, 25:241–246, 1996. (Cited on pages 138, 151 and 152.)
- [DR95] V. Diekert and G. Rozenberg, editors. *The book of traces*. World Scientific, 1995. (Cited on pages 3, 26, 30, 52, 134, 141, 152, 185 and 197.)
- [Dub86] C. Duboc. Mixed product and asynchronous automata. *TCS*, 48:183–199, 1986. (Cited on pages 35, 55, 57, 62, 138 and 152.)
- [ER90] A. Ehrenfeucht and G. Rozenberg. Partial (set) 2-structures I and II. *Acta Informatica*, 27(4):315–368, 1990. (Cited on pages 2, 49 and 72.)
- [ER99] J. Esparza and S. Römer. An unfolding algorithm for synchronous products of transition systems. In *CONCUR'99*, volume 1664 of *LNCS*, pages 2–20. Springer, 1999. (Cited on page 174.)

- [GJ79] M. Garey and D. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. Freeman, 1979. (Cited on pages 18, 71, 106, 107, 109, 112 and 113.)
- [GLM04a] E. Giunchiglia, Y. Lierler, and M. Maratea. SAT-based answer set programming. In *AAAI'04*, pages 61–66. AAAI Press, 2004. (Cited on page 187.)
- [GLM04b] E. Giunchiglia, Y. Lierler, and M. Maratea. A SAT-based polynomial space algorithm for answer set programming. In *NMR'04*, pages 189–196, 2004. (Cited on page 187.)
- [GM02] P. Gastin and M. Mukund. An elementary expressively complete temporal logic for Mazurkiewicz traces. In *ICALP'02*, volume 2380 of *LNCS*, pages 938–949. Springer, 2002. (Cited on pages 1 and 159.)
- [GM06] B. Genest and A. Muscholl. Constructing exponential-size deterministic Zielonka automata, 2006. Manuscript. (Cited on page 152.)
- [Hel99] K. Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. *Fundamenta Informatica*, 37(3):247–268, 1999. (Cited on pages 174 and 177.)
- [HKV02] D. Harel, O. Kupferman, and M.Y. Vardi. On the complexity of verifying concurrent transition systems. *Information and Computation*, 173(2):143–161, 2002. (Cited on page 127.)
- [HMU01] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to automata theory, languages, and computation*. Addison Wesley, second edition edition, 2001. (Cited on pages 71, 107, 112 and 113.)
- [Hol04] G. Holzmann. *The Spin model checker*. Addison-Wesley, 2004. (Cited on page 174.)
- [HŞ04] K. Heljanko and A. Ştefănescu. Complexity results for checking distributed implementability. Technical Report 05/2004, Universität Stuttgart, 2004. 37 pp. (Cited on pages 8, 71, 155 and 179.)
- [HŞ05] K. Heljanko and A. Ştefănescu. Complexity results for checking distributed implementability. In *Proceedings of the 5th International Conference on Application of Concurrency to System Design*, pages 78–87. IEEE Computer Society, 2005. (Cited on pages 8 and 71.)
- [HU79] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation*. Addison Wesley, 1979. (Cited on pages 15, 16, 22, 25, 101, 108, 113, 114 and 117.)
- [Ina84] Y. Inaba. An implementation of synthesizing synchronization skeletons using temporal logic specifications. Master's thesis, Department of Computer Science, The University of Texas at Austin, 1984. Completed under the supervision of E.A. Emerson. (Cited on page 197.)

- [Jan03] T. Janhunen. A counter-based approach to translating logic programs into set of clauses. In *Proceedings of the 2nd International Workshop on Answer Set Programming (ASP'03)*, volume 78, pages 166–180. Sun SITE Central Europe (CEUR), 2003. (Cited on page 180.)
- [KMS94] N. Klarlund, M. Mukund, and M. Sohoni. Determinizing asynchronous automata. In *ICALP'94*, volume 820 of *LNCS*, pages 130–141. Springer, 1994. (Cited on pages 65, 138, 148 and 152.)
- [KV01] O. Kupferman and M.Y. Vardi. Synthesizing distributed systems. In *LICS'01*. IEEE Computer Society, 2001. (Cited on pages 2, 153, 154 and 159.)
- [Lal79] G. Lallement. *Semigroups and combinatorial applications*. J. Wiley and Sons, 1979. (Cited on page 16.)
- [Loh03] M. Lohrey. Realizability of high-level message sequence charts: closing the gaps. *Theoretical Computer Science*, 309(1-3):529–554, 2003. (Cited on page 127.)
- [Maz77] A. Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Report PB-78, Aarhus University, 1977. (Cited on pages 1, 3 and 25.)
- [Maz87] A. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, number 255 in *LNCS*, pages 279–324. Springer, 1987. (Cited on pages 26 and 52.)
- [Mét87] Y. Métivier. An algorithm for computing asynchronous automata in the case of acyclic non-commutation graph. In *ICALP'87*, volume 267 of *LNCS*, pages 226–236. Springer, 1987. (Cited on pages 138 and 151.)
- [Mil89] R. Milner. *Communication and concurrency*. Prentice-Hall, 1989. (Cited on page 20.)
- [MMP<sup>+</sup>95] O. Matz, A. Miller, A. Potthoff, W. Thomas, and E. Valkema. Report on the program AMoRE. Technical Report 9507, Institut für Informatik und Praktische Mathematik, CAU Kiel, 1995. <http://www-i7.informatik.rwth-aachen.de/d/research/amore.html>. (Cited on pages 171 and 173.)
- [Mor98] R. Morin. Decompositions of asynchronous systems. In *CONCUR'98*, volume 1466 of *LNCS*, pages 549–564. Springer, 1998. (Cited on pages 3, 42, 44, 47, 49, 72, 73, 83, 92, 93, 111 and 152.)
- [Mor99a] R. Morin. *Catégories de modèles du parallélisme*. PhD thesis, Université Paris XI, 1999. (Cited on pages 92 and 93.)
- [Mor99b] R. Morin. Hierarchy of asynchronous automata. In *WDS'99*, volume 28 of *Electronic Notes in Theoretical Computer Science*, pages 59–75, 1999. (Cited on pages 44, 47, 72, 73, 84, 92 and 93.)

- [MP96] A. Muscholl and H. Petersen. A note on the commutative closure of star-free languages. *Information Processing Letters*, 57(2):71–74, 1996. (Cited on page 119.)
- [MS72] A.R. Meyer and L. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *3th IEEE Symposium on Switching and Automata Theory*, pages 125–129, 1972. (Cited on page 108.)
- [MS94] M. Mukund and M. Sohoni. Gossiping, asynchronous automata and Zielonka’s theorem. Report TCS-94-2, School of Mathematics, SPIC Science Foundation, Madras, India, 1994. (Cited on pages 134, 138 and 152.)
- [MS03] F. Moller and S. Smolka. On the computational complexity of bisimulation, redux. In *PCK50*, pages 55–59. ACM, 2003. (Cited on pages 116 and 117.)
- [MT02] P. Madhusudan and P.S. Thiagarajan. A decidable class of asynchronous distributed controllers. In *CONCUR’02*, volume 2421 of *LNCS*, pages 145–160. Springer, 2002. (Cited on pages 2, 154 and 197.)
- [Muk02] M. Mukund. From global specifications to distributed implementations. In B. Caillaud, P. Darondeau, and L. Lavagno, editors, *Synthesis and control of discrete event systems*, pages 19–34. Kluwer, 2002. (Cited on pages 3, 35, 44, 45, 47, 52, 55, 65, 72, 73, 74, 84, 92, 100, 107, 115, 116, 131 and 152.)
- [Mus94] A. Muscholl. *Über die Erkennbarkeit unendlicher Spuren*. PhD thesis, Universität Stuttgart, 1994. Published by Teubner, 1996. (Cited on pages 65 and 107.)
- [Mus96] A. Muscholl. On the complementation of Büchi asynchronous cellular automata. *TCS*, 169:123–145, 1996. (Cited on page 138.)
- [MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(1):68–93, 1984. (Cited on pages 1, 2, 153, 154, 155, 157 and 159.)
- [MW03] S. Mohalik and I. Walukiewicz. Distributed games. In *FSTTCS 2003*, volume 2914 of *LNCS*, pages 338–351. Springer, 2003. (Cited on page 2.)
- [NRT92] M. Nielsen, G. Rozenberg, and P.S. Thiagarajan. Elementary transition systems. *TCS*, 96:3–33, 1992. (Cited on pages 3 and 72.)
- [NT02] M. Nielsen and P.S. Thiagarajan. Regular event structures and finite Petri nets: The conflict-free case. In *ICATPN’02*, volume 2360 of *LNCS*, pages 335–351. Springer, 2002. (Cited on page 146.)
- [Pap94] Ch. H. Papadimitriou. *Computational complexity*. Addison Wesley, 1994. (Cited on pages 71, 107, 113 and 119.)

- [PEP] The PEP tool. Online at <http://theoretica.informatik.uni-oldenburg.de/~pep>. (Cited on page 174.)
- [Pet62] C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn, Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. (Cited on pages 2 and 25.)
- [Pig93a] G. Pighizzini. *Recognizable trace languages and asynchronous automata*. PhD thesis, Università degli Studi di Milano, 1993. (Cited on pages v, 17, 51, 134, 138, 152 and 187.)
- [Pig93b] G. Pighizzini. Synthesis of nondeterministic asynchronous automata. In *Semantics of Programming Languages and Model Theory, Algebra, Logic and Applications*, volume 5, pages 109–126. Gordon and Breach Science Publ., 1993. (Cited on pages 138 and 152.)
- [Pig94] G. Pighizzini. Asynchronous automata and asynchronous cellular automata. *TCS*, 132:179–207, 1994. (Cited on page 187.)
- [PR89] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *ICALP'89*, volume 372 of *LNCS*, pages 652–671. Springer, 1989. (Cited on pages 2, 153, 154 and 159.)
- [PWW98] D. Peled, T. Wilke, and P. Wolper. An algorithmic approach for checking closure properties of temporal logic specifications and  $\omega$ -regular languages. *TCS*, 195(2):183–203, 1998. (Cited on pages 106, 107 and 109.)
- [Rab97] A. Rabinovich. Complexity of equivalence problems for concurrent systems of finite agents. *Information and Computation*, 139(2):111–129, 1997. (Cited on page 127.)
- [Ray86] M. Raynal. *Algorithms for mutual exclusion*. North Oxford Academic, 1986. (Cited on page 156.)
- [Roh04] K.R. Rohloff. *Computations on distributed discrete-event systems*. PhD thesis, University of Michigan, 2004. (Cited on page 127.)
- [Sai02] H. Saïdi. Towards automatic synthesis of security protocols. In *In Logic-Based Program Synthesis Workshop, AAAI 2002 Spring Symposium*, pages 45–52. Stanford University, California, 2002. (Cited on page 198.)
- [Sak92] J. Sakarovitch. The “last” decision problem for rational trace languages. In *LATIN 1992*, volume 583 of *LNCS*, pages 460–473. Springer, 1992. (Cited on page 119.)
- [Sch00] K. Schmidt. LoLA – A Low Level Analyser. In *ICATPN'00*, volume 1825 of *LNCS*, pages 465–474. Springer, 2000. (Cited on page 174.)

- [SEM03] A. Ștefănescu, J. Esparza, and A. Muscholl. Synthesis of distributed algorithms using asynchronous automata. In R. Amadio and D. Lugiez, editors, *CONCUR'03*, volume 2761 of *LNCS*, pages 27–41. Springer, 2003. (Cited on pages 8, 52, 65, 71, 74, 129, 152, 155 and 179.)
- [SHRS96] S.K. Shukla, H.B. Hunt III, D.J. Rosenkrantz, and R.E. Stearns. On the complexity of relational problems for finite state processes. In *ICALP'96*, volume 1099 of *LNCS*, pages 466–477. Springer, 1996. (Cited on pages 104, 105, 116 and 127.)
- [SNS02] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002. (Cited on pages 128, 179 and 201.)
- [SS00] Ph. Schnoebelen and N. Sidorova. Bisimulation and the reduction of Petri nets. In *ICATPN'00*, volume 1825, pages 409–423. Springer, 2000. (Cited on page 187.)
- [SSE03] C. Schröter, S. Schwoon, and J. Esparza. The Model-Checking Kit. In *ICATPN'03*, volume 2679 of *LNCS*, pages 463–472. Springer, 2003. <http://www.informatik.uni-stuttgart.de/fmi/szs/tools/mckit/>. (Cited on page 176.)
- [Ște02] A. Ștefănescu. Automatic synthesis of distributed systems. In *Proc. of 17th IEEE International Conference on Automated Software Engineering*, page 315. IEEE Computer Society, 2002. Position paper. Long version appearing in the Proc. of ASE'02 Doctoral Symposium, pages 1–6. (Cited on page 8.)
- [Tab06] P. Tabuada. Distributing specifications and distributed supervisory control, 2006. Manuscript. (Cited on pages 127, 131 and 137.)
- [TH98] P.S. Thiagarajan and J.G. Henriksen. Distributed versions of linear time temporal logic: A trace perspective. In *Petri Nets 1996*, volume 1491 of *LNCS*, pages 643–681. Springer, 1998. (Cited on pages 1, 35 and 159.)
- [Thi95] P.S. Thiagarajan. A trace consistent subset of PTL. In *Concur'95*, volume 962 of *LNCS*, pages 438–452. Springer, 1995. (Cited on pages 55, 56 and 57.)
- [vG90] R.J. van Glabbeek. The linear time – branching time spectrum (extended abstract). In *CONCUR'90*, volume 458 of *LNCS*, pages 278–297. Springer, 1990. (Cited on page 115.)
- [VHL97] K. Varpaaniemi, K. Heljanko, and J. Lilius. PROD 3.2 – an advanced tool for efficient reachability analysis. In *CAV'97*, volume 1254 of *LNCS*, pages 472–475. Springer, 1997. (Cited on page 174.)



- [VK02] A. Valmari and A. Kervinen. Alphabet-based synchronisation is exponentially cheaper. In *CONCUR'02*, volume 2421 of *LNCS*, pages 161–176. Springer, 2002. (Cited on page 127.)
- [Vog99] W. Vogler. Concurrent implementation of asynchronous transition systems. In *ICAPTN'99*, volume 1639 of *LNCS*, pages 284–303. Springer, 1999. (Cited on page 72.)
- [Wal98] I. Walukiewicz. Difficult configurations – on the complexity of LTrL. In *ICALP'98*, volume 1443 of *LNCS*, pages 140–151. Springer, 1998. (Cited on pages 1 and 159.)
- [Wal02] I. Walukiewicz. Local logics for traces. *Journal of Automata, Languages and Combinatorics*, 7:259–290, 2002. (Cited on pages 1 and 159.)
- [Yak98] A. Yakovlev. Designing control logic for Counterflow Pipeline Processor using Petri nets. *Formal Methods in System Design*, 12(1):39–71, 1998. (Cited on pages 182 and 198.)
- [Zie87] W. Zielonka. Notes on finite asynchronous automata. *R.A.I.R.O. Inform. Théor. Appl.*, 21:99–135, 1987. (Cited on pages v, 1, 3, 25, 30, 31, 32, 35, 37, 40, 55, 57, 62, 63, 67, 69, 72, 73, 111, 127, 132, 133, 134, 138, 152 and 153.)
- [Zie89] W. Zielonka. Safe executions of recognizable trace languages by asynchronous automata. In *Logical Foundations of Computer Science*, volume 363 of *LNCS*, pages 278–289. Springer, 1989. (Cited on pages 51, 63, 64, 73, 132 and 138.)





# INDEX

---

- $L(\mathcal{A}, F)$ , *see* language of an automaton
- $L(TS)$ , *see* language of a transition system
- $[L]$ , *see* trace-closure of a language
- $[w]$ , *see* trace
- $\#_a$ , *see* number of occurrences
- $M(\Sigma, \parallel)$ , *see* trace monoid
- $\mathbb{N}$ , *see* natural numbers
- $\text{Prefix}(-)$ , *see* prefix-closure
- $\text{PrefReg}(\Sigma)$ , *see* prefix-closed regular languages
- $\text{Reg}(\Sigma)$ , *see* regular languages
- $\text{Reg}(\Sigma, \parallel)$ , *see* regular trace languages
- $\Sigma(-)$ , *see* alphabet of
- $| - |$ , *see* cardinality
- $\mathcal{C}$ , *see* complement
- $\text{dom}(a)$ , *see* domain of an action
- $\text{lin}(T)$ , *see* linearization
- $\Sigma_{\text{loc}}(p)$ , *see* local alphabet
- $\mathcal{P}(-)$ , *see* power set
- $\prod_{i \in I}$ , *see* cartesian product
- $\subsetneq$ , *see* proper inclusion
- $\downarrow_S$ , *see* projection of
- $\varepsilon$ , *see* empty word
- BD, *see* backward diamond
- FD, *see* forward diamond
- ID, *see* independent diamond
- accepting state, *see* final state
- action, 11
- acyclic
  - asynchronous automaton, 38, 44, 50, 65
  - finite automata, 23
  - graph, 17
  - synchronous product, 35, 43, 50, 58
  - transition system, 19, 23
- alphabet, 11, 18
  - concurrent, *see* concurrent alphabet
  - of a language, 13, 20
  - of a transition system, 19, 118, 122
  - of a word, 12
- antisymmetry, 11
- arc, 16
- asynchronous automaton, 32, 37
- automaton, *see* finite automaton
- backward diamond, 139
- behavior, *see* language
- bisimulation, 20, 69, 72, 114
- cardinality, 9
- cartesian product, 10, 27, 35, 51, 58, 59
- clique, 5, 17
  - cover, 17
- closed distributed system, 31
- co-reachable, 19, 21
- complement, 10, 13, 16, 28
- concatenation, 11, 13, 14, 16, 28
- concurrency, 25
- concurrency relation, *see* independence relation
- concurrent alphabet, 26, 41, 92, 139
- congruence relation, 11, 27
- connected
  - component, 17
  - graph, 17
  - nodes, 17
- counterexample, 174
- cup of coffee, 31
- cycle, 17
- deadlock-free, 126
- dependence
  - graph, 5, 26
  - relation, 5, 26

- deterministic
  - asynchronous automaton, 38
  - asynchronous automaton, 43, 50, 132
  - finite automaton, 21
  - synchronous product, 35, 43, 50, 57
  - transition system, 19, 21, 47, 49, 92
- diamond rules, *see* ID and FD
- disjoint union, 58, 60
- distributed alphabet, *see* distribution
- distributed implementability, 69, 71
- distributed system, 31
- distribution, 5, 32, 34
- domain of an action, 33
- domain of an action, 34, 35, 38, 45
- edge, 16
- empty word, 12
- equivalence
  - class, 10
  - relation, 10, 27, 44
- event, 133
- execution, 11
- expressiveness, 65
- final states, 20, 21, 37, 40, 51, 54, 59, 65
- finite automaton, 20, 21
- finite index, *see* index of a congruence
- finite language, 23, 58, 65
- forward diamond, 41
- forward-closed language, 52
- free partially commutative monoid, 27
- game, 2
- graph, 16
  - complete, 17
  - directed, 16
  - isomorphism, 17
  - undirected, 16
- identity element, 11
- independence
  - graph, 26
  - relation, 3, 5, 26, 34, 41, 105
- independent diamond, 41, 122
- index of a congruence, 11, 16, 31, 134
- initial state, 18, 20
- initial states, 51, 57, 73
- interleaving, 25
- intersection, 10, 13, 14, 16, 28
- isomorphic embedding, 122
- isomorphism, *see* transition system isomorphism
- iteration, 13, 14, 16, 28
- Kleene-\*, *see* iteration
- Kripke structure, 157
- label, 18
- language, 12, 51
  - of a synchronous product, 37, 55
  - of a transition system, 19
  - of an asynchronous automaton, 40, 63
  - of an automaton, 21
- language equivalence, 20, 69, 72, 99, 130, 132
- linearization, 27, 31
- local
  - alphabet, 33–35
  - final states, 59
  - initial states, 59
  - state, 35
  - transition system, 34
- local alphabet, 5
- minimal
  - deterministic finite automaton, 22, 55
  - deterministic transition system, 117
  - deterministic transition system, 22, 53
- monoid, 11
- morphism, 11
  - canonical, 11
- natural numbers, 10
- node, 16
- nondeterminism, 51
- number of occurrences, 12
- occurrence, 133
- open distributed system, 31
- partial order, 11, 133
- partially order set, *see* poset
- path, 17
- Petri net, 2, 32

- poset, 11, 27
- power set, 10
- prefix, 12
- prefix of a trace, 134
- prefix-closed
  - language, 13, 30, 57
  - regular language, 16, 21
- prefix-closure, 13, 14, 16, 28, 107
- process, 5
- Producer-Consumer problem, 3
- product language, 56, 99, 130
- projection
  - of a language, 13, 14, 16, 28
  - of a word, 12
- proper inclusion, 10, 65
- quotient, 11, 19, 47
- reachable, 19, 100
- recognizable trace languages, 30
- reflexivity, 10, 11
- regular
  - expression, 15, 107, 112
  - language, 15, 21, 30, 31, 57
  - trace language, 30, 31
- run, 19
- safe asynchronous automaton, 51, 138
- safe Petri net, 187
- sequential system, 31
- shuffle product
  - of languages, 6, 13, 14, 16, 28
  - of words, 12
- singleton, 10
- sink state, 22, 54
- spanning tree, 17, 189
- state, 18
- state space, 18, 35
- state space explosion problem, 37
- subgraph, 17
  - induced, 17
- suffix, 12
- suffix of a trace, 134
- symmetry, 10
- synchronization on common actions, 32, 34, 37, 55
- synchronous product, *see* synchronous product of transition systems
- synchronous product of transition systems, 32, 34
- syntactic congruence
  - of a language, 16, 134, 135
  - of a trace language, 31
- synthesis, 2, 4, 7, 47, 68, 129
- theory of regions, 2, 49, 178
- trace, 3, 26, 27, 30, 31, 67
  - equivalence, 27, 41
  - language, 3, 27, 28
  - monoid, 27
- trace-closed language, 28, 31, 52
- trace-closure of a language, 30, 118
- transition relation, 18
- transition system, 18, 31
  - isomorphism, 19, 20, 44, 47, 72, 74, 129, 131
- transitive
  - closure, 10
  - graph, 18
- transitivity, 10, 11
- tree, 17, 189
- union, 10, 13, 14, 16, 28
- vertex, 16
- word, 11

This is the last page of this thesis.  
Congratulations for reaching the end!