From Supervisory Control to Nonblocking Controllers for Discrete Event Systems

Vom Fachbereich Informatik
der Universität Kaiserslautern

zur Verleihung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)
genehmigte Dissertation

von Petra Malik

Dekan:	Prof. Dr. Hans Hagen
Berichterstatter:	Prof. Dr. Otto Mayer
	Prof. Dr. Wolfram Büttner

Tag der wissenschaftlichen Aussprache: 21. November 2003

D386

Kaiserslautern, im November 2003

Abstract

With supervisory control theory it is possible to describe controllers that influence the behaviour of a discrete event system by disabling certain events. This thesis considers the design of controllers that send commands to actuators and receive information from sensors. If such a controller is generated from a supervisory control model, the property to be nonblocking may be lost. This thesis characterises a class of systems for which each derived controller is necessarily nonblocking. Efficient algorithms for checking if a model is contained in this class are presented. The introduced approach is applied to the design and verification of an industrial example.

Contents

List of Figures

Chapter 1

Introduction

A discrete event system (DES) is a dynamic system that is discrete in time, discrete in state space, and event-driven. Application domains include manufacturing systems, traffic systems, car electronics, and communication protocols. It is a difficult and error-prone task to develop controllers for such systems.

The nonblocking property, which states that a system can never run into a deadlock or livelock, is crucial for dynamic systems. If a system blocks, i.e. if it contains a deadlock or a livelock, a state can be reached from which the current task cannot be completed; there is no more progress. For example, a robot stops working, a traffic light never changes to green, or two communication partners wait for each other to send some message. In all these cases, the system is blocked, and the desired functionality cannot be achieved. Such a blocking condition may in some cases be easily fixed by restarting the system, but it may also have a serious impact such as the loss of many hours of productive work.

The theory of supervisory control of discrete event systems, initiated by Ramadge and Wonham [RW87b], provides a powerful abstract framework for the control of discrete event systems. The environment (also called plant) is modelled as a generator of a formal language over an alphabet of events. The control feature is represented by the fact that certain events can be disabled by a so-called supervisor. The general control problem is to find a supervisor such that the composed behaviour of environment and supervisor meets some specification. Supervisory control theory states that, given a plant model and a specification, it is possible to synthesise the least restrictive supervisor such that the controlled system satisfies the specification and is nonblocking. In the case of regular languages, i.e. finite-state systems, this supervisor can be effectively computed.

Many application domains require a special kind of supervisor, which we call controller. Controllers receive their information from the plant by sensors and send commands to actuators. For instance, in a dosing tank, information about the system is received by level sensors. The controller sends, depending on the filling level, open and close commands to valves. Supervisory control theory provides the means to compute a model of the possible system behaviour, which defines a set of admissible events for each state. A controller implementing such a model needs a criterion to choose the command to be given next from this event set. The result of the supervisor synthesis process, however, does not include such a criterion. While it is possible to think of controllers that choose the commands at runtime, many systems require a unique choice to be made at design time. The controller has to be designed so that there is at most one command to send at each state.

At the same time, bad choices can take the system to a livelock because, by always sending the "wrong" commands, the controller may be able to keep changing the state of the system cyclically, thereby preventing it from completing any task. Obviously, this fact has a serious impact on the applicability of supervisory control models, since even if a system has been proven to be nonblocking, its actual controller realisation may indeed be blocking and thus fail to fulfil its required functionality. Additional effort is required to derive a controller that does not have the possibility of running into a livelock.

This thesis describes a novel approach for the design of controllers. A new class of discrete event systems is characterised by introducing three properties. These models are nonblocking for each controller realisation. A controller generated from such models is necessarily free from deadlocks and livelocks. Usually, the model has to be refined to obtain a system that satisfies the three properties. This refinement process can be supported by counter-examples, which are generated whenever one of the properties is violated. They point to problems in the design, sometimes revealing subtle errors, and tell the designer what has to be refined further. If a model finally satisfies the three properties, then any possible controller for this model is nonblocking.

A detailed description of the problem solved in this thesis and references to related work are given in chapter 2. The underlying notations are introduced in chapter 3. In chapter 4, the problem is formalised and the three properties are introduced. The main result states that discrete event systems satisfying the three properties are nonblocking for each derived controller.

In chapter 5, algorithms for checking the introduced properties are proposed. First, state exploration algorithms are introduced, which need to construct all reachable states of a system. It is well known that the number of states of a system can grow exponentially in the number of constituent subsystems—this fact is also known as the state explosion problem. Therefore, state exploration algorithms are limited to small examples. In order to overcome this problem, incremental algorithms are introduced. Incremental algorithms try to conclude properties of a large system by inspecting only parts of the entire system. They are, in many cases, able to handle models of realistic size.

The proposed ideas are illustrated using two examples in chapter 6. The problems and their solutions are first presented using a batch plant example, which is often used in literature. This example shows that the suggested approach can be widely used within its application domains. Next, a more elaborate industrial example is considered, which is taken from car electronics. The whole modelling process of some parts of a window control system is performed, leading to a model which is nonblocking for each derived controller. This example shows that the approach proposed in this thesis is also applicable and useful for quite large real-world systems.

In chapter 7, we summarise the main contributions of this thesis and outline directions for future research.

Chapter 2

Background

As discrete event systems and their control problems get increasingly complex, the use of formal methods in analysis and design of controllers attracts more and more attention. *Supervisory control theory* [RW89] aims at solving a couple of control synthesis problems for discrete event systems. Based on such well-established concepts as automata and formal languages, supervisory control permits the formulation and solution of a variety of problems, including decentralised and hierarchical control as well as control based on partial observation. It gives a better understanding of the fundamental concepts concerning analysis and control of discrete event systems.

The vision of this thesis is the following: to automatically derive a controller from a supervisory control model. More precisely, we are interested in code for a controller that is automatically generated from a supervisory control model. This simplifies the process of designing a controller and avoids man-made errors during the coding phase. Since the model satisfies the specification, the generated controller also satisfies the specification as long as the code generation procedure is correct. Changes of the controller logic can be validated for the model, and the corrected controller is synthesised automatically.

Several problems have to be solved to achieve this goal. In this thesis, a solution to the command selection problem is proposed, which arises from the fact that a model usually allows a larger behaviour than an actual controller will exploit. In section 2.1, problems related to the generation of controllers are illustrated by means of some simple examples. In section 2.2, existing solutions to these problems are discussed and a short overview on supervisory control theory and related research activities is given.

2.1 Problem Description

In this thesis, the design of controllers for discrete event systems based on supervisory control is studied. We think of a *controller* as a piece of software accepting input from so-called *sensors* and generating output to so-called *actuators*.

The Ramadge-Wonham approach starts with the abstract notion of *supervisors*. A supervisor disables certain events in order to control a system. However, the cause of the occurrence of an event is not modelled, enabling the design of a wide variety of systems. Nevertheless, a designer of controllers faces problems concerning the generation of events and communication delays, which may occur when data are exchanged.

Consider a small example[1] taken from manufacturing to illustrate these problems. The system to be controlled consists of two machines, named A and B, that can be started and, after a while, finish their work. We want to design a controller that starts the machines and ensures that only one machine is working at a time. Two switches for starting each machine serve as actuators and are connected to the output of the controller. The input is provided by sensors indicating that a machine has finished its work.

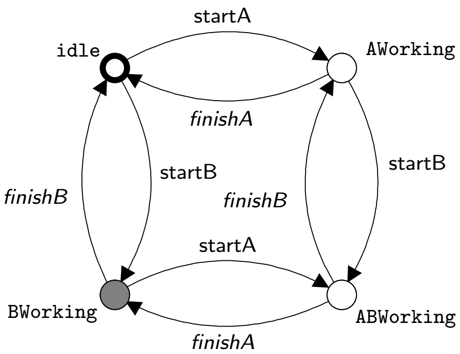

Figure 2.1: Open-loop behaviour.

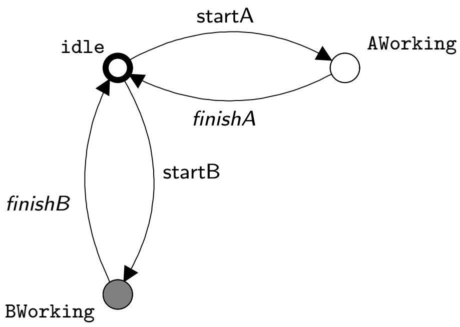

Figure 2.2: Closed-loop behaviour.

An automaton model of the uncontrolled system is given in figure 2.1. The events startA and startB model the starting of the machines. These are *controllable* events: a supervisor may disable them. The events *finishA* and *finishB* model the completion of the work. These are *uncontrollable* events: they cannot be influenced by a supervisor. We distinguish four states: state `idle` in which no machine is working, state `AWorking` in which machine A is working, state `BWorking` in which machine B is working, and state `ABWorking` in which both machines are working. The initial state `idle` is bordered black. Shaded states are called marked and are used to characterise the nonblocking property. We discuss this property later.

Supervisory control theory provides algorithms for synthesising the least restrictive and nonblocking supervisor which ensures that only one machine is working at a time. The system controlled by such a supervisor is given in figure 2.2. The undesired state `ABWorking` cannot be reached any more and is therefore omitted.

[1]More elaborate examples are discussed in chapter 6.

If the physical system generates uncontrollable and controllable events on its own or is driven by an agent, for instance, manually by a human, then the supervisor is easily implementable. The task of the implemented supervisor is to disable certain controllable events. But it is often the case that the plant does not generate all events on its own without being initiated. Normally, simple machines do not start their work unless the start command (for instance by pushing the start button) is given. In this case, it is desirable to have a controller which not only disables controllable events but also initiates the occurrence of particular events by sending commands to actuators.

The question now arises: how do we obtain such a controller from a given model? For the example above, the desired controller is a realisation of the automaton model in figure 2.2. At state `idle`, the controller produces either startA or startB as output and changes its state respectively. At states `AWorking` and `BWorking`, the controller waits for input and changes to state `idle` when the expected input arrives.

At state `idle`, a choice has to be made which machine should be started next. This decision can be performed by a controller non-deterministically. However, such behaviour is not desired in many practical applications. Most people do not want to have controllers that role dices in order to decide how to proceed further. Thus, we are interested in software for which each output is uniquely defined by the sequence of events occurred so far. The controller which always starts machine A, and thus restricts the behaviour of the controlled system as shown in figure 2.3, satisfies this requirement and its behaviour is still contained in the desired behaviour.

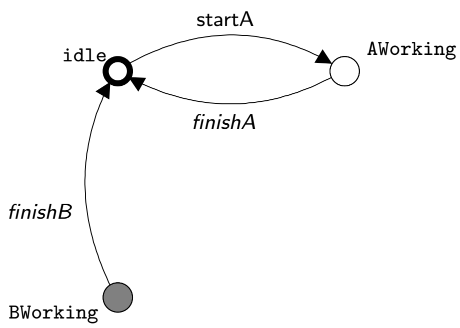

Figure 2.3: Further restricted closed-loop behaviour.

To be nonblocking is, as mentioned in the introduction, a crucial property for discrete event systems. In supervisory control theory, the marked states are used to describe completed tasks; they represent a set of states which we always want to be reachable by any behaviour. If the system ever enters a state from which no marked state can be reached any more, it is considered to be blocked. In figures 2.1 and 2.2, state `BWorking` is marked. Thus, the task is completed when machine B is working and machine A is idle. Both systems are nonblocking: from every state reachable from the initial state `idle`, it is possible to reach the marked state `BWorking`. The system in figure 2.3 is blocking, since the marked state cannot be reached, for example, from the initial state `idle`. Therefore, by deriving a controller from the model, we lost the property of being nonblocking. Even if a model is nonblocking, the actual controller realisation may be blocking. We refer to this problem as the *command selection problem*.

The aim of this thesis is to solve the command selection problem for a class of discrete event systems characterised by three properties. We introduce Ξ-terminating, Ξ-confluent, and Ξ-nonblocking discrete event systems. These systems are nonblocking for each controller realisation: any possible controller derived from such a model is nonblocking. Note that we do not provide a selection procedure to choose the next command, but we ensure that each selection procedure behaves correctly.

Consider another example. Assume that the controller uses a timer, which is part of the plant. An automaton model of the timer is given in figure 2.4. The timer is started by the controllable event Timer. After starting the timer, an uncontrollable event *TIMEOUT* will occur unless the timer is reset by the controllable event Reset.

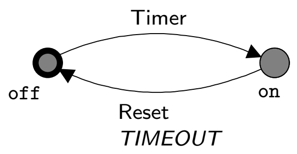

Figure 2.4: A timer model.

When identifying controllable events as output and uncontrollable events as input of the controller, it is still not defined how the controller proceeds in state `on` of figure 2.4. The controller can either wait for input or produce an output. If it always waits for input whenever input is possible, the timer cannot be reset. In this thesis, we assume that the controller does not withhold output, i.e. that it always reacts as quickly as possible. Therefore, the obtained results of this thesis are most suitable for controllers that react faster than the system to be controlled. These are systems for which commands sent by the controller are seldom interrupted by new sensor messages.

The *communication problem* is another problem related to building controllers from supervisory control models. It occurs when communication between plant and controller must be taken into account. Consider the timer model in figure 2.4 once more, and assume that the timer is running. If the controller sends a reset command at exactly the same time as the environment produces a timeout, then the controller receives a *TIMEOUT* message and the environment receives a Reset message when in state `off`. Both behaviours are not modelled, and it is not clear how plant and controller react in such a situation. Since such behaviour may be a serious problem when running a controller, special precautions must be taken to avoid it. The communication problem gets worse when communication delays may occur.

Although we do not give a solution to the communication problem in this thesis, the results obtained can still be used by modelling communication explicitly. Each message is modelled by means of a sending and a receiving event. However, this approach increases the complexity of the model and is therefore only applicable to small models. Subject to future work is the development of techniques allowing the extraction of controllers that are not affected by the communication problem without modelling communication explicitly.

In this thesis, we introduce Ξ-terminating, Ξ-confluent, and Ξ-nonblocking discrete event systems. We show that these systems do not suffer from the command selection problem. We develop checking algorithms, which can be used to develop Ξ-terminating, Ξ-confluent, and Ξ-nonblocking systems. Unlike most other approaches in supervisory control, we do not concentrate on synthesis algorithms, but we develop checking algorithms for the introduced properties. So far, it is left to the designer to fix the model if it does not satisfy all three properties. Counter-examples, which can be automatically computed, point to problems in the design, and give hints on how the model can be fixed.

The decision to concentrate on checking algorithms was made after the author got some experience in the design of controllers for discrete event systems. Starting from an informal description of system and requirements, developing a formal model is vulnerable to errors and omission. Under these circumstances, the author found it more convenient to use checking algorithms rather than synthesis algorithms to develop the desired controller. If the uncontrolled system model and the requirements are known to be correct, then synthesis algorithms are more convenient to use. It is left to future work to develop a universal synthesis algorithm for controllers that are not affected by the command selection problem.

2.2 Related Work

The theory of supervisory control was originally initiated by Ramadge and Wonham [RW82, RW87b, WR87]. Since then, supervisory control has been extended, applied, and studied by many researchers. See Ramadge and Wonham [RW89], Wonham [Won99], and Cassandras and Lafortune [CL99] for more detailed descriptions of the theory and more references to related research activities. This subsection starts with a short overview over research related to the theory of supervisory control. Afterwards, we discuss approaches that deal with implementation aspects of controllers.

One of the main obstacles to the use of formal methods in industry is the complexity of the models to be designed, synthesised, or verified and the resulting difficulty to deal with these systems efficiently. To be of practical relevance, complexity issues have to be mastered. Symbolic state space representation, which is known to be very successful for model checking [BCM+92, McM93], can also be applied to discrete event systems [HWT92, BHG+93, Zil02, ZW02]. Modular design [RW87a, WR88, WW98] allows the decomposition of complex systems into simpler components, which is the basis for describing complex industrial applications. Incremental algorithms [BMD00] exploit the modular structure of the systems to verify certain properties. But synthesis still suffers from complexity problems, which is the starting point of many research activities. Some approaches restrict themselves to systems with special structure [Ram87, LBW00, Led02].

Another possibility is to consider decentralised supervisory control [LW88a, RW92a], which is also of practical interest on its own. Many variations of how supervisors communicate exist [PKK97, BL98b, BL98a, YL00]. Decentralised supervisors have only partial information of the entire system, which opens the view to another interesting research field. Control with partial observations is considered in [CDFV88, LW88b, CM89].

A classical (untimed) discrete event model only observes the order of events. Timed systems [BW92, BW94] have been introduced to be able to model timing constraints. As in untimed systems, many extensions exist. Timed systems under partial observation [LW94] and hierarchical control [WW96] are considered. With time, a new dimension is introduced into modelling and control of discrete event systems, which increases the complexity of the systems as well. Therefore, strategies for avoiding the blow up of the system space [GW00] are necessary.

Supervisory control theory has been applied to several application domains, for example, manufacturing systems [RSO94, LW95, Led96, KCM00], chemical systems [HK94], automated traffic control [Var93], and communication protocols [RW92b, HESV93, Var93, MM02]. An application from the semiconductor industry for controlling a rapid thermal multiprocessor has been presented in [BHG+93].

Anyhow, most examples are academic and few industrial applications exist. An important reason seems to be the difficulty to implement a supervisor in a way required in industry [FH98, CAD99, Mal02]. The command selection problem and the communication problem introduced in the previous subsection are some of the problems that have to be mastered. In the following, we introduce and discuss approaches that aim at solving these problems.

Forced events [GR87] can be used to solve the communication problem. Unfortunately, this approach can be only used for some application domains such as synchronous environments. This approach assumes that forcing an event preempts the occurrence of other events. But in general, this cannot be guaranteed in asynchronous environments. Balemi [Bal94] proposes an input/output interpretation for the control of discrete event systems that is comparable with our interpretation of actuators and sensors. He characterises a class of supervisors and states conditions on the plant behaviour in order to prove that the closed-loop systems for these models affected by communication delays behave as expected, and thus solves the communication problem for a class of systems.

In [MCK99] an optimal controller is synthesised for which, regardless of the state of the plant, the worst cost of all paths from that state to a marked state is minimised. This approach solves the command selection problem, but for acyclic discrete event processes only. Spectool [AH98, HCOG00], a set of software tools based on Petri nets, is able to synthesise controllers. It converts high-level specifications of the desired plant behaviour into control code by detailing sequences of actuation and sensing.

Arnold, Vincent, and Walukiewicz [AVW02] extend the theory of supervisory control by expressing specifications as well as constraints on the controller in the modal μ-calculus. This enables the synthesis of models in which at most one output event is eligible at each state. A controller derived from such a model does not suffer from the command selection problem. Unfortunately, the synthesis is only applicable to quite small systems. Furthermore, the algorithm selects an arbitrary solution out of the set of possible solutions, which does not necessarily represent the behaviour expected by the designer. If the synthesised model is not suitable, more requirements must be given and synthesis must be repeated.

The approach proposed in this thesis solves the command selection problem. Properties are given which, when met by a model, ensure that each derived controller can be used and is not affected by the command selection problem. Modular design is supported, and the introduced incremental algorithms can handle real-world systems. Furthermore, the designer of a controller has the freedom to choose the desired controller out of a set of solutions. This enables code generation of an optimised controller without affecting the design process of the model.

Chapter 3

Notations

This thesis surveys modelling aspects of discrete event systems. Most of the material is based on the theory of formal languages, which provides a modelling independent means of representing such systems. Finite-state automata are used in examples and when algorithms are described. The newly introduced properties of discrete event systems are motivated by properties of rewrite systems.

In this chapter, the notations and concepts used throughout this thesis are introduced. Readers familiar with rewrite systems, formal languages, finite-state automata, and supervisory control theory may skip this chapter and use it as a reference for notations.

We start with some basic mathematical notations and definitions followed by a short introduction into rewrite systems, languages, and automata theory in section 3.1. In section 3.2, the basic concepts and notations for supervisory control of discrete event systems are presented.

3.1 Basics

3.1.1 Basic Notations

We assume that the reader is familiar with basic set theory. Let $\mathbb{N} = \{1, 2, \ldots\}$ denote the set of all natural numbers, and let $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$ be the set of all natural numbers together with zero.

In the following, we introduce some important properties of binary relations. A *binary relation* R on a set S is a subset of $S \times S$, the Cartesian product of S with itself. For $x, y \in S$, we also write $x \; R \; y$ instead of $(x, y) \in R$. We start by defining partial orderings.

Definition 3.1.1 (partial ordering)
Let S be a set. A binary relation $R \subseteq S \times S$ which is

(PO1) *reflexive, i.e. for all $x \in S$ we have $(x, x) \in R$,*

(PO2) *anti-symmetric, i.e. for all* $x, y \in S$ *we have that* $(x, y) \in R$ *and* $(y, x) \in R$ *implies* $x = y$*, and*

(PO3) *transitive, i.e. for all* $x, y, z \in S$ *we have that* $(x, y) \in R$ *and* $(y, z) \in R$ *implies* $(x, z) \in R$

is said to be a partial order *on* S*.*

Equivalence relations play an important role in language and automata theory, and thus throughout this thesis. Next, we provide a formal definition of an equivalence relation.

Definition 3.1.2 (equivalence relation)
Let S *be a set. A relation* $R \subseteq S \times S$ *which is*

(ER1) *reflexive, i.e. for all* $x \in S$ *we have* $(x, x) \in R$*,*

(ER2) *symmetric, i.e. for all* $x, y \in S$ *we have that* $(x, y) \in R$ *implies* $(y, x) \in R$*, and*

(ER3) *transitive, i.e. for all* $x, y, z \in S$ *we have that* $(x, y) \in R$ *and* $(y, z) \in R$ *implies* $(x, z) \in R$

is said to be an equivalence relation *on* S*.*
If R *is an equivalence relation, we also write* $x \equiv y \bmod R$ *instead of* $(x, y) \in R$*.*

An important property of an equivalence relation R on a set S is that R partitions S into disjoint nonempty equivalence classes. For $x \in S$, we define

$$[x]_R = \{\, y \in S \mid (x, y) \in R \,\}$$

to be the *equivalence class* of x with respect to R. If R is clear from the context, we also write $[x]$ instead of $[x]_R$. The set of all equivalence classes is defined according to

$$S/R = \{[x] \mid x \in S\}.$$

3.1.2 Rewrite Systems

This section summarises some standard notations and results of rewrite systems. For a comprehensive introduction and proofs of the given results see Dershowitz and Jouannaud [DJ90] or Avenhaus [Ave95].

A *rewrite system* is a binary relation $\rightarrow$ on a set S used to describe transformations of the elements of S. An element of S can be repeatedly replaced by other elements of S. *Derivations*, that are finite or infinite chains $s_0 \rightarrow s_1 \rightarrow \ldots$ of elements of S, are used to represent such rewrites.

We use the following notations. Let $\rightarrow$ be a binary relation. Then the binary relations $\leftarrow$ and $\leftrightarrow$ are defined according to

- $x \leftarrow y$ if and only if $y \rightarrow x$ and
- $x \leftrightarrow y$ if and only if $x \rightarrow y$ or $y \rightarrow x$.

Furthermore, we write R^* for the reflexive-transitive closure of an arbitrary binary relation R. For example, $\xrightarrow{*}$ is the reflexive-transitive closure of $\rightarrow$.

A binary relation is *terminating* if there does not exist an infinite derivation, i.e. only finite derivations are possible. A binary relation $\rightarrow$ on a set S is *confluent* if two finite derivations starting from the same element can be continued to a common element of S. More formally, the relation $\rightarrow$ is confluent if $x \xrightarrow{*} y_1$ and $x \xrightarrow{*} y_2$ for some $x, y_1, y_2 \in S$ imply that there exists $z \in S$ such that $y_1 \xrightarrow{*} z$ and $y_2 \xrightarrow{*} z$. For a binary relation $\rightarrow$ on a set S, an element $x \in S$ is *irreducible* if there is no $y \in S$ such that $x \rightarrow y$. An element $x \in S$ is a *normal form* for $y \in S$ if $y \xrightarrow{*} x$ and x is irreducible.

An important result is that for terminating and confluent rewrite systems the choice among possible rewrites at each step does not affect the normal form computed.

Proposition 3.1.1
A binary relation on a set S is terminating and confluent if and only if each element of S has a unique normal form.

Termination also provides a means of checking for confluence. A binary relation $\rightarrow$ on a set S is *locally confluent* if $x \rightarrow y_1$ and $x \rightarrow y_2$ for some $x, y_1, y_2 \in S$ imply that there exists $z \in S$ such that $y_1 \xrightarrow{*} z$ and $y_2 \xrightarrow{*} z$. In contrast to confluence, only derivations of length one that lead to different elements are considered.

Proposition 3.1.2 (Diamond or Newman lemma)
A terminating relation is confluent if and only if it is locally confluent.

This fundamental result provides the basis for an efficient check whether a terminating relation is confluent or not.

3.1.3 Languages

This subsection introduces basic notations on formal languages, which are used throughout this thesis to represent system behaviours.

An *alphabet* is a finite set of symbols. For an alphabet Σ, let Σ^+ denote the set of all finite *strings* (or *words*) of the form $\sigma_1\sigma_2 \ldots \sigma_k$, for which $k \in \mathbb{N}$ and $\sigma_i \in \Sigma$ for each $1 \leq i \leq k$. The *empty string* is denoted by ε; we assume that $\varepsilon \notin \Sigma$. Define $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. The *concatenation* of two strings is defined as

$$\mathrm{cat}\colon \Sigma^* \times \Sigma^* \rightarrow \Sigma^*;\;\; \mathrm{cat}(s,t) \mapsto st.$$

Let Σ be an alphabet. The *length* $|s|$ of a string $s \in \Sigma^*$ is defined according to $|\varepsilon| = 0$ and $|\sigma_1 \ldots \sigma_k| = k$, where $\sigma_1, \ldots, \sigma_k \in \Sigma$. The number of occurrences of $\sigma \in \Sigma$ in a string $s \in \Sigma^*$ is denoted by $|s|_\sigma$. We have $|\varepsilon|_\sigma = 0$, $|s\sigma|_\sigma = |s|_\sigma + 1$, and $|s\sigma'|_\sigma = |s|_\sigma$ if $\sigma \neq \sigma'$. For $s, t \in \Sigma^*$, if $s = tu$ for some $u \in \Sigma^*$, then t is a *prefix* of s, written $t \sqsubseteq s$.

A *language* over Σ is any subset $\mathcal{L} \subseteq \Sigma^*$. The *prefix-closure* $\overline{\mathcal{L}}$ of $\mathcal{L} \subseteq \Sigma^*$ is the set of all prefixes of strings in $\mathcal{L}$, i.e.

$$\overline{\mathcal{L}} = \{\, t \in \Sigma^* \mid t \sqsubseteq s \text{ for some } s \in \mathcal{L} \,\}.$$

If $\mathcal{L} = \overline{\mathcal{L}}$, then $\mathcal{L}$ is called *prefix-closed*. The *left quotient* of a language $\mathcal{L} \subseteq \Sigma^*$ by a string $s \in \Sigma^*$ is defined as

$$\mathcal{L}/s = \{\, t \in \Sigma^* \mid st \in \mathcal{L} \,\}.$$

The left quotient describes the possible *continuations* of a string in a language. Note that $(\mathcal{L}/s)/t = \mathcal{L}/st$ for all $s, t \in \Sigma^*$. The *eligible symbols* after a string $s \in \Sigma^*$ in a language $\mathcal{L} \subseteq \Sigma^*$ are

$$\textsc{Elig}_{\mathcal{L}}(s) = \{\, \sigma \in \Sigma \mid s\sigma \in \mathcal{L} \,\}.$$

This set describes the set of continuations of length 1. Note that $\textsc{Elig}_{\mathcal{L}}(s) = \emptyset$ if $s \notin \mathcal{L}$.

Furthermore, we use *natural projection* in order to restrict strings to a given set of events. For $\Sigma' \subseteq \Sigma$, we define the projection function $P_{\Sigma'}$ as the function which deletes all events not in Σ' from a given string:

$$\begin{array}{rcl} P_{\Sigma'}(\varepsilon) & = & \varepsilon; \\ P_{\Sigma'}(s\sigma) & = & \begin{cases} P_{\Sigma'}(s)\sigma, & \text{if } \sigma \in \Sigma'; \\ P_{\Sigma'}(s), & \text{otherwise.} \end{cases} \end{array}$$

Two strings of a language are called *Myhill-Nerode equivalent* if their sets of continuations in this language are equivalent. More formally, let $\mathcal{L} \subseteq \Sigma^*$ be a language, and let $s, t \in \mathcal{L}$. The Myhill-Nerode equivalence $\text{Nerode}(\mathcal{L}) \subseteq \Sigma^* \times \Sigma^*$ is defined as

$$s \equiv t \bmod \text{Nerode}(\mathcal{L}) \quad \text{if and only if} \quad \mathcal{L}/s = \mathcal{L}/t.$$

Myhill-Nerode equivalence is an equivalence relation.

An equivalence relation $R \subseteq \Sigma^* \times \Sigma^*$ is called *right invariant* (with respect to concatenation) if $x \equiv y \bmod R$ implies $xz \equiv yz \bmod R$ for all $x, y, z \in \Sigma^*$. Note that Myhill-Nerode equivalence $\text{Nerode}(\mathcal{L})$ is right invariant.

3.1.4 Automata

It is useful to have means of language representation. For our purposes, this is most conveniently done by representing languages by finite automata. The material of this section is standard theory of finite automata[1]; for an introduction and proofs of the given results see Hopcroft and Ullman [HU79] or Perrin [Per90].

Definition 3.1.3 (automaton)
A (finite-state) automaton *is a 5-tuple*

$$A = (\Sigma, Q, \Delta, J, M),$$

where

- **(A1)** *Σ is an alphabet of* events*,*
- **(A2)** *Q is a finite set of* states*,*
- **(A3)** *$\Delta \subseteq Q \times \Sigma \times Q$ is the* state transition relation*,*
- **(A4)** *$J \subseteq Q$ is a nonempty set of* initial states*, and*
- **(A5)** *$M \subseteq Q$ is the set of* marked states*.*

For the sake of convenience, we use the notation Σ_A as a shorthand for the event set the automaton A is defined over. Similarly, we use Q_A, Δ_A, J_A and M_A for the other components of the automaton. For automaton A above, $\Sigma_A = \Sigma$, $Q_A = Q$, $\Delta_A = \Delta$, $J_A = J$, and $M_A = M$.

If there exists a transition originating in a state and labelled with an event, then we say that this event is eligible in the state. We also say that this event may occur or may be executed at the given state.

Definition 3.1.4 (eligible event set)
Let $A = (\Sigma, Q, \Delta, J, M)$ be an automaton. The set of eligible events *in a state $p \in Q$ is defined as*

$$\text{ELIG}_A(p) = \{\, \sigma \in \Sigma \mid \textit{there exists } q \in Q \textit{ such that } (p, \sigma, q) \in \Delta \,\}.$$

Two automata are said to be isomorphic if they are equivalent with respect to unique state renaming.

Definition 3.1.5 (isomorphic automata)
Let A and B be automata over the alphabet Σ.
A and B are said to be isomorphic, written $A \cong B$, if there exists a bijective function (called isomorphism*)*

$$\varphi \colon Q_A \to Q_B$$

such that

- **(EQ1)** *$(p, \sigma, q) \in \Delta_A$ if and only if $(\varphi(p), \sigma, \varphi(q)) \in \Delta_B$,*
- **(EQ2)** *$\varphi(J_A) = J_B$, and*
- **(EQ3)** *$\varphi(M_A) = M_B$.*

---

[1]Although a somewhat different definition of the language accepted by an automaton is used herein, the results are still the same.

Accepted Languages

To describe the languages accepted by an automaton $A = (\Sigma, Q, \Delta, J, M)$, the transition relation Δ is extended to operate on strings instead of single elements of the alphabet. Let $\Delta' \subseteq Q \times \Sigma^* \times Q$ be the smallest set which satisfies the following two conditions:

- We have $(p, \varepsilon, p) \in \Delta'$ for each $p \in Q$.
- For each $p, q \in Q$, $s \in \Sigma^*$, and $\sigma \in \Sigma$, we have $(p, s\sigma, q) \in \Delta'$ if and only if there exists $p' \in Q$ such that $(p, s, p') \in \Delta'$ and $(p', \sigma, q) \in \Delta$.

In the following we identify Δ' and Δ. Let $Q_1, Q_2 \subseteq Q$, and let $\mathcal{L} \subseteq \Sigma^*$. If there exist $q_1 \in Q_1$, $s \in \mathcal{L}$, and $q_2 \in Q_2$ such that $(q_1, s, q_2) \in \Delta$, we also write $(Q_1, \mathcal{L}, Q_2) \in \Delta$. If one of the sets Q_1, Q_2, or $\mathcal{L}$ contains exactly one element, we often use this element instead. For instance, given $s \in \Sigma^*$, the notation $(J, s, Q) \in \Delta$ means that there exist $q_0 \in J$ and $q \in Q$ such that $(q_0, s, q) \in \Delta$.

Now we can say when a string $s \in \Sigma^*$ is *accepted* by A; this is the case if there exist $q_0 \in J$ and $q \in Q$ such that $(q_0, s, q) \in \Delta$, i.e. there exists a path starting from an initial state and following transitions labelled by exactly the events occurring in the string s. Alternatively, if we are interested in *marked* acceptance, we require that the path ends in a marked state, i.e. there exists $q_0 \in J$ and $q_m \in M$ such that $(q_0, s, q_m) \in \Delta$. Now, we can formally define the language and the marked language accepted by an automaton.

Definition 3.1.6 (accepted language and marked languages)
Let $A = (\Sigma, Q, \Delta, J, M)$ be an automaton. We define the language $\mathcal{L}(A)$ accepted by A *and the* marked language $\mathcal{L}_m(A)$ accepted by A *as*

$$\begin{aligned} \mathcal{L}(A) &= \{ s \in \Sigma^* \mid (J, s, Q) \in \Delta \}; \\ \mathcal{L}_m(A) &= \{ s \in \Sigma^* \mid (J, s, M) \in \Delta \}. \end{aligned}$$

Note that isomorphic automata accept the same languages. More precisely, $\mathcal{L}(A) = \mathcal{L}(B)$ and $\mathcal{L}_m(A) = \mathcal{L}_m(B)$ if the automata A and B are isomorphic. Not for every language $\mathcal{L} \subseteq \Sigma^*$ there exists an automaton accepting $\mathcal{L}$. If a language is accepted by some automaton, it is called regular.

Definition 3.1.7 (regular language)
A language that is equal to the accepted or marked language of an automaton is said to be regular.

An important result is the Myhill-Nerode theorem. It gives a characterisation of regular languages by means of Myhill-Nerode equivalence.

Proposition 3.1.3 (Myhill-Nerode theorem)
Let $\mathcal{L}$ be a language. The following statements are equivalent.

- *$\mathcal{L}$ is regular.*
- Nerode($\mathcal{L}$) *has a finite number of equivalence classes.*

Paths, States, and Reachability

The transition relation of an automaton can be considered as a rewrite system on the states, which replaces states of the automaton by successor states reachable by a certain transition.

Let $A = (\Sigma, Q, \Delta, J, M)$ be an automaton, and let $\Sigma' \subseteq \Sigma$. Define the relation $\rightarrow^A_{\Sigma'} \subseteq Q \times Q$ as follows:

$$p \rightarrow^A_{\Sigma'} q \;\text{ if and only if there exists } \sigma' \in \Sigma' \text{ such that } (q, \sigma', q) \in \Delta. \tag{3.1}$$

If A is known from the context, we also write $\rightarrow_{\Sigma'}$ instead of $\rightarrow^A_{\Sigma'}$ and $\rightarrow$ instead of $\rightarrow^A_{\Sigma}$. We also write $p \xrightarrow{\sigma} q$ if $(p, \sigma, q) \in \Delta$.

A *path* in the automaton $A = (\Sigma, Q, \Delta, J, M)$ is a finite derivation

$$\pi \;=\; p_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} p_n \,,$$

for some $n \in \mathbb{N}_0$. Note that each state $p \in Q$ is a path in A. We use the following notations.

- $L(\pi) = \sigma_1 \cdots \sigma_n$ is the *label* of the path π.
- $s_o(\pi) = p_0$ is the *origin* of path π.
- $s_e(\pi) = p_n$ is the *end* of path π.
- $|\pi| = n$ is the length of path π.

The path π is said to be *marked* if $p_n \in M$. Let $\Xi \subseteq \Sigma$. If $L(\pi) \subseteq \Xi^*$, then π is also called a Ξ-*path* in A. Paths are closely related to the transition relation Δ. If $(p, s, q) \in \Delta$, then there exists a path π in A with origin p, end q, and label s, i.e. $s_o(\pi) = p$, $s_e(\pi) = q$, and $L(\pi) = s$. Conversely, if π is a path in A, then we have $(s_o(\pi), L(\pi), s_e(\pi)) \in \Delta$. A *self-loop* is a path of length one with the same origin and end. Two paths $\pi_1 = p_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_m} p_m$ and $\pi_2 = p_m \xrightarrow{\sigma_{m+1}} \cdots \xrightarrow{\sigma_n} p_n$ of an automaton A, for which $m \leq n$, can be *concatenated* to a new path $\pi_1\pi_2 = p_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} p_n$ in A, provided that $s_e(\pi_1) = s_o(\pi_2)$.

A state of an automaton is reachable if there exists a path originating from an initial state and ending in this state.

Definition 3.1.8 (reachable state)
Let $A = (\Sigma, Q, \Delta, J, M)$ be an automaton, and let $p, q \in Q$.
The state q is said to be reachable from state p *in A if there exists a path π in A such that $s_o(\pi) = p$ and $s_e(\pi) = q$. The state q is said to be* reachable *in A if q is reachable from an initial state. Write*

$$Q_{\text{reach}}(A) = \{\, q \in Q \mid q \textit{ is reachable in } A \,\}$$

for the set of all reachable states of an automaton.

The language and the marked language accepted by an automaton are only influenced by reachable states. Therefore, states that are not reachable may be deleted without changing the accepted languages. The resulting automaton is called the *reachable automaton* of the original automaton.

Definition 3.1.9 (reachable subautomaton)
Let $A = (\Sigma, Q, \Delta, J, M)$ be an automaton.
The reachable subautomaton *of A is defined as*

$$\mathcal{R}(A) = (\Sigma, Q_{\text{reach}}(A), \Delta_R, J, M \cap Q_{\text{reach}}(A)),$$

where

$$\Delta_R = \{ (p, \sigma, q) \in \Delta \mid p, q \in Q_{\text{reach}}(A) \}.$$

Note that $\mathcal{R}(\mathcal{R}(A)) = \mathcal{R}(A)$.

Synchronous Product

Two automata over the same alphabet can be combined to a new automaton by means of the synchronous product operation defined below.

Definition 3.1.10 (synchronous product)
Let $A = (\Sigma, Q_A, \Delta_A, J_A, M_A)$ and $B = (\Sigma, Q_B, \Delta_B, J_B, M_B)$ be automata. The synchronous product *of A and B is the automaton*

$$A \,\|\, B = (\Sigma, Q, \Delta, J_A \times J_B, M_A \times M_B),$$

where

$$\begin{aligned} Q &= Q_A \times Q_B \text{ and} \\ \Delta &= \{ ((p_1, q_1), \sigma, (p_2, q_2)) \in Q \times \Sigma \times Q \mid (p_1, \sigma, p_2) \in \Delta_A, (q_1, \sigma, q_2) \in \Delta_B \}. \end{aligned}$$

The state set of the composed automaton is the Cartesian product of the state sets of the original automata. An event is accepted at a certain state if it is accepted in the corresponding states of each original automaton.

Let A, B, and C be automata over the same alphabet. It can be easily verified that the synchronous product can be considered as a commutative and associative operation, i.e.

$$\begin{aligned} A \,\|\, B &\cong B \,\|\, A \\ A \,\|\, (B \,\|\, C) &\cong (A \,\|\, B) \,\|\, C \end{aligned}$$

The accepted and the marked language of the synchronous product of two automata are the intersections of the corresponding languages of the original automata. More formally, we have

$$\begin{aligned} \mathcal{L}(A \,\|\, B) &= \mathcal{L}(A) \cap \mathcal{L}(B) \\ \mathcal{L}_m(A \,\|\, B) &= \mathcal{L}_m(A) \cap \mathcal{L}_m(B) \end{aligned}$$

Deterministic Automata

An automaton is said to be deterministic if it has exactly one initial state and there exists at most one successor state for each state and event. This is formalised in the following definition.

Definition 3.1.11 (deterministic automaton)
Let $A = (\Sigma, Q, \Delta, J, M)$ be an automaton. A is said to be deterministic *if*

(DET1) $|J| = 1$ *and*

(DET2) $(p, \sigma, q_1) \in \Delta$ *and* $(p, \sigma, q_2) \in \Delta$ *imply that* $q_1 = q_2$, *for all* $p, q_1, q_2 \in Q$ *and* $\sigma \in \Sigma$.

A deterministic automaton, denoted by G according to [Won99] (where the phrase *generator* is used instead) can also be defined as a five-tuple

$$G = (\Sigma, Q, \delta, q_0, M),$$

where

(DA1) Σ is an alphabet of events,

(DA2) Q is a finite set of states,

(DA3) $\delta \colon Q \times \Sigma \to Q$ is the (partial) transition function,

(DA4) $q_0 \in Q$ is the initial state, and

(DA5) $M \subseteq Q$ is the set of marked states.

This is another representation for the automaton $(\Sigma, Q, \Delta, \{q_0\}, M)$ where the transition relation Δ satisfies

$$(p, \sigma, q) \in \Delta \quad \Leftrightarrow \quad \delta(p, \sigma) \text{ is defined and } \delta(p, \sigma) = q$$

for all $p, q \in Q$ and $\sigma \in \Sigma$.

Similar to non-deterministic automata, the transition function δ is extended to operate on strings instead of single elements of the alphabet, i.e. $\delta \colon Q \times \Sigma^* \to Q$. Note that the synchronous product of two deterministic automata is a deterministic automaton. Note further that

$$\begin{aligned} \mathcal{L}(G) &= \{ s \in \Sigma^* \mid \delta(q_0, s) \text{ defined} \}; \\ \mathcal{L}_m(G) &= \{ s \in \mathcal{L}(G) \mid \delta(q_0, s) \in M \}. \end{aligned}$$

A consequence from the Myhill-Nerode theorem is that there is a unique minimum-state deterministic automaton for every regular language.

Lemma 3.1.4
The minimum-state deterministic automaton accepting a language is unique up to an isomorphism, i.e. a renaming of states.

Graph Representation of Automata

For illustration purposes, directed graphs are used to represent automata. The nodes of the graph correspond to the states of the automaton. Edges represent state transitions and are labelled with event names. Initial states are bordered black. Marked states are shaded. Figure 3.1 shows a representation of the automaton $A_1 = (\Sigma_1, Q_1, \Delta_1, J_1, M_1)$, where

$$\begin{aligned}
\Sigma_1 &= \{\mathsf{a}, \mathsf{b}\}, \\
Q_1 &= \{\mathsf{s0}, \mathsf{s1}\}, \\
\Delta_1 &= \{(\mathsf{s0}, \mathsf{a}, \mathsf{s0}), (\mathsf{s1}, \mathsf{a}, \mathsf{s1}), (\mathsf{s0}, \mathsf{a}, \mathsf{s1}), (\mathsf{s0}, \mathsf{b}, \mathsf{s1})\}, \\
J_1 &= \{\mathsf{s0}\}, \\
M_1 &= \{\mathsf{s1}\}.
\end{aligned}$$

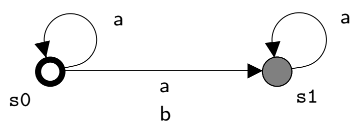

Figure 3.1: A graph representation of an automaton.

Sometimes so-called *superstates*, graphically represented as a box, are used to improve readability. A transition that originates from a superstate is a shorthand for transitions originating from each of the states contained. Figure 3.2 shows a representation of the automaton $A_2 = (\Sigma_2, Q_2, \Delta_2, J_2, M_2)$, where

$$\begin{aligned}
\Sigma_2 &= \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}, \\
Q_2 &= \{\mathsf{s0}, \mathsf{s1}\}, \\
\Delta_2 &= \{(\mathsf{s0}, \mathsf{a}, \mathsf{s0}), (\mathsf{s1}, \mathsf{a}, \mathsf{s1}), (\mathsf{s0}, \mathsf{b}, \mathsf{s0}), (\mathsf{s1}, \mathsf{b}, \mathsf{s0}), (\mathsf{s0}, \mathsf{c}, \mathsf{s1})\}, \\
J_2 &= \{\mathsf{s0}\}, \\
M_2 &= \{\mathsf{s1}\}.
\end{aligned}$$

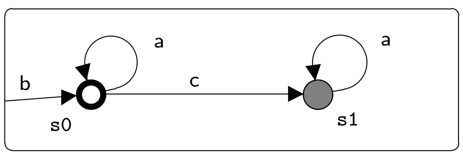

Figure 3.2: A graph representation containing a superstate.

Usually, only relevant events for an automaton are given in its graph representation. An event is relevant for an automaton if the event is either disabled at some state or the occurrence of this event may cause a state change in the automaton.

Definition 3.1.12 (relevant, irrelevant events)
Let $A = (\Sigma, Q, \Delta, J, M)$ be an automaton. The event $\sigma \in \Sigma$ is irrelevant *for A if, for each $p \in Q$, we have that*

(REL1) $(p, \sigma, p) \in \Delta$ *and*

(REL2) $(p, \sigma, q) \in \Delta$ *for some $q \in Q$ implies $p = q$.*

If there exists a state such that (REL1) *or* (REL2) *is not satisfied, then σ is* relevant *for A.*

The event a is relevant for automaton A_1 in figure 3.1 and irrelevant for automaton A_2 in figure 3.2. If the event set is given, irrelevant events for an automaton are usually omitted in its graph representation. If the event set is known to be Σ_2, we can represent the automaton A_2 given in figure 3.2 by the graph in figure 3.3.

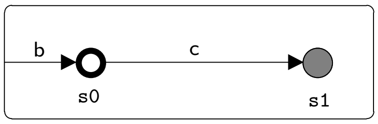

Figure 3.3: A graph representation without irrelevant events.

Quotient Automata

Given an automaton A and an equivalence relation $\sim$ on the state set of A, we can construct a new automaton $A/{\sim}$ using the set of equivalence classes with respect to $\sim$ as the new state set. The new automaton has a transition between two classes if there exists a transition between two states belonging to these two classes in the original automaton. A class is initial (marked) in the new automaton if there exists an initial (marked) state belonging to this class.

Definition 3.1.13 (quotient automaton)
Let $A = (\Sigma, Q, \Delta, J, M)$ be an automaton, and let $\sim \subseteq Q \times Q$ be an equivalence relation. We define the quotient automaton $A/{\sim}$ of A as

$$A/{\sim} = (\Sigma, Q/{\sim}, \widetilde{\Delta}, \widetilde{J}, \widetilde{M}),$$

where

$$\begin{aligned}
\widetilde{\Delta} &= \{\, ([p], \sigma, [q]) \in Q/{\sim} \times \Sigma \times Q/{\sim} \mid (p, \sigma, q) \in \Delta \,\}, \\
\widetilde{J} &= \{\, [q_0] \in Q/{\sim} \mid q_0 \in J \,\}, \\
\widetilde{M} &= \{\, [q_m] \in Q/{\sim} \mid q_m \in M \,\}.
\end{aligned}$$

Next, we show that each path in an automaton is also a path in the quotient automaton.

Lemma 3.1.5
Let $A = (\Sigma, Q, \Delta, J, M)$ be an automaton, let $\sim\ \subseteq Q \times Q$ be an equivalence relation, and let

$$\pi = p_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} p_n$$

be a path in A. Then

$$\tilde{\pi} = [p_0] \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} [p_n]$$

is a path in $A/{\sim}$.

Proof. Let $A = (\Sigma, Q, \Delta, J, M)$ be an automaton, let $\sim\ \subseteq Q \times Q$ be an equivalence relation, let $\pi = p_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} p_n$ be a path in A, and let $\tilde{A} = A/{\sim}$. Let $i \in \{0, \ldots, n-1\}$. Since π is a path in A, we have $(p_i, \sigma_{i+1}, p_{i+1}) \in \Delta$. It follows by definition 3.1.13 that $([p_i], \sigma_{i+1}, [p_{i+1}]) \in \Delta_{\tilde{A}}$. This proves that $[p_0] \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} [p_n]$ is a path in $\tilde{A}$. $\square$

This result also proves that $\mathcal{L}(A) \subseteq \mathcal{L}(A/{\sim})$.

3.2 Supervisory Control of Discrete Event Systems

In this section, we introduce basic notations of supervisory control theory. More detailed introductions can be found in Wonham [Won99] or Cassandras and Lafortune [CL99].

3.2.1 Discrete Event Systems

In the framework of Ramadge and Wonham, discrete event systems (DES) are modelled as generators of two formal languages $\mathcal{L}$ and $\mathcal{L}_m$ such that $\mathcal{L}$ is a prefix-closed language describing the possible behaviours of the system, and $\mathcal{L}_m \subseteq \mathcal{L}$ is a language describing completed tasks.

Definition 3.2.1 (DES)
A DES is a tuple $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ such that

- Σ *is an alphabet of* events*,*
- $\emptyset \neq \mathcal{L} \subseteq \Sigma^*$ *is a prefix-closed language, and*
- $\mathcal{L}_m \subseteq \mathcal{L}$ *is the* marked language.

Automata Representation of DES

Automata can be used to represent discrete event systems. Let A be an automaton. We have $\mathcal{L}_m(A) \subseteq \mathcal{L}(A)$. Furthermore, $\mathcal{L}(A)$ is prefix-closed. Therefore, the automaton A represents the DES

$$(\Sigma_A, \mathcal{L}(A), \mathcal{L}_m(A)).$$

In the following, we identify deterministic automata and the represented DES. We say that a deterministic automaton has a certain property in order to state that the corresponding DES has this property. For example, we will say that a deterministic automaton is "nonblocking" (as defined below) if its DES has this property.

Note that not every DES can be represented by means of a (finite-state) automaton, since the language and the marked language accepted by an automaton is necessarily regular. A DES containing a non-regular language cannot be represented by an automaton.

Nonblocking DES

The nonblocking property is crucial for discrete event systems. It states that it is always possible to complete a task, i.e. to continue a string of the generated language to a string of the marked language. Formally, this is the case if all strings of the generated language are contained in the prefix-closure of the marked language.

Definition 3.2.2 (blocking DES, nonblocking DES)
Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES. D is said to be nonblocking *if*

$$\mathcal{L} \subseteq \overline{\mathcal{L}_m},$$

otherwise D is said to be blocking.

In other words, we can say that a DES $(\Sigma, \mathcal{L}, \mathcal{L}_m)$ is nonblocking if for each $s \in \mathcal{L}$ there exists $t \in \Sigma^*$ such that $st \in \mathcal{L}_m$.

Myhill-Nerode equivalence

The Myhill-Nerode equivalence describes sets of strings that have the same continuations in the languages of a DES.

Definition 3.2.3 (Myhill-Nerode equivalence)
Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES. The Myhill-Nerode equivalence $\mathrm{Nerode}(D)$ *over D is defined as*

$$\mathrm{Nerode}(D) = \mathrm{Nerode}(\mathcal{L}) \cap \mathrm{Nerode}(\mathcal{L}_m).$$

3.2.2 Supervisors and Closed-Loop Systems

A supervisor influences the behaviour of a discrete event system, which represents the system to be controlled, in such a way that the control objectives are met. The event set is partitioned into *controllable* and *uncontrollable* events. A supervisor can only enable or disable controllable events, uncontrollable events cannot be disabled by a supervisor.

Definition 3.2.4 (supervisor)
Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, and let $\Sigma_c \subseteq \Sigma$ be the set of controllable events of D. A supervisor *S for D is a function*

$$S\colon \Sigma^* \to 2^{\Sigma}$$

such that $\Sigma \backslash \Sigma_c \subseteq S(s)$ for all $s \in \Sigma^*$.

A supervisor maps each string to the set of events enabled after the occurrence of that string. A system D can be controlled by a supervisor S in the sense that the controllable events are dynamically enabled or disabled by S. Figure 3.4 shows the general structure of a supervisor interacting with a discrete event system. Here, s is the string of all events generated so far by D, and $S(s)$ is the set of events enabled after s. The controlled system, denoted by S/D (S controlling D), is a DES and its closed-loop behaviour is defined as follows.

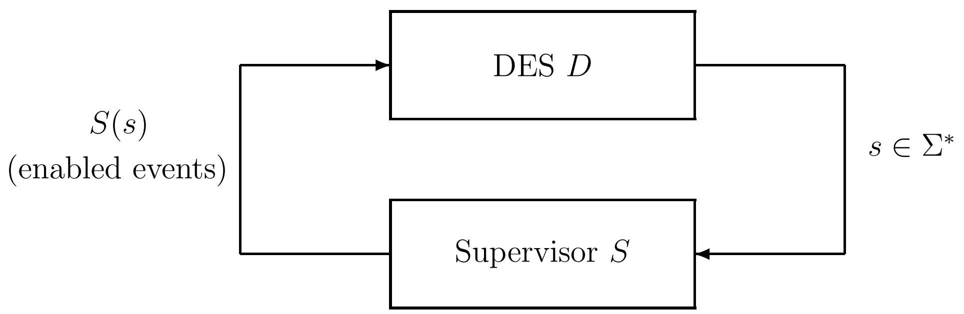

Figure 3.4: The closed-loop system.

Definition 3.2.5 (closed-loop system, controlled system)
Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, and let S be a supervisor for D.
The closed-loop system *(or* controlled system*) of S controlling D is defined as*

$$S/D = (\Sigma, \mathcal{L}(S/D), \mathcal{L}_m(S/D)),$$

where $\mathcal{L}(S/D)$ is defined recursively as follows:

- $\varepsilon \in \mathcal{L}(S/D)$
- $s\sigma \in \mathcal{L}(S/D) \Leftrightarrow s \in \mathcal{L}(S/D)$, $s\sigma \in \mathcal{L}$, *and* $\sigma \in S(s)$

and

- $\mathcal{L}_m(S/D) = \mathcal{L}(S/D) \cap \mathcal{L}_m$.

The languages of the DES S/D contains all strings of D which remain feasible under control by S. A supervisor restricts the behaviour of the given DES: the language of the controlled system is contained in the language of the uncontrolled system. This is summarised in the following lemma.

Lemma 3.2.1
Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, and let S be a supervisor for D. Then we have

$$\mathcal{L}(S/D) \subseteq \mathcal{L}$$

and

$$\mathcal{L}_m(S/D) \subseteq \mathcal{L}_m .$$

Furthermore, we want the controlled system to be nonblocking.

Definition 3.2.6 (blocking, nonblocking supervisor)
Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES. A supervisor S for D is said to be nonblocking *if*

$$\overline{\mathcal{L}_m(S/D)} = \mathcal{L}(S/D),$$

i.e. if the closed-loop system S/D is nonblocking; otherwise it is blocking.

Ramadge and Wonham show that if there exists a nonblocking supervisor for a DES such that the closed-loop system is contained in a specification language, then there exists a most general supervisor that is contained in the specification language and is nonblocking. More precisely, consider a DES $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$, a specification language $E \subseteq \Sigma^*$, and a nonblocking supervisor T for D such that $\mathcal{L}(T/D) \subseteq E$. There exists a supervisor S for D such that

- S is nonblocking for D,
- $\mathcal{L}(S/D) \subseteq E$, and
- S is least restrictive, i.e. for each nonblocking supervisor S' for D such that $\mathcal{L}(S'/D) \subseteq E$ we have $\mathcal{L}(S'/D) \subseteq \mathcal{L}(S/D)$.

Note that the least restrictive supervisor S disables as little as possible. Ramadge and Wonham also show that this least restrictive supervisor is computable in the case of regular languages. Thus, the least restrictive supervisor can be generated automatically.

Chapter 4

Ensuring Nonblocking for Each Controller Realisation

In this chapter, we characterise a class of discrete event systems that do not suffer from the command selection problem described in section 2.1. For these systems, the choice among possible commands does not affect the possibility to complete the current task: the controller cannot block the completion of the task by choosing "wrong" commands. A model contained in the characterised class can be easily used to generate a nonblocking controller. Each controller derived from such a model is necessarily nonblocking. We may choose a controller that is optimised in some sense and can still be sure that the chosen controller is nonblocking.

Furthermore, most controllers for real-world applications are supposed to satisfy the properties used to represent this new class of discrete event systems. A controller usually reacts to an input by sending a finite number of commands. Each model contained in the class is Ξ-terminating, i.e. no infinite command sequence is possible. Therefore, each controller derived from such a model also satisfies this desired property; such a controller cannot send an infinite command sequence without getting new input.

Usually, the controller reacts faster than the system to be controlled. Each model contained in the class to be characterised is Ξ-nonblocking. This implies that the current task can be completed by a behaviour in which the controller is not interrupted by the plant to be controlled. To complete the current task, the system does not have to relay on behaviours in which a new input arrives before the controller has completed sending its commands.

Models of the characterised class are Ξ-confluent. Together with Ξ-termination this ensures that if there is a choice between different commands, this does not affect the future behaviour of the system as long as the controller is not interrupted by a new input. Different command sequences are considered as equal in the sense that the same target state can be reached. This ensures that a controller may choose any of these command sequences and still achieves the desired functionality.

This section is organised as follows. In section 4.1, we introduce the concept of live Ξ-supervisors as abstract representations of controllers. By introducing the properties of Ξ-termination in section 4.2, Ξ-confluence in section 4.3, and Ξ-nonblocking in section 4.4, we characterise systems that are nonblocking for each live Ξ-supervisor. This is proven in section 4.5.

4.1 Live Ξ-Supervisors

Assume that a DES, which models a plant behaviour, and a (least restrictive) supervisor, which ensures that the controlled system behaves as desired, are given. In the following, we refer to this closed-loop system simply as a DES, also called (discrete event) system or model, and do not distinguish plant model and supervisor. From such a model, we want to generate a controller. More precisely, we want to generate software accepting input messages and generating output messages, also called commands. For this purpose, the event set of the model is partitioned into *input* events and *output* events.

In this thesis, the set of output events generated by a controller is denoted by Ξ. A controller must choose the next output to be given as a command from this set. It is natural to assume that output events are controllable, but in principle this assumption is not required. Furthermore, there may exist controllable events that are input events. It is possible to implement a controller which disables such events, if necessary, but does not initiate the occurrence of this events. However, since we are interested in output events, we do not distinguish between input events that are controllable and input events that are uncontrollable. In both cases, the controller must be informed about the occurrence of such events, i.e. sensors must provide the necessary information if an input event has occurred.

A controller *implements* a discrete event system for a given plant if the composition of plant and controller behaves as specified in the model. On the one hand, the controller may only send a message if the corresponding event is enabled in the model. On the other hand, a controller must not withhold an output if the corresponding event is enabled in the model. Each event must have the chance to occur if it is eligible in the model.

Usually, several output events may be eligible at each state of the model. It is possible to think of a controller that chooses the next output at runtime, but many applications require *deterministic* controllers[1], for which each output is uniquely defined by the sequence of events occurred so far. For deterministic controllers, after the occurrence of an event sequence, there exists at most one output event which may be generated as the next output message. Thus, the

[1]Note that we consider the non-determinism caused by the choice of output events here. This is different from non-determinism in automata theory which is the possibility to enter different target states after the occurrence of the same event.

model accepts a larger behaviour than a deterministic controller exploits. We formalise such restrictions on a model by introducing live Ξ-supervisors.

Definition 4.1.1 (live Ξ-supervisor)
Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, and let $\Xi \subseteq \Sigma$.
A live Ξ-supervisor[2] *for D is a mapping $S\colon \Sigma^* \to 2^\Sigma$ such that*

(L1) $\Sigma \backslash \Xi \subseteq S(s)$ *and*

(L2) $\Xi \cap \mathrm{E{\scriptsize LIG}}_{\mathcal{L}}(s) \neq \emptyset$ *implies* $S(s) \cap \Xi \cap \mathrm{E{\scriptsize LIG}}_{\mathcal{L}}(s) \neq \emptyset$

for each $s \in \Sigma^*$.

A live Ξ-supervisor only restricts the occurrence of output events, but it must not disable every output event enabled in the model. The behaviour of a deterministic controller can be described by a live Ξ-supervisor. Therefore, we are interested in controllers implementing the closed-loop system of a model and a live Ξ-supervisor for this model.

For the introductory example in section 2.1, a model is given in figure 2.2 on page 6. Let $\Xi = \{\mathsf{startA}, \mathsf{startB}\}$ be the set of output events. A controller implementing this model is necessarily non-deterministic, since two output events are eligible at state `idle` and must have the chance to occur. The closed-loop system of this model and the live Ξ-supervisor that disables the starting of machine A when both machines are idle is given in figure 2.3 on page 7. A controller implementing this further restricted model is deterministic.

We have seen in section 2.1 that a deterministic controller derived from a nonblocking model may be blocking. Now we use live Ξ-supervisors to rephrase this statement in a more formal way: a live Ξ-supervisor for a nonblocking discrete event system may be blocking. In the following sections, we introduce Ξ-terminating, Ξ-confluent, and Ξ-nonblocking discrete event systems. We prove that each live Ξ-supervisors for such a model is nonblocking. Consequently, each deterministic controller derived from such a model is nonblocking.

4.2 Ξ-Termination

The first property used to ensure that each controller for a discrete event system is nonblocking is Ξ-termination. This property ensures that a controller reacts to an input by sending a finite sequence of outputs. The system stabilises if no input is given for a sufficiently long time. Nothing happens until new input is given. This property is formalised as follows.

[2]If Ξ is a set of controllable events, a live Ξ-supervisor is a supervisor as defined in definition 3.2.4 on page 26.

Definition 4.2.1 (Ξ-terminating language, Ξ-terminating DES)
Let Σ be an alphabet, and let $\mathcal{L} \subseteq \Sigma^*$ be a prefix-closed language.
$\mathcal{L}$ is said to be Ξ-terminating *if for each $s \in \mathcal{L}$ there exists $n \in \mathbb{N}$ such that $t \in \mathcal{L}/s$ with $|t| \geq n$ implies $t \notin \Xi^*$.*
A DES $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ is said to be Ξ-terminating *if $\mathcal{L}$ is Ξ-terminating.*

Assuming that $\Xi = \{\mathsf{startA}, \mathsf{startB}\}$, the DES given in figure 2.2 on page 6 is Ξ-terminating. Each string of the generated language can be continued by at most one output event. Assuming that $\Xi = \{\mathsf{Timer}, \mathsf{Reset}\}$, the DES given in figure 2.4 on page 8, which models a timer, is not Ξ-terminating. For all $n \in \mathbb{N}$, the empty string can be continued by n occurrences of Timer each followed by Reset.

In the following, we study some useful properties of terminating languages. The first lemma shows that the set of all continuations of a given string in a Ξ-terminating language is Ξ-terminating.

Lemma 4.2.1
Let Σ be an alphabet, let $\Xi \subseteq \Sigma$, and let $\mathcal{L} \subseteq \Sigma^*$ be Ξ-terminating.
Then, $\mathcal{L}/w$ is Ξ-terminating for each $w \in \Sigma^*$.

Proof. Let Σ be an alphabet, let $\Xi \subseteq \Sigma$, let $\mathcal{L}$ be a Ξ-terminating language over Σ, and let $w \in \Sigma^*$. Let $s \in \mathcal{L}/w$. Then $ws \in \mathcal{L}$ and, since $\mathcal{L}$ is Ξ-terminating, there exists $n \in \mathbb{N}$ such that $t \in (\mathcal{L}/w)/s = \mathcal{L}/ws$ with $|t| \geq n$ implies $t \notin \Xi^*$. □

For a Ξ-terminating DES, there is no infinite sequence consisting of output events. Then, assuming that only output events occur, the system will eventually stabilise. It will reach a state in which only input events are eligible. The history of a stabilised system is a Ξ-complete string of events defined below.

Definition 4.2.2 (Ξ-complete string)
Let $\mathcal{L}$ be a prefix-closed language over the alphabet Σ, and let $\Xi \subseteq \Sigma$.
A string $w \in \mathcal{L}$ is said to be Ξ-complete *in $\mathcal{L}$ if*

$$\Xi \cap \textsc{Elig}_{\mathcal{L}}(w) = \emptyset.$$

A Ξ-complete string in a language $\mathcal{L}$ is a string which cannot be continued with an output event, i.e. only continuations starting with an input event are possible in $\mathcal{L}$. Note that for all $s, t \in \Sigma^*$ such that $t \in \mathcal{L}/s$ we have that t is Ξ-complete in $\mathcal{L}/s$ if and only if st is Ξ-complete in $\mathcal{L}$.

Every Ξ-complete string in the closed-loop behaviour of a DES and a live Ξ-supervisor is also Ξ-complete in the open-loop behaviour of the DES. This is shown in the following lemma.

Lemma 4.2.2
Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, let $\Xi \subseteq \Sigma$, and let S be a live Ξ-supervisor for D.
If a string $w \in \mathcal{L}(S/D)$ is Ξ-complete in $\mathcal{L}(S/D)$, then w is Ξ-complete in $\mathcal{L}$.

Proof. Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, let $\Xi \subseteq \Sigma$, and let S be a live Ξ-supervisor for D. Let $w \in \mathcal{L}(S/D)$ be Ξ-complete in $\mathcal{L}(S/D)$. We have $w \in \mathcal{L}(S/D) \subseteq \mathcal{L}$. Since w is Ξ-complete in $\mathcal{L}(S/D)$, we obtain $\emptyset = \Xi \cap \textsc{Elig}_{\mathcal{L}(S/D)}(w) = \Xi \cap \textsc{Elig}_{\mathcal{L}}(w) \cap S(w)$. By property (L2) of definition 4.1.1, we obtain $\Xi \cap \textsc{Elig}_{\mathcal{L}}(w) = \emptyset$, and therefore w is Ξ-complete in $\mathcal{L}$. □

For a Ξ-terminating language, we finally end up with a Ξ-complete string if we continue an arbitrary string with output events.

Lemma 4.2.3
Let $\mathcal{L}$ be a prefix-closed language over the alphabet Σ, and let $\Xi \subseteq \Sigma$. If $\mathcal{L}$ is Ξ-terminating and $s \in \Sigma^*$, then the set $C_\Xi(s) = (\mathcal{L}/s) \cap \Xi^*$ satisfies the following properties:

- *$C_\Xi(s)$ is finite and*
- *for each $c \in C_\Xi(s)$ which is maximal in $C_\Xi(s)$ (i.e. for all $c' \in C_\Xi(s)$ we have that $c \sqsubseteq c'$ implies $c = c'$) it is the case that sc is Ξ-complete in $\mathcal{L}$.*

Proof. Let $\mathcal{L}$ be a prefix-closed language over the alphabet Σ, let $\Xi \subseteq \Sigma$, let $\mathcal{L}$ be Ξ-terminating, let $s \in \Sigma^*$, and let $C_\Xi(s) = (\mathcal{L}/s) \cap \Xi^*$. Since $\mathcal{L}$ is Ξ-terminating, there exists $n \in \mathbb{N}$ such that $C_\Xi(s) \subseteq \{\, c \in \Xi^* \mid |c| < n \,\}$. Since the right-hand side is a finite set, $C_\Xi(s)$ is also finite. Now let $c \in C_\Xi(s)$ be maximal, i.e. for all $c' \in C_\Xi(s)$ we have that $c \sqsubseteq c'$ implies $c = c'$. Assume sc is not Ξ-complete in $\mathcal{L}$, i.e. $\Xi \cap \textsc{Elig}_{\mathcal{L}}(sc) \neq \emptyset$. Let $\xi \in \Xi \cap \textsc{Elig}_{\mathcal{L}}(sc)$. Then we have $sc\xi \in \mathcal{L}$, so that $c\xi \in C_\Xi(s)$. But then c cannot be maximal in $C_\Xi(s)$, and the assumption that sc is not Ξ-complete in $\mathcal{L}$ must be wrong. This proves that sc is Ξ-complete in $\mathcal{L}$. □

4.3 Ξ-Confluence

Given the same history of events, it is acceptable that two controllers derived from the same model generate different output messages, i.e. different commands. However, we want to ensure that all controllers end up accepting the same input messages after generating their output messages. With the following property of Ξ-confluence, we ensure that independently of the choices taken by different controllers we can always reach states with the same future by using only output events. In accordance with the theory of term rewriting [DJ90], we define Ξ-confluent systems as follows.

Definition 4.3.1 (Ξ-confluent DES)
Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, and let $\Xi \subseteq \Sigma$. D is said to be Ξ-confluent *if for all $s \in \mathcal{L}$, $s_1, s_2 \in \Xi^*$ such that $ss_1, ss_2 \in \mathcal{L}$ there exist $t_1, t_2 \in \Xi^*$ such that $ss_1t_1, ss_2t_2 \in \mathcal{L}$ and $ss_1t_1 \equiv ss_2t_2 \bmod \mathrm{Nerode}(D)$.*

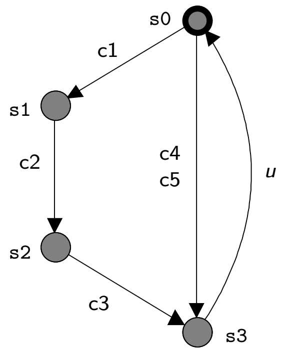

Figure 4.1: A {c1, c2, c3, c4, c5}-confluent DES.

Consider the DES represented in figure 4.1, and let $\Xi = \{\mathsf{c1}, \mathsf{c2}, \mathsf{c3}, \mathsf{c4}, \mathsf{c5}\}$ be the set of output events. This system is Ξ-confluent. At state s0 are three output events eligible. From this state, each string $s \in \Xi^*$ can be continued by a string $t \in \Xi^*$ such that state s3 is reached: c1 can be continued by c2 c3; string c1 c2 can be continued by c3; and strings c1 c2 c3, c4, and c5 can be continued by the empty string ε.

At state s0, a controller for this system may generate the output message c4; another controller may generate the output messages c1, c2, and finally c3. By doing this, both controllers reach a state that corresponds to state s3 in the model. The event sequences c1 c2 c3, c4, and c5 can be interpreted as different possibilities to reach a goal, and we do not want to care about which is chosen by a controller.

The DES given in figure 2.2 on page 6 is not Ξ-confluent assuming that $\Xi = \{\mathsf{startA}, \mathsf{startB}\}$, since there exist strings startA and startB which, starting from the initial state, cannot be continued to states with the same future using strings contained in Ξ^*. Therefore, the future system behaviour depends on which machine is started. In this case, the designer should refine the model to get a Ξ-confluent model.

In combination with Ξ-termination, Ξ-confluence ensures that, for each sequence of inputs received, all possible controllers, although generating different outputs, eventually end up accepting the same inputs. The following lemma shows that, independently from the messages generated, two controllers for a Ξ-terminating and Ξ-confluent model will finally reach states with the same future unless interrupted by input events.

Lemma 4.3.1
Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, let $\Xi \subseteq \Sigma$, and let S be a live Ξ-supervisor for D. Furthermore, let D be Ξ-terminating and Ξ-confluent, and let $s \in \mathcal{L}(S/D)$. For each $t \in \Xi^*$ such that st is Ξ-complete in $\mathcal{L}$, there exists a string $t' \in \Xi^*$ such that st' is Ξ-complete in $\mathcal{L}(S/D)$ and $st \equiv st' \bmod \mathrm{Nerode}(D)$.

Proof. Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, let $\Xi \subseteq \Sigma$, and let S be a live Ξ-supervisor for D. Furthermore, let D be Ξ-terminating and Ξ-confluent, and let $s \in \mathcal{L}(S/D)$. Let $t \in \Xi^*$ such that st is Ξ-complete in $\mathcal{L}$. Let $C = \Xi^* \cap \mathcal{L}(S/D)/s$. Since $\mathcal{L}(S/D) \subseteq \mathcal{L}$ we have $C \subseteq \Xi^* \cap \mathcal{L}/s$. Furthermore, C is nonempty, since $\varepsilon \in C$, and C is finite by lemma 4.2.3. Thus, there exists a string $t' \in C$ which is maximal. By lemma 4.2.3, st' is Ξ-complete in $\mathcal{L}(S/D)$. We have $st, st' \in \mathcal{L}$. Since D is Ξ-confluent, there exist $u, u' \in \Xi^*$ such that $stu, st'u' \in \mathcal{L}$ and $stu \equiv st'u' \bmod \mathrm{Nerode}(D)$. Since st is Ξ-complete in $\mathcal{L}$, we have $u = \varepsilon$. Since st' is Ξ-complete in $\mathcal{L}(S/D)$, st' is Ξ-complete in $\mathcal{L}$ by lemma 4.2.2. Thus, $u' = \varepsilon$. But this means that $st \equiv st' \bmod \mathrm{Nerode}(D)$. □

4.4 Ξ-Nonblocking

If a DES is nonblocking, then for all strings s of the generated language there exists a continuation t such that st is a marked string of the DES. There are no restrictions on the string t; it can be any string of events. Consider the DES given in figure 4.2, and let $\Xi = \{\mathsf{a}, \mathsf{b}\}$ be the set of output events. This DES is nonblocking, Ξ-terminating, and Ξ-confluent, but blocking for the live Ξ-supervisor disabling the event a at state `s0`. In order to capture such situations,

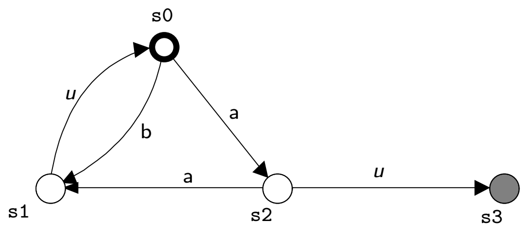

Figure 4.2: An $\{\mathsf{a},\ \mathsf{b}\}$-blocking DES.

we introduce a stronger definition of the nonblocking property by restricting strings such that preference is given to output events. A Ξ-controlled string is one in which input events occur only if no output event is eligible.

Definition 4.4.1 (Ξ-controlled string)
Let $\mathcal{L}$ be a prefix-closed language over the alphabet Σ, and let $\Xi \subseteq \Sigma$.
The string $w \in \Sigma^*$ is said to be Ξ-controlled *in $\mathcal{L}$ if*

- **(CC1)** *w is Ξ-complete in $\mathcal{L}$ and*
- **(CC2)** *for all $a \sqsubseteq w$ and $\iota \in \Sigma \backslash \Xi$ such that $a\iota \sqsubseteq w$ we have that the string a is Ξ-complete in $\mathcal{L}$.*

This definition is used to define a strengthened version of the nonblocking property. Note that we do not restrict the behaviour of the system; we only restrict strings used as continuations to a marked string.

Definition 4.4.2 (Ξ-blocking, Ξ-nonblocking DES)
Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, and let $\Xi \subseteq \Sigma$.
D is said to be Ξ-nonblocking *if for all $s \in \mathcal{L}$ there exists a Ξ-controlled string t in $\mathcal{L}/s$ such that $st \in \mathcal{L}_m$; otherwise D is said to be* Ξ-blocking.

The DES given in figure 4.2 is nonblocking, but Ξ-blocking. Starting at the initial state s0, the string a*u* leads to the marked state s3. But after the occurrence of event a, the output event a is eligible at state s2. Thus, the string a*u* is not Ξ-controlled. Moreover, we cannot find a Ξ-controlled string which is marked in the given DES.

Ξ-nonblocking models are desirable if the controller to be generated is in general faster than the plant, i.e. output events are more likely to occur than input events when both are eligible. In the given example, for the event string a*u* to occur, the event *u* has to be generated by the plant before the controller generates the event a for the second time. This is considered to be unlikely to happen. The aim of the Ξ-nonblocking property is to ensure that reaching a marked state is possible by only considering strings in which the controller is not interrupted by the plant. It is not the case that strings in which the controller is interrupted by an input message cannot happen; we cannot and do not want to prevent them from occurring. But we want to ensure that reaching a marked state is possible without using these strings.

If a system is Ξ-nonblocking, it is also nonblocking. This is formalised in the following result.

Proposition 4.4.1
Let D be a DES over the alphabet Σ, and let $\Xi \subseteq \Sigma$.
If D is Ξ-nonblocking, then D is nonblocking.

Proof. This follows directly from definition 4.4.2. □

Ensuring that a DES is Ξ-nonblocking saves an additional test if the system is nonblocking. This is valuable in the later stages of the design process of a controller when implementation aspects of the model are considered.

4.5 Main Result

Now we are able to prove the main result of this chapter. It states that if a DES is Ξ-terminating, Ξ-confluent, and Ξ-nonblocking, then it is nonblocking for every live Ξ-supervisor.

In order to do this, we first provide the following lemma. It states that each Ξ-controlled string in a Ξ-terminating and Ξ-confluent DES has a Nerode equivalent Ξ-controlled string that is accepted by a live Ξ-supervisor.

Lemma 4.5.1
Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, let $\Xi \subseteq \Sigma$, let D be Ξ-terminating and Ξ-confluent, let S be a live Ξ-supervisor for D, and let $s \in \mathcal{L}(S/D)$.
Then, for each Ξ-controlled string t in $\mathcal{L}/s$, there exists a Ξ-controlled string t' in $\mathcal{L}(S/D)/s$ such that $st \equiv st'$ mod Nerode(D).

Proof. Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, let $\Xi \subseteq \Sigma$, let D be Ξ-terminating and Ξ-confluent, let S be a live Ξ-supervisor for D, let $s \in \mathcal{L}(S/D)$, and let $t \in \Sigma^*$ be Ξ-controlled in $\mathcal{L}/s$. We show by induction on the number n of events in t, which are not contained in Ξ, that there exists a Ξ-controlled string t' in $\mathcal{L}(S/D)/s$ such that $st \equiv st'$ mod Nerode(D).

Base case: $n = 0$ and therefore $t \in \Xi^*$. Since t is Ξ-controlled in $\mathcal{L}/s$, we know that t is Ξ-complete in $\mathcal{L}/s$, which implies that st is Ξ-complete in $\mathcal{L}$. By lemma 4.3.1, we obtain a string $t' \in \Xi^*$ such that st' is Ξ-complete in $\mathcal{L}(S/D)$, i.e. t' is Ξ-complete in $\mathcal{L}(S/D)/s$, and $st \equiv st'$ mod Nerode(D). Since t' is Ξ-complete in $\mathcal{L}(S/D)/s$ and $t' \in \Xi^*$, we obtain that t' is Ξ-controlled in $\mathcal{L}(S/D)/s$.

Inductive step: $n \to n+1$. Let t contain $n+1$ events from $\Sigma \backslash \Xi$. Then there exist $a \in \Sigma^*$, $\iota \in \Sigma \backslash \Xi$, and $b \in \Xi^*$ such that $t = a\iota b$, in which the string a contains n events from $\Sigma \backslash \Xi$. Using the inductive assumption, we get a Ξ-controlled string a' in $\mathcal{L}(S/D)/s$ such that $sa \equiv sa'$ mod Nerode(D). Since $sa\iota \in \mathcal{L}$, we obtain $sa'\iota \in \mathcal{L}$. Then $sa'\iota \in \mathcal{L}(S/D)$, since $\iota \in \Sigma \backslash \Xi \subseteq S(sa')$. Furthermore, $sa'\iota b$ is Ξ-complete in $\mathcal{L}$, since $sa\iota b = st$ is Ξ-complete in $\mathcal{L}$. Recall that $b \in \Xi^*$. By lemma 4.3.1, we obtain a string $b' \in \Xi^*$ such that $sa'\iota b'$ is Ξ-complete in $\mathcal{L}(S/D)$ and $sa'\iota b \equiv sa'\iota b'$ mod Nerode(D). Since Nerode equivalence is a right congruence, we have $sa\iota b \equiv sa'\iota b$ mod Nerode(D), thus $sa\iota b \equiv sa'\iota b'$ mod Nerode(D). It suffices to show that $t' := a'\iota b'$ is Ξ-controlled in $\mathcal{L}(S/D)/s$. But this follows from the fact that a' is Ξ-controlled in $\mathcal{L}(S/D)/s$, $b' \in \Xi^*$, and $a'\iota b'$ is Ξ-complete in $\mathcal{L}(S/D)/s$. □

Now, the desired theorem follows easily from the previous lemma.

Theorem 4.5.2
Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, let $\Xi \subseteq \Sigma$, let D be Ξ-terminating, Ξ-confluent, and Ξ-nonblocking.
For each live Ξ-supervisor S for D, we have that S/D is Ξ-nonblocking and therefore nonblocking.

Proof. Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, let $\Xi \subseteq \Sigma$, let D be Ξ-terminating, Ξ-confluent, and Ξ-nonblocking, let S be a live Ξ-supervisor for D, and let $s \in \mathcal{L}(S/D)$. Since D is Ξ-nonblocking, there exists a Ξ-controlled string t in $\mathcal{L}/s$ such that $st \in \mathcal{L}_m$. Using lemma 4.5.1 we obtain a Ξ-controlled string t' in $\mathcal{L}(S/D)/s$ such that $st \equiv st'$ mod Nerode(D). Therefore, since $st \in \mathcal{L}_m$, we also have $st' \in \mathcal{L}_m$. Thus, we obtain $st' \in \mathcal{L}_m \cap \mathcal{L}(S/D) = \mathcal{L}_m(S/D)$. □

Theorem 4.5.2 shows that, to obtain nonblocking controllers, it is sufficient to design a model that is Ξ-terminating, Ξ-confluent, and Ξ-nonblocking. If these three properties are met, every possible live Ξ-supervisor, and thus every possible controller, is nonblocking. By choosing any controller, possibly optimised in some sense, we obtain a suitable controller for controlling the system.

Note that not each DES that is nonblocking for each live Ξ-supervisor is necessarily Ξ-terminating, Ξ-confluent, and Ξ-nonblocking. In fact, there exist discrete event systems that are nonblocking for each live Ξ-supervisor and do not even satisfy one of these properties. The introduced properties reflect qualities required by many practical controllers. Each property on its own is worth to be verified. From this point of view, theorem 4.5.2 can be considered as a nice side-effect of the desired properties.

Chapter 5

Property Checking Algorithms

In chapter 4, Ξ-terminating, Ξ-confluent, and Ξ-nonblocking discrete event systems have been introduced as appropriate models for controllers. Any live Ξ-supervisor, and therefore any controller derived from such a model, is necessarily nonblocking. In this chapter, we address the question how these properties can be checked algorithmically. We develop algorithms that check if given discrete event systems are Ξ-terminating, Ξ-confluent, and Ξ-nonblocking. If one of these properties is not satisfied, a counter-example is computed, which gives a reason why the property does not hold. This counter-example can be used to understand the problem and to refine or fix the model.

In this thesis, deterministic automata are used to model the behaviour of discrete event systems. Real-world discrete event systems are usually too large to be represented by a single automaton. Instead, a set of automata is used, and each automaton describes a certain part of the system. The behaviour of the entire system can be derived by computing the synchronous product of all automata contained in the set. Recall that a deterministic automaton is identified with its represented discrete event system. Thus, we are interested how the synchronous product of deterministic automata can be checked if it is Ξ-terminating, Ξ-confluent, and Ξ-nonblocking.

In section 5.1, state exploration algorithms are presented. These algorithms compute the synchronous product of all automata explicitly and check the property by means of the resultant automaton. Since the size of this automaton may be exponential in the number of constituent automata, this approach is only applicable to quite small systems. Consider n automata, each having m states. The synchronous product of these automata may have m^n reachable states. This is called the *state explosion problem*.

For applications of realistic size, it is practically impossible to construct the composed system. Symbolic model checking [BCM+92, McM93] tries to overcome this problem by using efficient data structures which allow the representation of transition structures in a way dependent on the structure of the represented system rather than on its size. Partial-order reduction algorithms [Val91, God91] consider only a few sequentialisations of concurrent events. They exploit the fact that for many properties the relative order of concurrent events is irrelevant. Incremental algorithms [BMD00] verify properties with respect to subsystems of the overall system and conclude to properties of the entire system.

In section 5.2, we develop approaches for checking a synchronous product of deterministic automata incrementally if it is Ξ-terminating, Ξ-confluent, and Ξ-nonblocking. For each property, an outline of an incremental check based on various theoretical results is given. These algorithms can be improved even more by using symbolic and partial-order approaches.

In the worst case, an incremental algorithm is forced to compute the entire synchronous product of all automata. However, models of real-world systems are usually well-structured and thus enable incremental algorithms to check properties very efficiently.

5.1 State Exploration Algorithms

Consider a discrete event system modelled by a set of deterministic automata. We want to check if this system satisfies certain properties. This can be done using so-called *state exploration algorithms* that first compute the synchronous product of all automata constituting a system and then perform a check by means of the resultant automaton.

State exploration algorithms usually need to explore all reachable states of the synchronous product automaton, which requires that the reachable state space is computed explicitly. This restricts the applicability of these algorithms to small examples. However, the results given in this section provide the basis for the more elaborate incremental algorithms given in section 5.2.

In this section, we assume that the synchronous product of all automata has already been computed. Therefore, we state certain results for a single automaton representing the discrete event system to be checked.

In section 5.1.1, we show that checking a deterministic automaton if it is Ξ-terminating can be performed by computing strongly Ξ-connected components. This can be dealt with using standard graph theoretic algorithms. In section 5.1.2, we show that normal forms of strongly Ξ-connected components can be used to check if a system is Ξ-confluent. The concept of Ξ-coreachability is used in section 5.1.3 to outline an algorithm that checks if a system is Ξ-nonblocking.

5.1.1 Ξ-Termination

In definition 4.2.1 on page 32, we have introduced Ξ-terminating discrete event systems. Now we investigate in a state exploration algorithm that checks if a deterministic automaton is Ξ-terminating. We show that checking a deterministic automaton for Ξ-termination is equivalent to the problem of finding reachable Ξ-loops. Such loops can be find by inspecting strongly Ξ-connected components and self-loops. Standard graph theoretic algorithms can be used to compute strongly Ξ-connected components. If a system is not Ξ-terminating, a counter-example is computed, which gives an explanation why the property does not hold.

A very illustrative way of checking if an automaton is Ξ-terminating is to check for reachable Ξ-loops. A Ξ-loop in an automaton is a path of length at least one with the same origin and end and labelled with output events. If there exists a Ξ-loop, we may visit certain states again and again by following transitions labelled with output events.

Definition 5.1.1 (Ξ-loop)
Let A be an automaton over the alphabet Σ, and let $\Xi \subseteq \Sigma$.
A path π in A is said to be a Ξ-loop *if $s_o(\pi) = s_e(\pi)$ and $L(\pi) \in \Xi^+$.*
A Ξ-loop is reachable *in A if its origin is a reachable state of A.*
If A does not contain any reachable Ξ-loop, A is Ξ-loop free*.*

There exists a reachable Ξ-loop in a deterministic automaton if and only if the automaton is not Ξ-terminating. Thus, checking if an automaton is Ξ-terminating is equivalent to the problem of finding reachable Ξ-loops. This is formally proven in the following result.

Proposition 5.1.1
Let $G = (\Sigma, Q, \delta, q_0, M)$ be a deterministic automaton, and let $\Xi \subseteq \Sigma$.
G is Ξ-loop free if and only if G is Ξ-terminating.

Proof. Let $G = (\Sigma, Q, \delta, q_0, M)$ be a deterministic automaton, and let $\Xi \subseteq \Sigma$.

(Only If) Let G be not Ξ-terminating, i.e. $\mathcal{L}(G)$ is not Ξ-terminating. Then there exists $s \in \mathcal{L}(G)$ such that, for each integer n, there exists $t \in \Xi^*$ such that $|t| \geq n$ and $st \in \mathcal{L}(G)$. In particular, there exists $t \in \Xi^*$ such that $|t| > |Q|$ and $st \in \mathcal{L}(G)$. Consider the path with origin $\delta(q_0, s)$, end $\delta(q_0, st)$, and label t. Since $|t| > |Q|$ there exists a state q which is visited at least twice, i.e. there exists a subpath π such that $s_o(\pi) = s_e(\pi)$ and $L(\pi) \in \Xi^+$. This path is a reachable Ξ-loop in G.

(If) Let π be a reachable Ξ-loop in G. Since π is reachable, there exists a string that leads to the origin of π. More precisely, there exists $s \in \Sigma^*$ such that $\delta(q_0, s) = s_o(\pi)$. Let $t = L(\pi)$. We know that $t \in \Xi^+$. For each integer k, we have $st^k \in \mathcal{L}(G)$, $t^k \in \Xi^*$, and $|t^k| \geq k$. This shows that $\mathcal{L}(G)$ is not Ξ-terminating. $\square$

Reachable Ξ-loops can be identified by finding strongly Ξ-connected components and self-loops labelled with output events. Recall that self-loops are paths of length one with the same origin and end, i.e. transitions with the same origin and target state. Strongly Ξ-connected components are sets of states that are mutually reachable by means of output events.

In order to provide the concept of strongly Ξ-connected components, we first define a relation over states. Two states are strongly Ξ-connected if they can be mutually be reached by means of output events. Recall that $\rightarrow_\Xi$ defined in section 3.1.4 on page 19 describes transitions of an automaton that are labelled with output events. The reachability relation $\xrightarrow{*}_\Xi$ is defined in section 3.1.2 on page 14 and contains pairs of states for which the second state is reachable from the first state by following transitions labelled with output events.

Definition 5.1.2 (strongly Ξ-connected states)
Let $A = (\Sigma, Q, \Delta, J, M)$ be an automaton, and let $\Xi \subseteq \Sigma$.
Two states $p, q \in Q$ are strongly Ξ-connected *in A, written $p \rightleftharpoons_\Xi q$, if $p \xrightarrow{*}_\Xi q$ and $q \xrightarrow{*}_\Xi p$.*

It can be easily shown that this relation is an equivalence relation. The equivalence classes of this equivalence relation are called strongly Ξ-connected components. Such an equivalence class contains states reachable by Ξ-paths from all other states in the equivalence class.

Definition 5.1.3 (strongly Ξ-connected component)
Let A be an automaton over the alphabet Σ, and let $\Xi \subseteq \Sigma$.
The equivalence classes of $\rightleftharpoons_\Xi$ are called the strongly Ξ-connected components *of A.*
A strongly Ξ-connected component is said to be reachable *in A if it contains a state that is reachable in A.*

Note that each state of a reachable strongly Ξ-connected component is reachable itself. Next, we show that if an automaton has a strongly Ξ-connected component with more than one state, then there exists a Ξ-loop in the automaton.

Lemma 5.1.2
Let $A = (\Sigma, Q, \Delta, J, M)$ be an automaton, and let $\Xi \subseteq \Sigma$.
If there exists $p, q \in Q$ such that $p \rightleftharpoons_\Xi q$ and $p \neq q$, then there exists a Ξ-loop in A.

Proof. Let $A = (\Sigma, Q, \Delta, J, M)$ be a deterministic automaton, let $\Xi \subseteq \Sigma$, and let $p, q \in Q$ such that $p \rightleftharpoons_\Xi q$ and $p \neq q$. We are able to construct a Ξ-loop in the following way. Since $p \rightleftharpoons_\Xi q$, there exists a Ξ-path π_1 with origin p and end q and a Ξ-path π_2 with origin q and end p. The path $\pi_1\pi_2$ is a Ξ-path in A with origin p and end p, i.e. $\pi_1\pi_2$ is a Ξ-loop in A. □

If the strongly Ξ-connected component considered in lemma 5.1.2 is reachable, the Ξ-loop constructed in the proof is also reachable. Thus, a deterministic automaton cannot be Ξ-terminating if it has a reachable strongly Ξ-connected component containing more than one state.

Iterative Ξ-pairs are appropriate counter-examples, which can be provided when an automaton is not Ξ-terminating. An iterative Ξ-pair consists of two strings such that the second string can be iterated while staying within the behaviour of the system.

Definition 5.1.4 (iterative Ξ-pair)
Let $G = (\Sigma, Q, \delta, q_0, M)$ be a deterministic automaton, and let $\Xi \subseteq \Sigma$. The pair $(s, t) \in \Sigma^* \times \Xi^+$ is an iterative Ξ-pair *in G if $st^n \in \mathcal{L}(G)$ for each $n \in \mathbb{N}$.*

If there exists a reachable Ξ-loop π in a deterministic automaton, then there exists an iterative Ξ-pair in this automaton. This can be seen as follows. Since π is reachable, there exists a string s that leads to the origin of π. Then we have that $(s, L(\pi))$ is an iterative Ξ-pair in the automaton. The first string shows how the Ξ-loop can be reached. The second string describes the Ξ-loop. This provides illustrative information why a system is not Ξ-terminating.

The results given so far enable us to outline an algorithm that checks if an automaton is Ξ-terminating. Let $G = (\Sigma, Q, \delta, q_0, M)$ be a deterministic automaton, and let $\Xi \subseteq \Sigma$ be the alphabet of output events. By checking the following two conditions, the algorithm is able to determine if G is Ξ-loop free. Furthermore, it can provide an iterative Ξ-pair as a counter-example if the property is not satisfied. Recall that $Q_{\text{reach}}(G)$ denotes the set of all reachable states of G.

(1) There exists $q \in Q_{\text{reach}}(G)$ and $\xi \in \Xi$ such that $\delta(q, \xi)$ is defined and $\delta(q, \xi) = q$.

(2) There exists a strongly Ξ-connected component in G consisting of at least two different reachable states.

With condition (1), reachable self-loops labelled with an output event are identified. If there exists such a self-loop, the automaton is not Ξ-terminating. The path $q \xrightarrow{\xi} q$ is a reachable Ξ-loop in G, and an iterative Ξ-pair can be computed as described above.

Strongly Ξ-connected components are used to check condition (2). There exist algorithms [Tar72, NSS94] that find strongly connected components in a directed graph in linear complexity (with respect to the number of nodes and edges). By lemma 5.1.2, a reachable Ξ-loop exists if there exists a reachable strongly Ξ-connected component containing more than one state. This Ξ-loop can be computed, and an iterative Ξ-pair can be derived as a counter-example.

If neither condition is satisfied, G is Ξ-terminating. This can be seen as follows. If G contains a reachable Ξ-loop, it is either a self-loop, and thus condition (1) is satisfied, or it contains at least two different states. But these states are in the same strongly Ξ-connected component and condition (2) is satisfied. Therefore, G cannot have a reachable Ξ-loop. By proposition 5.1.1, G is Ξ-terminating.

5.1.2 Ξ-Confluence

In definition 4.3.1 on page 33, we have defined Ξ-confluent discrete event systems. In this section, we develop an algorithm which checks whether or not a deterministic automaton is Ξ-confluent. In order to do this, we introduce Ξ-state-confluent deterministic automata and show that a minimum-state[1] deterministic automaton is Ξ-confluent if and only if it is Ξ-state-confluent. Thus, to determine if an automaton is Ξ-confluent, we may also minimise the automaton and check if the minimised automaton is Ξ-state-confluent. Then we show that checking if an automaton is Ξ-state-confluent can be performed by checking if normal forms of reachable strongly Ξ-connected components are unique.

We start by defining Ξ-state-confluent automata. An automaton is Ξ-state-confluent if the relation $\rightarrow_\Xi$ for the reachable subautomaton of the given automaton is confluent. This is the case if for all pairs of Ξ-paths originating from a reachable state of the automaton there exist Ξ-paths which continue the given Ξ-paths to a common state.

Definition 5.1.5 (Ξ-state-confluent automaton)
Let A be an automaton over the alphabet Σ, and let $\Xi \subseteq \Sigma$. A is Ξ-state-confluent *if the relation*

$$\rightarrow_\Xi^{\mathcal{R}(A)} \subseteq Q_{\text{reach}}(A) \times Q_{\text{reach}}(A)$$

is confluent.

If a deterministic automaton is Ξ-state-confluent, then it is Ξ-confluent. Thus, to show that a system is Ξ-confluent, it is sufficient to show that it is Ξ-state-confluent. This is proven in the next lemma.

Lemma 5.1.3
Let G be a deterministic automaton over the alphabet Σ, and let $\Xi \subseteq \Sigma$.
If G is Ξ-state-confluent, G is Ξ-confluent.

Proof. Let $G = (\Sigma, Q, \delta, q_0, M)$ be a deterministic automaton, and let $\Xi \subseteq \Sigma$.

Let G be Ξ-state-confluent, let $s \in \mathcal{L}(G)$, and let $s_1, s_2 \in \Xi^*$ such that $ss_1, ss_2 \in \mathcal{L}(G)$. Let $p = \delta(q_0, s)$, $p_1 = \delta(p, s_1)$, and $p_2 = \delta(p, s_2)$. We have $p, p_1, p_2 \in Q_{\text{reach}}(G)$, $p \overset{*}{\rightarrow}_\Xi p_1$, and $p \overset{*}{\rightarrow}_\Xi p_2$. Since G is Ξ-state-confluent, there exists $q \in Q_{\text{reach}}(G)$ such that $p_1 \overset{*}{\rightarrow}_\Xi q$ and $p_2 \overset{*}{\rightarrow}_\Xi q$, i.e. there exists a Ξ-path π_1 originating in p_1 and ending in q and a Ξ-path π_2 originating in p_2 and ending in q. But then we have $L(\pi_1), L(\pi_2) \in \Xi^*$ and $\delta(p, s_1L(\pi_1)) = \delta(p, s_2L(\pi_2))$, i.e. $ss_1L(\pi_1) \equiv ss_2L(\pi_2) \bmod \text{Nerode}(G)$. Thus, G is Ξ-confluent. $\square$

[1] Recall that the minimum-state automaton has a minimal number of states: there does not exist an automaton with fewer states accepting the same languages. Also recall that all deterministic minimum-state automata are isomorphic.

If a deterministic automaton is Ξ-confluent, then it is not necessarily Ξ-state-confluent, since Myhill-Nerode equivalent strings must not end in the same state. However, if we consider a minimum-state automaton, Ξ-confluence implies Ξ-state-confluence. Thus, to conclude that an automaton is not Ξ-confluent from the fact that it is not Ξ-state-confluent, we must ensure that it is a minimum-state automaton.

Lemma 5.1.4
Let G be a minimum-state deterministic automaton over the alphabet Σ, and let $\Xi \subseteq \Sigma$.
If G is Ξ-confluent, then G is Ξ-state-confluent.

Proof. Let $G = (\Sigma, Q, \delta, q_0, M)$ be a minimum-state deterministic automaton, and let $\Xi \subseteq \Sigma$.

Let G be Ξ-confluent, and let $p, p_1, p_2 \in Q$ be reachable such that $p \xrightarrow{*}_{\Xi} p_1$ and $p \xrightarrow{*}_{\Xi} p_2$. Then there exists $s \in \mathcal{L}(G)$ and $s_1, s_2 \in \Xi^*$ such that $p = \delta(q_0, s)$, $p_1 = \delta(p, s_1)$, and $p_2 = \delta(p, s_2)$, i.e. $ss_1, ss_2 \in \mathcal{L}(G)$. Since G is Ξ-confluent, there exist $t_1, t_2 \in \Xi^*$ such that $ss_1t_1, ss_2t_2 \in \mathcal{L}(G)$ and $ss_1t_1 \equiv ss_2t_2 \bmod \mathrm{Nerode}(G)$. Since G is a minimum-state automaton, there exists $q \in Q_{\mathrm{reach}}(G)$ such that $p_1 \xrightarrow{*}_{\Xi} q$ and $p_2 \xrightarrow{*}_{\Xi} q$. Thus, G is Ξ-state-confluent. $\square$

Next we show that a deterministic automaton is Ξ-state-confluent if and only if the quotient automaton with respect to $\rightleftharpoons_{\Xi}$ is Ξ-state-confluent. But first we proof the following lemma, which states that each path in the quotient automaton has a corresponding path in the original automaton.

Lemma 5.1.5
Let A be an automaton over the alphabet Σ, and let $\Xi' \subseteq \Xi \subseteq \Sigma$. For each Ξ'-path π in $A/\rightleftharpoons_{\Xi}$ and for each $p \in s_o(\pi)$ and $q \in s_e(\pi)$ there exists a Ξ'-path in A with origin p and end q.

Proof. Let A be an automaton over the alphabet Σ, and let $\Xi' \subseteq \Xi \subseteq \Sigma$. Let π be a Ξ'-path in $A/\rightleftharpoons_{\Xi}$, let $p \in s_o(\pi)$, and let $q \in s_e(\pi)$. We show by induction on the length n of π that there exists a Ξ'-path in A with origin p and end q.

Base case ($n = 0$). Then we have $\pi = C$ for some $C \in Q_{A/\rightleftharpoons_{\Xi}}$ and $p, q \in C$. Since $p \rightleftharpoons_{\Xi} q$, there exists a Ξ-path in A with origin p and end q. This is also a Ξ'-path in A.

Inductive step ($n > 0$). Then we have $\pi = C_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_{n+1}} C_{n+1}$, $p \in C_0$, and $q \in C_{n+1}$. Since $(C_0, \sigma_1, C_1) \in \Delta_{A/\rightleftharpoons_{\Xi}}$, there exist $p_0 \in C_0$ and $p_1 \in C_1$ such that $(p_0, \sigma_1, p_1) \in \Delta_A$. Since $p_0 \rightleftharpoons_{\Xi} p$, there exist a Ξ-path π_1 with origin p and end p_0. By the inductive assumption, there exists a Ξ'-path π_2 with origin p_1 and end q. Then we have that $\pi_1 \, (p_0 \xrightarrow{\sigma_1} p_1) \, \pi_2$ is a Ξ'-path in A with origin p and end q. $\square$

Now we can show that an automaton is Ξ-state-confluent if and only if its quotient automaton with respect to $\rightleftharpoons_{\Xi}$ is Ξ-state-confluent.

Lemma 5.1.6
Let A be an automaton over the alphabet Σ, and let $\Xi \subseteq \Sigma$.
A is Ξ-state-confluent if and only if $A/\rightleftharpoons_\Xi$ is Ξ-state-confluent.

Proof. Let A be an automaton over the alphabet Σ, let $\Xi \subseteq \Sigma$, and let $\bar{A} = (A/\rightleftharpoons_\Xi)$.

(If) Let A be Ξ-state-confluent. We want to show that $\bar{A}$ is Ξ-state-confluent. Let $C, C_1, C_2 \in Q_{\text{reach}}(\bar{A})$ such that $C \xrightarrow{*}{}_{\Xi}^{\mathcal{R}(\bar{A})} C_1$ and $C \xrightarrow{*}{}_{\Xi}^{\mathcal{R}(\bar{A})} C_2$. Let $p \in C$, $p_1 \in C_1$, and $p_2 \in C_2$. By lemma 5.1.5, we have $p, p_1, p_2 \in Q_{\text{reach}}(A)$, $p \xrightarrow{*}{}_{\Xi}^{\mathcal{R}(A)} p_1$, and $p \xrightarrow{*}{}_{\Xi}^{\mathcal{R}(A)} p_2$. Since A is Ξ-state-confluent, there exists $q \in Q_{\text{reach}}(A)$ such that $p_1 \xrightarrow{*}{}_{\Xi}^{\mathcal{R}(A)} q$ and $p_2 \xrightarrow{*}{}_{\Xi}^{\mathcal{R}(A)} q$. By lemma 3.1.5, $[p_1] \xrightarrow{*}{}_{\Xi}^{\mathcal{R}(\bar{A})} [q]$ and $[p_2] \xrightarrow{*}{}_{\Xi}^{\mathcal{R}(\bar{A})} [q]$. This proves that $\bar{A}$ is Ξ-state-confluent.

(Only If) Let $\bar{A}$ be Ξ-state-confluent. We want to show that A is Ξ-state-confluent. Let $p, p_1, p_2 \in Q_{\text{reach}}(A)$ such that $p \xrightarrow{*}{}_{\Xi}^{\mathcal{R}(A)} p_1$ and $p \xrightarrow{*}{}_{\Xi}^{\mathcal{R}(A)} p_2$. By lemma 5.1.5, we have $[p], [p_1], [p_2] \in Q_{\text{reach}}(A)$, $[p] \xrightarrow{*}{}_{\Xi}^{\mathcal{R}(\bar{A})} [p_1]$ and $[p] \xrightarrow{*}{}_{\Xi}^{\mathcal{R}(\bar{A})} [p_2]$. Since $\bar{A}$ is Ξ-state-confluent, there exist $C \in Q_{\text{reach}}(\bar{A})$ such that $[p_1] \xrightarrow{*}{}_{\Xi}^{\mathcal{R}(\bar{A})} C$ and $[p_2] \xrightarrow{*}{}_{\Xi}^{\mathcal{R}(\bar{A})} C$. Let $q \in C$. By lemma 5.1.5, $p_1 \xrightarrow{*}{}_{\Xi}^{\mathcal{R}(A)} q$ and $p_2 \xrightarrow{*}{}_{\Xi}^{\mathcal{R}(A)} q$. This proves that A is Ξ-state-confluent. $\square$

When considering a Ξ-terminating system, checking for Ξ-state-confluence and checking if there exist unique normal forms gives the same result. If the automaton to be checked is not Ξ-terminating, we can consider its quotient automaton with respect to $\rightleftharpoons_\Xi$. This automaton does not contain Ξ-loops containing more than one state. Note that self-loops labelled with output events may be still contained. Thus, checking an automaton for Ξ-state-confluence may be performed by computing the normal forms of strongly Ξ-connected components. This is shown in the following.

Lemma 5.1.7
Let A be an automaton over the alphabet Σ, and let $\Xi \subseteq \Sigma$. Furthermore, let $\rightsquigarrow \subseteq Q_{\text{reach}}(A/\rightleftharpoons_\Xi) \times Q_{\text{reach}}(A/\rightleftharpoons_\Xi)$ defined as $C \rightsquigarrow C'$ if $C \rightarrow_{\Xi}^{A/\rightleftharpoons_\Xi} C'$ and $C \neq C'$.
Then the following two conditions are equivalent.

- *A is Ξ-state-confluent.*
- *Each reachable state in $A/\rightleftharpoons_\Xi$ has a unique normal form with respect to $\rightsquigarrow$.*

Proof. Let A be an automaton over the alphabet Σ, and let $\Xi \subseteq \Sigma$. Furthermore, let $\rightsquigarrow \subseteq Q_{\text{reach}}(A/\rightleftharpoons_\Xi) \times Q_{\text{reach}}(A/\rightleftharpoons_\Xi)$ defined as $C \rightsquigarrow C'$ if $C \rightarrow_{\Xi}^{A/\rightleftharpoons_\Xi} C'$ and $C \neq C'$.

First we claim that $\rightsquigarrow$ is terminating. Assume there exists an infinite chain $C_0 \rightsquigarrow C_1 \cdots$. Since A has a finite number of states, $A/\rightleftharpoons_\Xi$ has also a finite number of states. Thus, there must exist $i, j \in \mathbb{N}$ such that $i < j$ and $C_i = C_j$.

Let $p_i \in C_i$ and $p_{i+1} \in C_{i+1}$. Since $C_i \rightsquigarrow C_{i+1}$, we have $C_i \rightarrow_{\Xi}^{A/\rightleftharpoons_\Xi} C_{i+1}$. By lemma 5.1.5, there exists a Ξ-path in A with origin p and end q. Since $C_{i+1} \rightsquigarrow \cdots \rightsquigarrow C_j$, we obtain a Ξ-path in A with origin q and end p. Therefore, we have $p \rightleftharpoons_\Xi q$, i.e. $C_i = C_{i+1}$ which contradicts the definition of $\rightsquigarrow$.

By lemma 5.1.6, we obtain that A is Ξ-state-confluent if and only if $A/\rightleftharpoons_\Xi$ is Ξ-state-confluent, i.e. $\rightarrow_{\Xi}^{\mathcal{R}(A/\rightleftharpoons_\Xi)}$ is confluent. It can easily be verified that $\rightarrow_{\Xi}^{\mathcal{R}(A/\rightleftharpoons_\Xi)}$ is confluent if and only if $\rightsquigarrow$ is confluent. Furthermore, $\rightsquigarrow$ is terminating. By lemma 3.1.1, $\rightsquigarrow$ is terminating and confluent if and only if each reachable state in $A/\rightleftharpoons_\Xi$ has a unique normal form with respect to $\rightsquigarrow$. $\square$

Corollary 5.1.8

Let G be a minimum-state deterministic automaton over the alphabet Σ, and let $\Xi \subseteq \Sigma$. Furthermore, let $\rightsquigarrow \subseteq Q_{\text{reach}}(G/\rightleftharpoons_\Xi) \times Q_{\text{reach}}(G/\rightleftharpoons_\Xi)$ defined as $C \rightsquigarrow C'$ if $C \rightarrow_{\Xi}^{G/\rightleftharpoons_\Xi} C'$ and $C \neq C'$.
Then the following two conditions are equivalent.

- *G is Ξ-confluent.*
- *Each reachable state in $G/\rightleftharpoons_\Xi$ has a unique normal form with respect to $\rightsquigarrow$.*

Corollary 5.1.8 enables us to outline an algorithm that checks whether or not a deterministic automaton is Ξ-confluent. Consider a deterministic automaton G over the alphabet Σ and an alphabet $\Xi \subseteq \Sigma$. In a first step, automaton G is minimised. Secondly, the reachable strongly Ξ-connected components of the minimised automaton are computed. Note that the latter step is not necessary if G is Ξ-terminating. In this case, each state forms its own strongly Ξ-connected component. Thirdly, the quotient automaton $\bar{G} = G/\rightleftharpoons_\Xi$ is computed. Let $\rightsquigarrow$ be defined as in corollary 5.1.8. For each $C \in Q_{\text{reach}}(\bar{G})$ the algorithm computes the normal forms $\mathcal{N}(C)$ with respect to $\rightsquigarrow$:

$$\mathcal{N}(C) = \begin{cases} C, & \text{if there does not exist } C' \in Q_{\text{reach}}(\bar{G}) \text{ such that } C \rightsquigarrow C'; \\ \bigcup \{ \mathcal{N}(C') \mid C' \in Q_{\text{reach}}(\bar{G}) \text{ such that } C \rightsquigarrow C' \}, & \text{otherwise.} \end{cases}$$

This computation evaluates the successors of C, i.e. strongly Ξ-connected components that can be reached by means of an output event. If C does not have a successor expect itself, C is the normal form of C. Otherwise, the set of normal forms of C is the union of all Ξ-normal forms of all successors. Since $\rightsquigarrow$ is terminating, this computation stops after a finite number of steps. A hash table can be used to avoid computing normal forms of a strongly Ξ-connected component more than once. If a reachable strongly Ξ-connected component has more than one normal form, G is not Ξ-confluent; otherwise G is Ξ-confluent.

A minimisation algorithm of complexity $n \log n$ for a deterministic finite automaton with n states is presented in Hopcroft [Hop71]. Note that lemma 5.1.3 does not require G to be a minimum-state automaton. Therefore, a deterministic automaton is Ξ-confluent if it is Ξ-state-confluent, even though it does not have a minimal number of states. This observation enables us to modify the above algorithm to compute the normal forms of strongly Ξ-connected components of the original automaton; minimisation is only performed in case that Ξ-state-confluence cannot be shown.

If a deterministic automaton G is not Ξ-confluent, the algorithm finds a strongly Ξ-connected component with at least two normal forms. Now consider a state contained in this strongly Ξ-connected component. There exist Ξ-paths that originate from this state and end in different normal forms. These paths cannot be continued to reach a common target state by means of output events only. These Ξ-paths, or the labels of these Ξ-paths, together with the information of how to reach its origin can be used as a counter-example and provide an explanation why the property does not hold.

5.1.3 Ξ-Nonblocking

In definition 4.4.2 on page 36, we have introduced Ξ-nonblocking discrete event systems. Now we investigate in an algorithm that checks whether or not a deterministic automaton is Ξ-nonblocking. The algorithm checks if each reachable state of a deterministic automaton is Ξ-coreachable to determine if it is Ξ-nonblocking.

The algorithm given herein works similar to the algorithm that checks if an automaton is nonblocking by checking if each reachable state is coreachable. A state is said to be coreachable if it is origin of a path that ends in a marked state. If there exists a reachable state which is not coreachable, the automaton is blocking: it is possible to reach a state from which a marked state cannot be reached. If each reachable state is coreachable, the automaton is nonblocking: after all behaviours it is possible to reach a marked state.

Next, we introduce the concept of Ξ-coreachability and formulate the algorithm by means of the new concept. We start by introducing Ξ-controlled paths.

Definition 5.1.6 (Ξ-controlled path)
Let A be an automaton over the alphabet Σ, let $\pi = p_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} p_n$ be a path in A, and let $\Xi \subseteq \Sigma$. The path π is said to be

- Ξ-preferring *in A if for all $i \in \{0, \ldots, n-1\}$ such that $\sigma_{i+1} \notin \Xi$ we have $\textsc{Elig}_A(p_i) \cap \Xi = \emptyset$.*
- Ξ-complete *in A if $\textsc{Elig}_A(p_n) \cap \Xi = \emptyset$.*
- Ξ-controlled *in A if it is Ξ-preferring and Ξ-complete in A.*

The definition of Ξ-controlled paths is very similar to the definition of Ξ-controlled words defined in definition 4.4.1 on page 35. For deterministic automata, these definitions are equivalent, i.e. if there is a Ξ-controlled string in the language of the automaton, then there exists a Ξ-controlled path in the automaton and vice versa.

Lemma 5.1.9
Let $G = (\Sigma, Q, \delta, q_0, M)$ be a deterministic automaton, let $\Xi \subseteq \Sigma$, and let $s \in \mathcal{L}(G)$.
The state $\delta(q_0, s)$ is Ξ-coreachable if and only if there exists a Ξ-controlled string t in $\mathcal{L}(G)/s$ such that $st \in \mathcal{L}_m(G)$.

Proof. Let $G = (\Sigma, Q, \delta, q_0, M)$ be a deterministic automaton, let $\Xi \subseteq \Sigma$, and let $s \in \mathcal{L}(G)$.

(If) Let $\delta(q_0, s)$ be Ξ-coreachable in G, i.e. there exists a Ξ-controlled and marked path π in G with origin $\delta(q_0, s)$. Consider $t = L(\pi)$. Then we have that $t \in \mathcal{L}(G)/s$ such that $st \in \mathcal{L}_m(G)$. It suffices to show that t is Ξ-controlled in $\mathcal{L}(G)/s$. Let $a \sqsubseteq t$ and $\iota \in \Sigma\backslash\Xi$ such that $a\iota \sqsubseteq t$. Then $\Xi \cap \textsc{Elig}_{\mathcal{L}(G)/s}(a) = \Xi \cap \textsc{Elig}_G(\delta(q_0, sa)) = \emptyset$, since π is Ξ-preferring. Furthermore, $\Xi \cap \textsc{Elig}_{\mathcal{L}(G)/s}(t) = \textsc{Elig}_G(\delta(q_0, st)) = \emptyset$, since π is Ξ-complete. Therefore, t is Ξ-controlled in $\mathcal{L}(G)/s$.

(Only If) Let t be a Ξ-controlled string in $\mathcal{L}(G)/s$ such that $st \in \mathcal{L}_m(G)$. There exists a unique path π with origin $\delta(q_0, s)$, end $\delta(q_0, st)$, and label t, since G is deterministic. Since $st \in \mathcal{L}_m(G)$ we have $\delta(q_0, st) \in M$, i.e. π is marked. It suffices to show that π is Ξ-controlled in G. Let $t = \sigma_1 \cdots \sigma_n$ such that $\sigma_i \in \Sigma$ for all $i \in \{1, \ldots, n\}$. Let $i \in \{0, \ldots, n-1\}$ such that $\sigma_{i+1} \notin \Xi$. Then we have $\Xi \cap \textsc{Elig}_G(\delta(q_0, s\sigma_1 \cdots \sigma_i)) = \Xi \cap \textsc{Elig}_{\mathcal{L}(G)/s}(\sigma_1 \cdots \sigma_i) = \emptyset$, since t is Ξ-preferring in $\mathcal{L}(G)/s$. Furthermore, $\Xi \cap \textsc{Elig}_G(\delta(q_0, st)) = \Xi \cap \textsc{Elig}_{\mathcal{L}(G)/s}(t) = \emptyset$, since t is Ξ-complete in $\mathcal{L}(G)/s$. Therefore, π is Ξ-controlled in G. This shows that $\delta(q_0, s)$ is Ξ-coreachable in G. □

Now we are able to introduce the concept of Ξ-coreachability. A state is Ξ-coreachable if it is origin of a path that prefers output events and ends in a marked state.

Definition 5.1.7 (Ξ-coreachable state)
Let $A = (\Sigma, Q, \Delta, J, M)$ be an automaton, and let $\Xi \subseteq \Sigma$.
A state $q \in Q$ is said to be Ξ-coreachable *if there exists a Ξ-controlled and marked path in A with origin q.*

The checking algorithm given below is based on the following result, which states that a deterministic automaton is Ξ-nonblocking if and only if all reachable states are Ξ-coreachable.

Proposition 5.1.10
Let G be a deterministic automaton over the alphabet Σ, and let $\Xi \subseteq \Sigma$.
The following two conditions are equivalent.

- *Each reachable state of G is Ξ-coreachable.*
- *G is Ξ-nonblocking.*

Proof. Let $G = (\Sigma, Q, \delta, q_0, M)$ be a deterministic automaton over the alphabet Σ, and let $\Xi \subseteq \Sigma$.

First, let each reachable state of G be Ξ-coreachable, and let $s \in \mathcal{L}(G)$. Then, $\delta(q_0, s)$ is reachable and, by assumption, Ξ-coreachable. By lemma 5.1.9, there exists a Ξ-controlled string t in $\mathcal{L}(G)$ such that $st \in \mathcal{L}_m(G)$. Thus, G is Ξ-nonblocking.

Next, let G be Ξ-nonblocking, and let $q \in Q$ be reachable. Then, there exists $s \in \mathcal{L}(G)$ such that $q = \delta(q_0, s)$. Since G is Ξ-nonblocking, there exists a Ξ-controlled string $t \in \mathcal{L}(G)/s$ such that $st \in \mathcal{L}_m(G)$ By lemma 5.1.9, $\delta(q_0, s) = q$ is Ξ-coreachable. □

Applying this result, we obtain the following algorithm. Assume that a deterministic automaton $G = (\Sigma, Q, \delta, q_0, M)$ and an alphabet $\Xi \subseteq \Sigma$ are given. Compute the set $Q_{\text{reach}}(G)$ of all reachable states of G and the set $Q_{\text{coreach}}(G)$ of states that are Ξ-coreachable. This latter set can be computed by means of a fix-point iteration:

$$\begin{aligned}
Q_0 &= \{\, q_m \in M \mid \textsc{Elig}_G(q_m) \cap \Xi = \emptyset \,\} \\
Q_{n+1} &= Q_n \cup \{\, p \in Q \mid p \rightarrow_\Xi q \text{ for some } q \in Q_n \,\} \\
&\qquad \cup \{\, p \in Q \mid p \rightarrow q \text{ for some } q \in Q_n \text{ and } \textsc{Elig}_G(p) \cap \Xi = \emptyset \,\}
\end{aligned}$$

Starting with the set of all marked states in which no output events are eligible, we increase the set by adding all predecessors with respect to output events and all predecessors with respect to input events in which no output events are eligible. Now we have that

$$Q_{\text{coreach}}(G) = \bigcup_{i \in \mathbb{N}} Q_i.$$

Since G has a finite number of states, this fix-point iteration can be computed in a finite number of steps. If

$$Q_{\text{coreach}}(G) \subseteq Q_{\text{reach}}(G),$$

then G is Ξ-nonblocking; otherwise it is Ξ-blocking. In the latter case, the algorithm finds a reachable state from which no marked state can be reached by using a Ξ-controlled path. A sequence of events that leads to this state can be used as a counter-example, pointing exactly to the problematic state.

5.2 Incremental Algorithms

Complex discrete event systems are usually modelled by a set of deterministic automata in a way that each automaton describes a certain part of the system. These automata have to be combined by using the synchronous product operation to obtain the entire system. We have seen that the size of the product automaton can be exponential in the number of constituent automata. Therefore, the product automaton cannot be computed explicitly for many real-world models, and state exploration algorithms cannot be used.

This section considers the design of incremental algorithms, which are able to check systems of realistic size. These algorithms avoid considering the synchronous product of the entire system by exploiting the modular structure of the given systems. The algorithms that check for Ξ-termination and Ξ-confluence conclude from properties of subsystems, i.e. the synchronous product of only a few automata, to properties of the entire system. The incremental algorithm that checks if a system is Ξ-nonblocking computes the synchronous product incrementally and simplifies intermediate results.

In the framework of this thesis, the introduced properties belong closely together. A model must be shown to be Ξ-terminating, Ξ-confluent, and Ξ-nonblocking to ensure that each derived controller is nonblocking. Furthermore, it is natural to first check if a system is Ξ-terminating, then check if it is Ξ-confluent, and finally if it is Ξ-nonblocking, because this order corresponds to the increasing difficulty in understanding the counter-examples and fixing the model. Thus, it does not restrict the applicability of the checking algorithms to assume that the system is already shown to be Ξ-terminating when checking for Ξ-confluence, and Ξ-terminating and Ξ-confluent when checking if it is Ξ-nonblocking. By exploiting these facts, we obtain more efficient incremental checking algorithms. The incremental algorithms proposed in this thesis are based on each other: the incremental Ξ-confluence checking algorithm requires a Ξ-terminating system to provide a correct result; the incremental algorithm that checks if a system is Ξ-nonblocking assumes that the given system is Ξ-terminating and Ξ-confluent.

In section 5.2.1, we start with an introduction to the principle of incremental property checking. Demonstrative examples of incremental checking algorithms are given. Section 5.2.2 considers the design of an incremental algorithm that checks if a discrete event system is Ξ-terminating. Section 5.2.3 introduces concepts for an incremental Ξ-confluence check. In section 5.2.4, results are given which show how a system can be incrementally checked if it is Ξ-nonblocking.

5.2.1 Introduction

Incremental algorithms use properties of subsystems to conclude properties of the entire system. This idea is now illustrated by means of two examples.

Language inclusion can be easily checked using an incremental algorithm as described in [BMD00]. Consider a finite index set I and a system $G = \|_{i\in I} G_i$ composed of deterministic automata. We want to know if the behaviour of the system is a subset of a given language $\mathcal{L}$. More formally, we want to verify if

$$\mathcal{L}(G) \subseteq \mathcal{L}\,. \tag{5.1}$$

Synchronous product composition of automata always restricts and never enlarges the accepted languages. For each $J \subseteq I$, we have $\mathcal{L}\left(\|_{i\in I} G_i\right) \subseteq \mathcal{L}\left(\|_{j\in J} G_J\right)$. Consequently, if the behaviour of a subsystem is contained in $\mathcal{L}$, the behaviour of the entire system is also contained in $\mathcal{L}$. Therefore, if we find a subsystem, i.e. an index set $J \subseteq I$ such that

$$\mathcal{L}\left(\|_{j\in J} G_j\right) \subseteq \mathcal{L}\,, \tag{5.2}$$

the language of the entire system is also contained in $\mathcal{L}$, i.e. condition (5.1) is satisfied. In order to show condition (5.1), it suffices to show condition (5.2).

In order to compute such a subsystem or, equivalently, such an index set as given in condition (5.2), we perform certain verification steps and increment subsystems as follows. For an index set $J \subseteq I$, check if $\mathcal{L}\left(\|_{j\in J} G_j\right) \subseteq \mathcal{L}$. If this is the case, then the property holds for the subsystem and therefore for the entire system. Otherwise, a counter-example, i.e. a string $s \in \mathcal{L}\left(\|_{j\in J} G_j\right)$ such that $s \notin \mathcal{L}$, can be obtained. If all automata of the system accept this counter-example, the entire system also accepts s. More precisely, if $s \in \mathcal{L}\left(G_i\right)$, for each $i \in I$, we have $s \in \mathcal{L}\left(G\right)$. But then we know that $\mathcal{L}\left(G\right) \nsubseteq \mathcal{L}$, i.e. condition (5.1) is not satisfied. If there exists an automaton that does not accept s, i.e. there exists $i \in I$ such that $s \notin G_i$, we increment J by i, i.e. $J' = J \cup \{i\}$. The new subsystem $\|_{j\in J'} G_j$ does not accept s, since $s \notin \mathcal{L}\left(G_i\right)$ and $\mathcal{L}\left(\|_{j\in J'} G_j\right) \subseteq \mathcal{L}\left(G_i\right)$. In the next verification step, we consider J'. Either we prove the property for this new subsystem or obtain another counter-example and proceed as explained by further incrementing the considered subsystem. Note that the incrementing process can be started with the empty index set or, equivalently, with no subsystem. Several heuristics can be used to increment a subsystem, since often several automata do not accept a counter-example and could be added.

As a second example, we consider the problem of checking if a pair of events is *commuting* for a synchronous product of several deterministic automata. The incremental algorithm for this property does not only help to understand the ideas of incremental algorithms; it can also be used as a part of the incremental Ξ-confluence checking procedure presented in section 5.2.3.

A commuting pair of events can be executed in any order when both events are eligible. Furthermore, the target state does not depend on the order. Starting from a common state, both execution sequences lead to the same state.

Definition 5.2.1 (commuting event pair)
Let $G = (\Sigma, Q, \delta, q_0, M)$ be a deterministic automaton, and let $q \in Q$.

A pair $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ is said to be commuting in state q *for G if $\sigma_1, \sigma_2 \in$ $\text{E}\textsc{lig}_G(q)$ implies that $\delta(q, \sigma_1\sigma_2)$ and $\delta(q, \sigma_2\sigma_1)$ are both defined and $\delta(q, \sigma_1\sigma_2) = \delta(q, \sigma_2\sigma_1)$.*

The pair $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ is said to be commuting for automaton G *if (σ_1, σ_2) is commuting in q, for each $q \in Q_{\text{reach}}(G)$.*

If a pair of events is not commuting for an automaton, there exists a reachable state in which the event pair is not commuting. A string to this state (starting from the initial state) can be considered as a counter-example. It provides an explanation—a state in which the pair is not commuting and how this state can be reached—why the property does not hold.

Definition 5.2.2 (commuting counter-example)
Let $G = (\Sigma, Q, \delta, q_0, M)$ be a deterministic automaton, and let $\sigma_1, \sigma_2 \in \Sigma$. A commuting counter-example *for (σ_1, σ_2) in G is a string $s \in \mathcal{L}(G)$ such that (σ_1, σ_2) is not commuting in $\delta(q_0, s)$.*

Note that a commuting counter-example $s \in \Sigma^*$ for an event pair (σ_1, σ_2) is *accepted* by an automaton G if $s\sigma_1 \in \mathcal{L}(G)$ and $s\sigma_2 \in \mathcal{L}(G)$. It is not sufficient to check if $s \in \mathcal{L}(G)$. Note further that a pair of events is commuting for a deterministic automaton if and only if there exists a commuting counter-example for this pair in the automaton.

Now we show under which conditions we can say that an event pair is commuting for a system composed of several deterministic automata—this is the case if the pair is commuting for all subsystems. In contrast to language inclusion, it is not sufficient to find a subsystem that satisfies the desired property.

Lemma 5.2.1
Let G_1 and G_2 be deterministic automata over the same alphabet Σ, and let $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ be commuting for G_1.
If (σ_1, σ_2) is also commuting for G_2, then (σ_1, σ_2) is commuting for $G_1 \,\|\, G_2$.

Proof. Let G_1 and G_2 be deterministic automata over the same alphabet Σ, let $G = G_1 \,\|\, G_2$, and let $\sigma_1, \sigma_2 \in \Sigma$ such that (σ_1, σ_2) is commuting for G_1 and commuting for G_2. Let $o = (p_1, p_2) \in Q_G$ such that $\sigma_1, \sigma_2 \in \textsc{Elig}_G(o)$. For $i \in \{1, 2\}$, we have $\sigma_1, \sigma_2 \in \textsc{Elig}_{G_i}(p_i)$. Since (σ_1, σ_2) is commuting for G_i, $\delta_{G_i}(p_i, \sigma_1\sigma_2)$ and $\delta_{G_i}(p_i, \sigma_2\sigma_1)$ are defined and $\delta_{G_i}(p_i, \sigma_1\sigma_2) = \delta_{G_i}(p_i, \sigma_2\sigma_1)$ for each $i \in \{1, 2\}$. Thus, we have that $\delta_G(o, \sigma_1\sigma_2)$ and $\delta_G(o, \sigma_2\sigma_1)$ are defined and $\delta_G(o, \sigma_1\sigma_2) = \delta_G(o, \sigma_2\sigma_1)$. Therefore, (σ_1, σ_2) is commuting for G. □

Consider a system composed of deterministic automata, i.e. $G = \|_{i \in I} G_i$, for some finite index set I. By lemma 5.2.1, a pair of events is commuting for the system G if this pair is commuting for each G_i, $i \in I$. But what can we do if the pair is not commuting for each original automaton G_i, $i \in I$? In this case, we search for automata G_j, $j \in J$, such that

1. the event pair is commuting for each G_j, $j \in J$, and

2. the subsystems constitute the entire system; that is, $\bigcup_{j \in J} G_j = G$.

If this succeeds, we can conclude by lemma 5.2.1 that the event pair is commuting for G. Finding such subsystems can be done similar to the language inclusion checking algorithm given above.

Before providing an outline for an incremental algorithm, we consider the case that the property does not hold for the composed system. We are interested how a counter-example can be computed without building the synchronous product of all automata. The following lemma provides the desired result. If there is a commuting counter-example that is accepted by all subsystems, then this is a commuting counter-example for the entire system.

Lemma 5.2.2
Let G_1 and G_2 be deterministic automata over the same alphabet Σ, let $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ be a pair of events, and let s be a commuting counter-example for (σ_1, σ_2) in G_1 such that $s\sigma_1, s\sigma_2 \in \mathcal{L}(G_2)$.
Then s is a commuting counter-example for (σ_1, σ_2) in $G_1 \parallel G_2$.

Proof. Let $G_1 = (\Sigma, Q_1, \delta_1, q_{0,1}, M_1)$ and $G_2 = (\Sigma, \delta_2, q_{0,2}, M_2)$ be deterministic automata, let $G = G_1 \parallel G_2 = (\Sigma, Q, \delta, q_0, M)$, let $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ be a pair of events, and let s be a commuting counter-example for (σ_1, σ_2) in G_1 such that $s\sigma_1, s\sigma_2 \in \mathcal{L}(G_2)$. Since s is a commuting counter-example for (σ_1, σ_2) in G_1, we have $s\sigma_1, s\sigma_2 \in \mathcal{L}(G_1)$, and either $s\sigma_1\sigma_2 \notin \mathcal{L}(G_1)$, $s\sigma_2\sigma_1 \notin \mathcal{L}(G_1)$, or $\delta_1(q_{0,1}, s\sigma_1\sigma_2) \neq \delta_1(q_{0,1}, s\sigma_2\sigma_1)$. Thus, we have $s\sigma_1, s\sigma_2 \in \mathcal{L}(G)$, and either $s\sigma_1\sigma_2 \notin \mathcal{L}(G)$, $s\sigma_2\sigma_1 \notin \mathcal{L}(G)$, or $\delta(q_0, s\sigma_1\sigma_2) \neq \delta(q_0, s\sigma_2\sigma_1)$. Therefore, s is a commuting counter-example for (σ_1, σ_2) in G. ☐

Based on the results provided so far, an incremental algorithm could work as follows. Consider a system $G = \|_{i \in I} G_i$, for some finite index set I, composed of deterministic automata defined over the alphabet Σ. Let $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ be the pair of events for which we want to know if it is commuting for G. By lemma 5.2.1, we need to find commuting subsystems that constitute the entire system.

If, for each $i \in I$, the event pair is commuting for G_i, then the pair is also commuting for G. Otherwise, consider the set of automata for which the event pair is not commuting. Note that only automata for which the events σ_1 and σ_2 are relevant are contained in this set. Now we increment each of these automata with other automata by synchronous product composition until we either get a counter-example that is accepted by the entire system or we find a subsystem that is commuting for the given event pair.

Incrementing subsystems is similar to the incremental language inclusion algorithm. For an index set $J \subseteq I$, we check if the event pair is commuting for $\|_{j \in J} G_j$. If this is the case, the property holds for this subsystem, and all constituent automata are successfully checked. Otherwise, a counter-example is computed. If this counter-example is accepted by all automata of the system, we have found a counter-example for the entire system, and the event pair is not commuting for G. If there exists an automaton that does not accept the counter-example, we increment J by the index of this automaton and start a new verification step using the new index set.

5.2.2 Ξ-Termination

In the following, we develop an incremental algorithm for checking whether or not a discrete event system, modelled as the synchronous product of several deterministic automata, is Ξ-terminating.

In section 5.1.1, we proved that checking if a deterministic automaton is Ξ-terminating is equivalent to finding self-loops labelled with output events or strongly Ξ-connected components containing more than one state. Furthermore, we presented an algorithm that checks if a given automaton is Ξ-terminating by exploring the reachable state space of the automaton. This algorithm also computes a counter-example, i.e. an iterative Ξ-pair, if the property is not satisfied. The incremental algorithm developed here uses the state exploration algorithm to check subsystems of the entire system. The results of these checks are used to conclude if the entire system is Ξ-terminating.

First we prove that it is sufficient to find a Ξ-terminating subsystem to show that the entire system is Ξ-terminating. If such a subsystem can be found, there is no need to compute the entire system. Next, we provide a result that enables us to consider subsets of Ξ. We remove those events from Ξ for which we already know that they cannot be contained in any Ξ-loop of the entire system. This makes it easier to find the desired subsystem, since some Ξ-loops still contained can be ignored. Another result shows how counter-examples can be used to show that a system is not Ξ-terminating. Finally, an outline of an incremental Ξ-termination checking algorithm based on the results is given.

We start by proving that it is sufficient to find a Ξ-terminating subsystem to show that the entire system is Ξ-terminating.

Proposition 5.2.3
Let G_1 and G_2 be deterministic automata over the alphabet Σ, and let $\Xi \subseteq \Sigma$. If G_1 is Ξ-terminating, then $G_1 \,\|\, G_2$ is also Ξ-terminating.

Proof. Let G_1 and G_2 be deterministic automata over the alphabet Σ, and let $\Xi \subseteq \Sigma$. Let G_1 be Ξ-terminating, i.e. $\mathcal{L}(G_1)$ is Ξ-terminating. Assume that $G_1 \,\|\, G_2$ is not Ξ-terminating, i.e. $\mathcal{L}(G_1 \,\|\, G_2)$ is not Ξ-terminating. Then there exists $s \in \mathcal{L}(G_1 \,\|\, G_2)$ such that for all $n \in \mathbb{N}$ there exists $t_n \in \Xi^*$ such that $|t_n| > n$ and $st_n \in \mathcal{L}(G_1 \,\|\, G_2)$. Since $\mathcal{L}(G_1 \,\|\, G_2) = \mathcal{L}(G_1) \cap \mathcal{L}(G_2)$, we have $st_n \in \mathcal{L}(G_1)$ for all $n \in \mathbb{N}$. But then $\mathcal{L}(G_1)$ is not Ξ-terminating, which contradicts the precondition. This proves that $\mathcal{L}(G_1 \,\|\, G_2)$ is Ξ-terminating. □

Another approach is to remove events from Ξ for which we know that they cannot occur in any Ξ-loop of the entire system. Assume we want to check if the synchronous product of several automata is Ξ-terminating and there exists an event $\xi \in \Xi$ and an automaton such that ξ is not contained in any Ξ-loop of this automaton. Then the event ξ cannot occur in a Ξ-loop of the synchronous product, since each Ξ-loop in the synchronous product must also be a Ξ-loop in each of the automata constituting the synchronous product. Therefore, it is sufficient to check if the system is $(\Xi\backslash\{\xi\})$-terminating instead of checking if it is Ξ-terminating. If it is possible to reduce Ξ to the empty set in this way, then the system has been shown to be Ξ-terminating.

Proposition 5.2.4
Let G_1 and G_2 be deterministic automata over the same alphabet Σ, and let $\Xi \subseteq \Sigma$. Furthermore, let $\xi \in \Xi$ such that for each Ξ-loop π of G_1 we have $\xi \notin L(\pi)$. Then $G_1 \,\|\, G_2$ is Ξ-terminating if and only if $G_1 \,\|\, G_2$ is $(\Xi\backslash\{\xi\})$-terminating.

Proof. Let G_1 and G_2 be deterministic automata over the alphabet Σ, let $\Xi \subseteq \Sigma$, and let $G = G_1 \,\|\, G_2$. Furthermore, let $\xi \in \Xi$ such that for each Ξ-loop π of G_1 we have $\xi \notin L(\pi)$, and let $\Xi' = \Xi\backslash\{\xi\}$.

(Only If) Let G be not Ξ'-terminating, i.e. G is not Ξ'-loop free. Then there exists a reachable Ξ'-loop in G. This is also a reachable Ξ-loop in G, so that G is not Ξ-loop free, i.e. G is not Ξ-terminating.

(If) Let G be not Ξ-terminating, i.e. there exists a Ξ-loop

$$\pi \quad = \quad (p_0, q_0) \xrightarrow{\xi_1} \cdots \xrightarrow{\xi_n} (p_n, q_n)$$

in $G = G_1 \,\|\, G_2$. Then

$$\pi_1 \quad = \quad p_0 \xrightarrow{\xi_1} \cdots \xrightarrow{\xi_n} p_n$$

is a Ξ-loop in G_1. Thus, we have $\xi \notin L(\pi_1) = L(\pi)$, i.e. $L(\pi) \in (\Xi')^*$. Therefore, G is not Ξ'-loop free, i.e. G is not Ξ'-terminating. □

So far, we have only investigated in incremental techniques proving a system to be Ξ-terminating. Next, we provide a result that shows how a system can be proven to be not Ξ-terminating without computing the entire system. If a counter-example is accepted by all automata, then it is a counter-example for the synchronous product of these automata, i.e. the system is not Ξ-terminating.

Proposition 5.2.5
Let G_1 and G_2 be deterministic automata over the alphabet Σ, and let $\Xi \subseteq \Sigma$. Furthermore, let (s, t) be an iterative Ξ-pair in $\mathcal{L}(G_1)$ and $\mathcal{L}(G_2)$. Then $G_1 \,\|\, G_2$ is not Ξ-terminating.

Proof. Let G_1 and G_2 be deterministic automata over the alphabet Σ, let $\Xi \subseteq \Sigma$, and let $G = G_1 \,\|\, G_2$. Furthermore, let (s, t) be an iterative Ξ-pair in $\mathcal{L}(G_1)$

and $\mathcal{L}(G_2)$. Then $t \in \Xi^+$, and for each integer n we have $st^n \in \mathcal{L}(G_1)$ and $st^n \in \mathcal{L}(G_2)$, and therefore $st^n \in \mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \mathcal{L}(G)$. Thus, (s,t) is an iterative Ξ-pair in $\mathcal{L}(G)$, and therefore G cannot be Ξ-terminating. $\square$

Now we can give an outline for an incremental Ξ-termination check. Consider a system $G = \|_{i\in I} G_i$, for some finite index set I, composed of deterministic automata for which we want to know whether or not it is Ξ-terminating. The main idea is to construct a Ξ-terminating subsystem, i.e. to find an index set $J \subseteq I$ such that $\|_{j\in J} G_j$ is Ξ-terminating. If such an index set exists, the entire system G is also Ξ-terminating by proposition 5.2.3.

Such a subsystem can be computed incrementally in a similar way as in the incremental language inclusion check given in section 5.2.1. For a given subsystem, we check by means of a state exploration algorithm if it is Ξ-terminating. If this succeeds, we are done and the system is Ξ-terminating. Otherwise, we get a counter-example, i.e. a iterative Ξ-pair (s,t). If this pair is accepted by all automata, i.e. $st^n \in \mathcal{L}(G_i)$, for all $n \in \mathbb{N}$ and $i \in I$, then this is a counter-example for the entire system G by proposition 5.2.5. If there exists an automaton that does not accept this counter-example, the subsystem is incremented by this automaton and the next verification step is started.

Additionally, if a subsystem has been verified to be not Ξ-terminating, we can compute the set Ξ' of all output events occurring in a Ξ-loop. By proposition 5.2.4, it is sufficient to consider this set of events; we continue by searching for Ξ'-terminating subsystems. If Ξ' is the empty set, the entire system is Ξ-terminating.

5.2.3 Ξ-Confluence

In this section, we provide results that permit the development of an incremental Ξ-confluence checking algorithm. An outline for such an algorithm is given. Note that this algorithm requires a Ξ-terminating system to be given. The algorithm can give a wrong result if the system to be checked is not Ξ-terminating.

In section 5.1.2, we have developed a state exploration algorithm that checks if a deterministic automaton is Ξ-confluent. In the following, we provide another approach that breaks down the problem into several smaller problems. Instead of checking a synchronous product of deterministic automata if it is Ξ-confluent, we check each event pair contained in $\Xi \times \Xi$ if it is Ξ-joinable. This is more suitable for an incremental algorithm, since each subtask can be solved with a smaller number of synchronous product operations. We show that many of the pairs can be easily checked without building any synchronous product for most well-modelled practical examples. The remaining event pairs are checked by means of repeating verifications of subsystems that are incremented based on counter-examples.

We start with a result that enables us to break down the task of checking a system if it is Ξ-confluent into several subtasks. To check a Ξ-terminating deterministic automaton if it is Ξ-confluent, it is possible to consider each pair of output events separately. An event pair is called Ξ-joinable in a deterministic automaton if for every reachable state in which both events of the pair are eligible, there exist paths labelled with output events that start at the corresponding successor states and end in states that are equivalent with respect to possible continuations.

Definition 5.2.3 (Ξ-joinable event pair)
Let $G = (\Sigma, Q, \delta, q_0, M)$ be a deterministic automaton, and let $\Xi \subseteq \Sigma$.
A pair $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ is Ξ-joinable *in G if for all $s \in \mathcal{L}(G)$ such that $s\sigma_1, s\sigma_2 \in \mathcal{L}(G)$ there exist $t_1, t_2 \in \Xi^*$ such that $s\sigma_1 t_1, s\sigma_2 t_2 \in \mathcal{L}(G)$ and $s\sigma_1 t_1 \equiv s\sigma_2 t_2 \bmod \mathrm{Nerode}(G)$.*

A Ξ-terminating automaton is Ξ-confluent if and only if all possible pairs of output events are Ξ-joinable. This is a well-known result in term rewriting [DJ90], known as the Diamond or Newman lemma.

Lemma 5.2.6
Let G be a deterministic automaton over the alphabet Σ, and let $\Xi \subseteq \Sigma$. Furthermore, let G be Ξ-terminating.
G is Ξ-confluent if and only if each pair of events in $\Xi \times \Xi$ is Ξ-joinable in G.

Proof. Let $G = (\Sigma, Q, \delta, q_0, M)$ be a deterministic automaton, let $\Xi \subseteq \Sigma$, and let G be Ξ-terminating.

(Only If) Let G be Ξ-confluent, and let $\xi_1, \xi_2 \in \Xi$. Furthermore, let $s \in \mathcal{L}(G)$ such that $s\xi_1, s\xi_2 \in \mathcal{L}(G)$. Since G is Ξ-confluent, there exist $t_1, t_2 \in \Xi^*$ such that $s\xi_1 t_1, s\xi_2 t_2 \in \mathcal{L}(G)$ and $s\xi_1 t_1 \equiv s\xi_2 t_2 \bmod \mathrm{Nerode}(G)$. Thus, (ξ_1, ξ_2) is Ξ-joinable.

(If) Consider the partial ordering

$$\rightarrow \;\subseteq\; \Sigma^* \times \Sigma^*$$

defined as follows: $s \rightarrow t$ for some $s, t \in \Sigma^*$ if $s, t \in \mathcal{L}(G)$ and there exists $s' \in \Xi^+$ such that $ss' = t$. Since G is Ξ-terminating, the relation $\rightarrow$ is a well-founded ordering on Σ^*, i.e. there exists no infinite chain $t_1 \rightarrow t_2 \rightarrow \cdots$ of elements of Σ^*. We show by well-founded induction[2] that for all $s \in \Sigma^*$ we have the following property.

(P) For all $s_1, s_2 \in \Xi^*$ such that $ss_1, ss_2 \in \mathcal{L}(G)$ there exist $t_1, t_2 \in \Xi^*$ such that $ss_1 t_1, ss_2 t_2 \in \mathcal{L}(G)$ and $ss_1 t_1 \equiv ss_2 t_2 \bmod \mathrm{Nerode}(G)$.

[2]Well-founded induction, also called Noetherian induction, is an induction based on well-founded partial orderings [DJ90].

Let $s \in \mathcal{L}(G)$ and assume that (P) is satisfied for all $s' \in \Sigma^*$ such that $s \to s'$. Let $s_1, s_2 \in \Xi^*$ such that $ss_1, ss_2 \in \mathcal{L}(G)$. Consider two cases.

Case 1: $s_1 = \varepsilon$ or $s_2 = \varepsilon$. Then we have $ss_1s_2 = ss_2s_1$. Then we have $ss_1s_2 \in \mathcal{L}(G)$, $ss_2s_1 \in \mathcal{L}(G)$, and $ss_1s_2 \equiv ss_2s_1 \bmod \text{Nerode}(G)$.

Case 2: $s_1 \neq \varepsilon$ and $s_2 \neq \varepsilon$. Let $s_1 = \xi_1 s_1'$ and $s_2 = \xi_2 s_2'$ for some $\xi_1, \xi_2 \in \Xi$, $s_1', s_2' \in \Xi^*$. We have $s\xi_1, s\xi_2 \in \mathcal{L}(G)$. Since (ξ_1, ξ_2) is Ξ-joinable in G, there exist $u_1, u_2 \in \Xi^*$ such that $s\xi_1u_1, s\xi_2u_2 \in \mathcal{L}(G)$ and

$$s\xi_1u_1 \equiv s\xi_2u_2 \bmod \text{Nerode}(G). \tag{5.3}$$

We have $ss_1 = s\xi_1s_1' \in \mathcal{L}(G)$ and $s\xi_1u_1 \in \mathcal{L}(G)$. Since $s \to s\xi_1$, property (P) is satisfied for $s\xi_1$, i.e. there exist $v_1, v_2 \in \Xi^*$ such that $s\xi_1s_1'v_1 \in \mathcal{L}(G)$, $s\xi_1u_1v_2 \in \mathcal{L}(G)$ and

$$s\xi_1s_1'v_1 \equiv s\xi_1u_1v_2 \bmod \text{Nerode}(G). \tag{5.4}$$

By equation (5.3) and since and $s\xi_1u_1v_2 \in \mathcal{L}(G)$, we have $s\xi_2u_2v_2 \in \mathcal{L}(G)$. Furthermore, we have $s\xi_2s_2' = ss_2 \in \mathcal{L}(G)$. Since $s \to s\xi_2$, property (P) is satisfied for $s\xi_2$, i.e. there exist $w_1, w_2 \in \Xi^*$ such that $s\xi_2u_2v_2w_1 \in \mathcal{L}(G)$, $s\xi_2s_2'w_2 \in \mathcal{L}(G)$, and

$$s\xi_2u_2v_2w_1 \equiv s\xi_2s_2'w_2 \bmod \text{Nerode}(G). \tag{5.5}$$

Consider $t_1 := v_1w_1$ and $t_2 := w_2$. We have $ss_2t_2 = s\xi_2s_2'w_2 \in \mathcal{L}(G)$ and

$$\begin{aligned} s\xi_1s_1'v_1w_1 &\overset{(5.4)}{\equiv} s\xi_1u_1v_2w_1 \\ &\overset{(5.3)}{\equiv} s\xi_2u_2v_2w_1 \bmod \text{Nerode}(G) \\ &\overset{(5.5)}{\equiv} s\xi_2s_2'w_2 \bmod \text{Nerode}(G). \end{aligned}$$

Thus, we have $ss_1t_1 \in \mathcal{L}(G)$, $ss_2t_2 \in \mathcal{L}(G)$, and $ss_1t_1 \equiv ss_2t_2 \bmod \text{Nerode}(G)$. □

Not all pairs of events must be explicitly checked to verify whether or not a deterministic automaton is Ξ-confluent, and many pairs can be very easily dealt with. Firstly, the order of the events in a pair is not relevant. If a pair (σ_1, σ_2) is known to be Ξ-joinable, then (σ_2, σ_1) is also known to be Ξ-joinable.

Lemma 5.2.7
Let G be a deterministic automaton over the alphabet Σ, and let $\Xi \subseteq \Sigma$.
If the pair $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ is Ξ-joinable in G, then (σ_2, σ_1) is Ξ-joinable in G.

Proof. This follows directly from definition 5.2.3. □

Secondly, we do not have to check pairs of equal events.

Lemma 5.2.8
Let G be a deterministic automaton over the alphabet Σ, and let $\sigma \in \Sigma$. The pair (σ, σ) is $\emptyset$-joinable in G.

Proof. Let G be a deterministic automaton over the alphabet Σ, and let $\sigma \in \Sigma$. Let $s \in \mathcal{L}(G)$ such that $s\sigma \in \mathcal{L}(G)$. Then $s\sigma\varepsilon \in \mathcal{L}(G)$ and $s\sigma\varepsilon \equiv s\sigma\varepsilon \bmod \mathrm{Nerode}(G)$, i.e. (σ, σ) is $\emptyset$-joinable. □

If a pair is Ξ-joinable with respect to an alphabet, then it is also Ξ-joinable with respect to a larger alphabet, i.e. an alphabet containing the original alphabet.

Lemma 5.2.9
Let G be a deterministic automaton over the alphabet Σ, and let $\Xi' \subseteq \Xi \subseteq \Sigma$. If the pair $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ is Ξ'-joinable in G, then (σ_1, σ_2) is Ξ-joinable in G.

Proof. This follows directly from definition 5.2.3. □

Another large number of event pairs can be checked for Ξ-joinability by exploiting special cases that are easily testable. For instance, many events are independent from each other, i.e. they may occur in any order. Each commuting pair of output events is Ξ-joinable. This is an immediate result of the following lemma.

Lemma 5.2.10
Let G be a deterministic automaton over the alphabet Σ, and let $\sigma_1, \sigma_2 \in \Sigma$. If (σ_1, σ_2) is commuting for G, then (σ_1, σ_2) is $\{\sigma_1, \sigma_2\}$-joinable in G.

Proof. Let $G = (\Sigma, Q, \delta, q_0, M)$ be a deterministic automaton, and let $\sigma_1, \sigma_2 \in \Sigma$ such that (σ_1, σ_2) is commuting for G, i.e. for all $q \in Q_{\text{reach}}(G)$ we have that $\sigma_1, \sigma_2 \in \textsc{Elig}_G(q)$ implies that $\delta(q, \sigma_1\sigma_2)$ and $\delta(q, \sigma_2\sigma_1)$ are both defined and $\delta(q, \sigma_1\sigma_2) = \delta(q, \sigma_2\sigma_1)$. Now let $s \in \mathcal{L}(G)$ such that $s\sigma_1, s\sigma_2 \in \mathcal{L}(G)$. Then $q := \delta(q_0, s)$ is reachable in G and $\sigma_1, \sigma_2 \in \textsc{Elig}_G(q)$. Therefore, $\delta(q, \sigma_1\sigma_2)$ and $\delta(q, \sigma_2\sigma_1)$ are both defined and $\delta(q, \sigma_1\sigma_2) = \delta(q, \sigma_2\sigma_1)$, i.e. $s\sigma_1\sigma_2 \in \mathcal{L}(G)$, $s\sigma_2\sigma_1 \in \mathcal{L}(G)$, and $s\sigma_1\sigma_2 \equiv s\sigma_2\sigma_1 \bmod \mathrm{Nerode}(G)$. □

Another special case is the following. An event pair is Ξ-joinable in the synchronous product of several automata if one of them enables at most one partner of this pair after each accepted string.

Lemma 5.2.11
Let G_1 and G_2 be deterministic automata over the same alphabet Σ, let $\Xi \subseteq \Sigma$, and let $\sigma_1, \sigma_2 \in \Sigma$. If for all $s \in \mathcal{L}(G_1)$ either $s\sigma_1 \notin \mathcal{L}(G_1)$ or $s\sigma_2 \notin \mathcal{L}(G_1)$, then (σ_1, σ_2) is Ξ-joinable in $G_1 \,\|\, G_2$.

Proof. Let G_1 and G_2 be deterministic automata over the same alphabet Σ, let $\Xi \subseteq \Sigma$, let $\sigma_1, \sigma_2 \in \Sigma$, and let, for all $s \in \mathcal{L}(G_1)$, either $s\sigma_1 \notin \mathcal{L}(G_1)$ or $s\sigma_2 \notin \mathcal{L}(G_1)$. We show that (σ_1, σ_2) is Ξ-joinable in $G_1 \| G_2$. Let $s \in \mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ such that $s\sigma_1, s\sigma_2 \in \mathcal{L}(G_1) \cap \mathcal{L}(G_2)$. Then $s\sigma_1, s\sigma_2 \in \mathcal{L}(G_1)$, but this we excluded in the assumption above. Such an s cannot exist, and therefore (σ_1, σ_2) is joinable in $G_1 \| G_2$. □

If an event pair is Ξ-joinable in two automata by using only events that are not relevant for the other automaton, then this pair is also Ξ-joinable in the composed system.

Proposition 5.2.12
Let G_1 and G_2 be deterministic automata over the same alphabet, let $\Upsilon_1 \subseteq \Sigma$ be not relevant for G_2, and let $\Upsilon_2 \subseteq \Sigma$ be not relevant for G_1.
If $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ is Υ_1-joinable in G_1 and Υ_2-joinable in G_2, then (σ_1, σ_2) is $(\Upsilon_1 \cup \Upsilon_2)$-joinable in $G_1 \| G_2$.

Proof. Let G_1 and G_2 be deterministic automata over the same alphabet Σ, let $\Upsilon_1 \subseteq \Sigma$ be not relevant for G_2, and let $\Upsilon_2 \subseteq \Sigma$ be not relevant for G_1. Furthermore, let $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ be Υ_1-joinable in G_1 and Υ_2-joinable in G_2. Consider $s \in \mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ such that $s\sigma_1, s\sigma_2 \in \mathcal{L}(G_1) \cap \mathcal{L}(G_2)$. Since (σ_1, σ_2) is Υ_1-joinable in G_1, there exist $s_1, s_2 \in \Upsilon_1^*$ such that $s\sigma_1 s_1, s\sigma_2 s_2 \in \mathcal{L}(G_1)$ and $s\sigma_1 s_1 \equiv s\sigma_2 s_2 \bmod \mathrm{Nerode}(G_1)$. Since (σ_1, σ_2) is Υ_2-joinable in G_2, there exist $t_1, t_2 \in \Upsilon_2^*$ such that $s\sigma_1 t_1, s\sigma_2 t_2 \in \mathcal{L}(G_2)$ and $s\sigma_1 s_1 \equiv s\sigma_2 s_2 \bmod \mathrm{Nerode}(G_2)$. Since Υ_1 is not relevant for G_2 and Υ_2 is not relevant for G_1, we have $s\sigma_1 s_1 t_2, s\sigma_2 s_2 t_2 \in \mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ and $s\sigma_1 s_1 t_1 \equiv s\sigma_2 s_2 t_2 \bmod \mathrm{Nerode}(G_1 \| G_2)$. Thus, (σ_1, σ_2) is $(\Upsilon_1 \cup \Upsilon_2)$-joinable in $G_1 \| G_2$. □

If an event pair is Ξ-joinable in two automata by means of the same strings, then this pair is also Ξ-joinable in the synchronous product of both automata. We first define (s_1, s_2)-joinable event pairs.

Definition 5.2.4 ((s_1, s_2)-joinable event pair)
Let $G = (\Sigma, Q, \delta, q_0, M)$ be a deterministic automaton, let $\Xi \subseteq \Sigma$, and let $s_1, s_2 \in \Xi^*$.
A pair $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ is (s_1, s_2)-joinable *in G if for all $s \in \mathcal{L}(G)$ such that $s\sigma_1, s\sigma_2 \in \mathcal{L}(G)$ we have $s\sigma_1 s_1, s\sigma_2 s_2 \in \mathcal{L}(G)$ and $s\sigma_1 s_1 \equiv s\sigma_2 s_2 \bmod \mathrm{Nerode}(G)$.*

Next, we show that if an event pair is (s_1, s_2)-joinable in two automata, then this pair is (s_1, s_2)-joinable in the synchronous product of both automata.

Proposition 5.2.13
Let G_1 and G_2 be deterministic automata over the same alphabet Σ, let $\Xi \subseteq \Sigma$, and let $s_1, s_2 \in \Xi^*$.
If $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ is (s_1, s_2)-joinable in G_1 and G_2, then (σ_1, σ_2) is Ξ-joinable in $G_1 \| G_2$.

Proof. Let G_1 and G_2 be deterministic automata over the same alphabet Σ, let $\Xi \subseteq \Sigma$, let $s_1, s_2 \in \Xi^*$, and let $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ be (s_1, s_2)-joinable in G_1 and G_2. Write $G = G_1 \,\|\, G_2$.

We show that (σ_1, σ_2) is Ξ-joinable in G. Let $s \in \Sigma^*$ such that $s\sigma_1, s\sigma_2 \in \mathcal{L}(G) = \mathcal{L}(G_1) \cap \mathcal{L}(G_2)$. Then we have $s\sigma_1, s\sigma_2 \in \mathcal{L}(G_1)$ and $s\sigma_1, s\sigma_2 \in \mathcal{L}(G_2)$. Since (σ_1, σ_2) is (s_1, s_2)-joinable in G_1 and G_2, we have $s\sigma_1 s_1, s\sigma_2 s_2 \in \mathcal{L}(G_1)$, $s\sigma_1 s_1, s\sigma_2 s_2 \in \mathcal{L}(G_2)$, $s\sigma_1 s_1 \equiv s\sigma_2 s_2 \bmod \mathrm{Nerode}(G_1)$, and $s\sigma_1 s_1 \equiv s\sigma_2 s_2 \bmod \mathrm{Nerode}(G_2)$. Thus, $s\sigma_1 s_1, s\sigma_2 s_2 \in \mathcal{L}(G)$ and $s\sigma_1 s_1, s\sigma_2 s_2 \in \mathcal{L}(G)$, i.e. (σ_1, σ_2) is Ξ-joinable in $G_1 \,\|\, G_2$. ☐

If a pair of events is not Ξ-joinable in an automaton, a counter-example should be computed. A Ξ-joinable counter-example for a pair of events is a string that leads to a state from which the pair cannot joined by means of output events only.

Definition 5.2.5 (Ξ-joinable counter-example)
Let G be a deterministic automaton, let $\Xi \subseteq \Sigma$, and let $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$. A Ξ-joinable counter-example *for (σ, σ_2) in G is a string $s \in \mathcal{L}(G)$ such that $s\sigma_1, s\sigma_2 \in \mathcal{L}(G)$ and for all $t_1, t_2 \in \Xi^*$ either $s\sigma_1 t_1 \notin \mathcal{L}(G)$, $s\sigma_2 t_2 \notin \mathcal{L}(G)$, or $s\sigma_1 t_1 \not\equiv s\sigma_2 t_2 \bmod \mathrm{Nerode}(G)$.*

If there exists a Ξ-joinable counter-example for a pair $(\xi_1, \xi_2) \in \Xi \times \Xi$ in an automaton, then this automaton cannot be Ξ-confluent. Conversely, if we can prove that there does not exist a Ξ-joinable counter-example for any pair in $\Xi \times \Xi$, then we have proven that the automaton is Ξ-confluent.

Lemma 5.2.14
Let G be a deterministic automaton over the alphabet Σ, let $\Xi \subseteq \Sigma$, and let G be Ξ-terminating. G is Ξ-confluent if and only if G does not have a pair in $\Xi \times \Xi$ and a Ξ-joinable counter-example for this pair.

Proof. This is an immediate consequence of lemma 5.2.6. ☐

In some cases, if we find a counter-example for a subsystem, this is also a counter-example for the composed system. This applies if the string extended by each of the two events is accepted by the other parts of the system.

Proposition 5.2.15
Let G_1 and G_2 be deterministic automata over the alphabet Σ, let $\Xi \subseteq \Sigma$, and let (s, σ_1, σ_2) be a Ξ-confluence counter-example for G_1. If $s\sigma_1, s\sigma_2 \in \mathcal{L}(G_2)$, then (s, σ_1, σ_2) is a Ξ-confluence counter-example for $G_1 \,\|\, G_2$.

Proof. Let G_1 and G_2 be deterministic automata over the alphabet Σ, let $\Xi \subseteq \Sigma$, and let (s, σ_1, σ_2) be a Ξ-confluence counter-example for G_1. Furthermore, let $s\sigma_1, s\sigma_2 \in \mathcal{L}(G_2)$. Since (s, σ_1, σ_2) is a Ξ-confluence counter-example for G_1, we have $s\sigma_1, s\sigma_2 \in \mathcal{L}(G_1)$, and for all $t_1, t_2 \in \Xi^*$ we have either $s\sigma_1 t_1 \notin \mathcal{L}(G_1)$,

$s\sigma_2 t_2 \notin \mathcal{L}(G_1)$, or $s\sigma_1 t_1 \not\equiv s\sigma_2 t_2 \bmod \mathrm{Nerode}(G_1)$. But $s\sigma_1 t_1 \notin \mathcal{L}(G_1)$ implies $s\sigma_1 t_1 \notin \mathcal{L}(G_1) \cap \mathcal{L}(G_2)$, $s\sigma_2 t_2 \notin \mathcal{L}(G_1)$ implies $s\sigma_2 t_2 \notin \mathcal{L}(G_1) \cap \mathcal{L}(G_2)$, and $s\sigma_1 t_1 \not\equiv s\sigma_2 t_2 \bmod \mathrm{Nerode}(G_1)$ implies $s\sigma_1 t_1 \not\equiv s\sigma_2 t_2 \bmod \mathrm{Nerode}(G_1 \parallel G_2)$. Since we also have $s\sigma_1 \in \mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ and $s\sigma_2 \in \mathcal{L}(G_1) \cap \mathcal{L}(G_2)$, this implies that (s, σ_1, σ_2) is a Ξ-confluence counter-example for $G_1 \parallel G_2$. □

Now we give an outline for an incremental Ξ-confluence checking algorithm. Let $G_1, \ldots, G_n$ be deterministic automata over the alphabet Σ. We want to check if $G = G_1 \parallel \cdots \parallel G_n$ is Ξ-confluent. The main idea is to apply lemma 5.2.6, which shows that it is sufficient to test all pairs of output events for Ξ-joinability. Actually, we do not have to test all pairs: lemma 5.2.7 states that a pair (σ_1, σ_2) is Ξ-joinable if and only if (σ_2, σ_1) is Ξ-joinable. Furthermore, lemma 5.2.8 states that each event is Ξ-joinable with itself, so this must not be tested explicitly.

We may validate many pairs (ξ_1, ξ_2) of events as Ξ-joinable by finding Ξ-commuting event pairs and applying lemma 5.2.10. A very efficient incremental algorithm that checks for commuting event pairs is described in section 5.2.1. Furthermore, by lemma 5.2.11, if there exists an automaton G_i such that, for each reachable state q of G_i, $\xi_1 \notin \textsc{Elig}_{G_i}(q)$ or $\xi_2 \notin \textsc{Elig}_{G_i}(q)$, then (ξ_1, ξ_2) is Ξ-joinable, since both events are never enabled together in G_i and thus in the synchronous product of all automata.

Now we pick a pair $(\xi_1, \xi_2) \in \Xi \times \Xi$ for which we do not yet know whether it is Ξ-joinable or not—if there is none, G is Ξ-confluent. Next, we compute the synchronous product of all automata G_k, $k \in I$, for which there exists a reachable state $q \in Q_{G_k}$ such that $\xi_1, \xi_2 \in \textsc{Elig}_{G_k}(q)$ and $\delta(q, \xi_1) \neq \delta(q, \xi_2)$ and check if (ξ_1, ξ_2) is Ξ-joinable in the resultant automaton.

If (ξ_1, ξ_2) is not Ξ-joinable, we obtain a counter-example s. If this counter-example is accepted by all automata, i.e. if $s\xi_1, s\xi_2 \in \mathcal{L}(G_i)$ for each $i \in I$, then we found a counter-example for the entire system by proposition 5.2.15. If there exists an automaton that does not accept the counter-example, we may add this automaton and start a new verification step.

If (ξ_1, ξ_2) is Ξ-joinable, we still have to check if this is also the case for the entire system. By 5.2.12, if there exists an $\Upsilon \subseteq \Xi$ such that Υ is not relevant for all other automata not considered and (ξ_1, ξ_2) is Υ-joinable, then (ξ_1, ξ_2) is Ξ-joinable in the entire system. By 5.2.13, if (ξ_1, ξ_2) is (s_1, s_2)-joinable in the automaton considered and in all remaining automata, for some $s_1, s_2 \in \Xi^*$, then again (ξ_1, ξ_2) is Ξ-joinable in G. If neither proposition 5.2.12 nor proposition 5.2.13 is applicable, we can either minimise the automaton or add other automata and check the remaining pairs using the new subsystem.

Note the following difficulty when checking a pair of events for Ξ-joinability. It is not practicable to check several states if they are Myhill-Nerode equivalent. Instead of checking for Ξ-joinability, we can check for Ξ-state-joinable event pairs.

Definition 5.2.6 (Ξ-state-joinable event pair)
Let $G = (\Sigma, Q, \delta, q_0, M)$ be a deterministic automaton, and let $\Xi \subseteq \Sigma$. A pair

$(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ *is* Ξ-state-joinable *in* G *if for all reachable states* $q \in Q_{\text{reach}}(G)$ *such that* $\sigma_1, \sigma_2 \in \textsc{Elig}_G(q)$ *there exist* $t_1, t_2 \in \Xi^*$ *such that* $\delta(q, \sigma_1 t_1)$ *and* $\delta(q, \sigma_2 t_2)$ *are both defined and* $\delta(q, \sigma_1 t_1) = \delta(q, \sigma_2 t_2)$*.*

Here, it is sufficient to find a common target state instead of finding Myhill-Nerode equivalent states. If a pair is Ξ-state-joinable, then it is Ξ-joinable.

Lemma 5.2.16
Let $G = (\Sigma, Q, \delta, q_0, M)$ *be a deterministic automaton, and let* $\Xi \subseteq \Sigma$*.*
If $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ *is* Ξ*-state-joinable, then* (σ_1, σ_2) *is* Ξ*-joinable.*

In general, not every Ξ-joinable pair is Ξ-state-joinable. However, every Ξ-joinable pair is Ξ-state-joinable if the automaton has a minimal number of states.

Lemma 5.2.17
Let G *be a deterministic minimum-state automaton over the alphabet* Σ*, and let* $\Xi \subseteq \Sigma$*. If* $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ *is* Ξ*-joinable, then* (σ_1, σ_2) *is* Ξ*-state-joinable.*

Therefore, if we have a Ξ-state-joinable pair of events, then we can be sure that this pair is also Ξ-joinable. Conversely, if a pair of events is not Ξ-state-joinable and we do not know if the automaton has minimal number of states, then we cannot conclude that the event pair is not Ξ-joinable. We then can apply a minimisation algorithm and check if the pair is still Ξ-state-joinable. If this is the case, the pair is also Ξ-joinable. Minimising an automaton is rather expensive[3] and should be applied as little as possible, therefore it may be a good idea to try proving Ξ-state-confluence using other automata first, before minimising.

Many variations and improvements of this algorithm are possible. One important step is the development of heuristics to determine which automata should be combined. Furthermore, some more special cases may be tested to find most of the Ξ-joinable pairs without building any synchronous product automata.

5.2.4 Ξ-Nonblocking

In the previous sections, we have proven results that it is possible to consider parts of a system to verify that the entire system is Ξ-terminating and Ξ-confluent. For instance, we were able to prove that if there exists a Ξ-terminating subsystem, then the entire system is also Ξ-terminating. Such results cannot be provided for a Ξ-nonblocking check. The synchronous product of two Ξ-nonblocking automata is in general not Ξ-nonblocking. Therefore, a somewhat different approach is introduced here that enables us to check this property incrementally.

Most models contain events that are relevant for a certain part of the model and are irrelevant for all other parts. We can substitute parts of the system by smaller representations of these parts by removing such irrelevant events.

[3] A minimisation algorithm of complexity $n \log n$ is given in [Hop71].

This idea can be used by an incremental Ξ-nonblocking check. The synchronous product of all automata describing a system is computed incrementally, adding more and more automata. Intermediate results of such computations are simplified before the next automaton is added. When simplifying automata it is convenient to assume that the entire system is Ξ-terminating and Ξ-confluent. The incremental algorithm proposed here may give a wrong answer if the system is not Ξ-terminating or not Ξ-confluent, but it is correct if the system satisfies these properties.

This section starts with an example demonstrating the incremental Ξ-nonblocking check. Afterwards, we describe the concepts and outline the incremental algorithm.

An Example

In the following, parts of an incremental Ξ-nonblocking check for some automata of the car electronics example described in section 6.2 are demonstrated. This motivates the formal definitions given later in this section.

Assume we want to know if the synchronous product of the automata given in figure 5.1 on page 66 is Ξ-nonblocking. We can check this system by means of a state exploration algorithm as introduced in section 5.1.3. In this case, we compute the synchronous product of all automata, consisting of 113 reachable states, and perform the check by inspecting this automaton.

In the following, an incremental algorithm is described, which does not need to compute all reachable states. We assume that we have already checked that the system is Ξ-terminating and Ξ-confluent. Figure 5.2 lists all events involved in this example, shows which events are output events, and indicates which event is relevant for which automaton. The basic idea of the incremental algorithm is to simplify automata containing relevant events that are irrelevant for all other automata. Figure 5.3 shows the steps performed by such an algorithm.

The algorithm considers sets of automata, starting with the given deterministic automata

$$\mathcal{S} = \{\texttt{comfort\_function}, \texttt{comfort\_conditions}, \texttt{comfort\_stop}\}$$

describing the system. In a first step, these automata are transformed to so-called η-partitioned automata. The given automata are duplicated and corresponding states are connected via transitions labelled with a new event η, leading to the new set

$$\mathcal{S}_1 = \{A_1, A_2, A_3\}$$

of η-partitioned automata.

These η-partitioned automata are constructed in such a way that the synchronous product of the original automata in $\mathcal{S}$ is Ξ-nonblocking if and only if

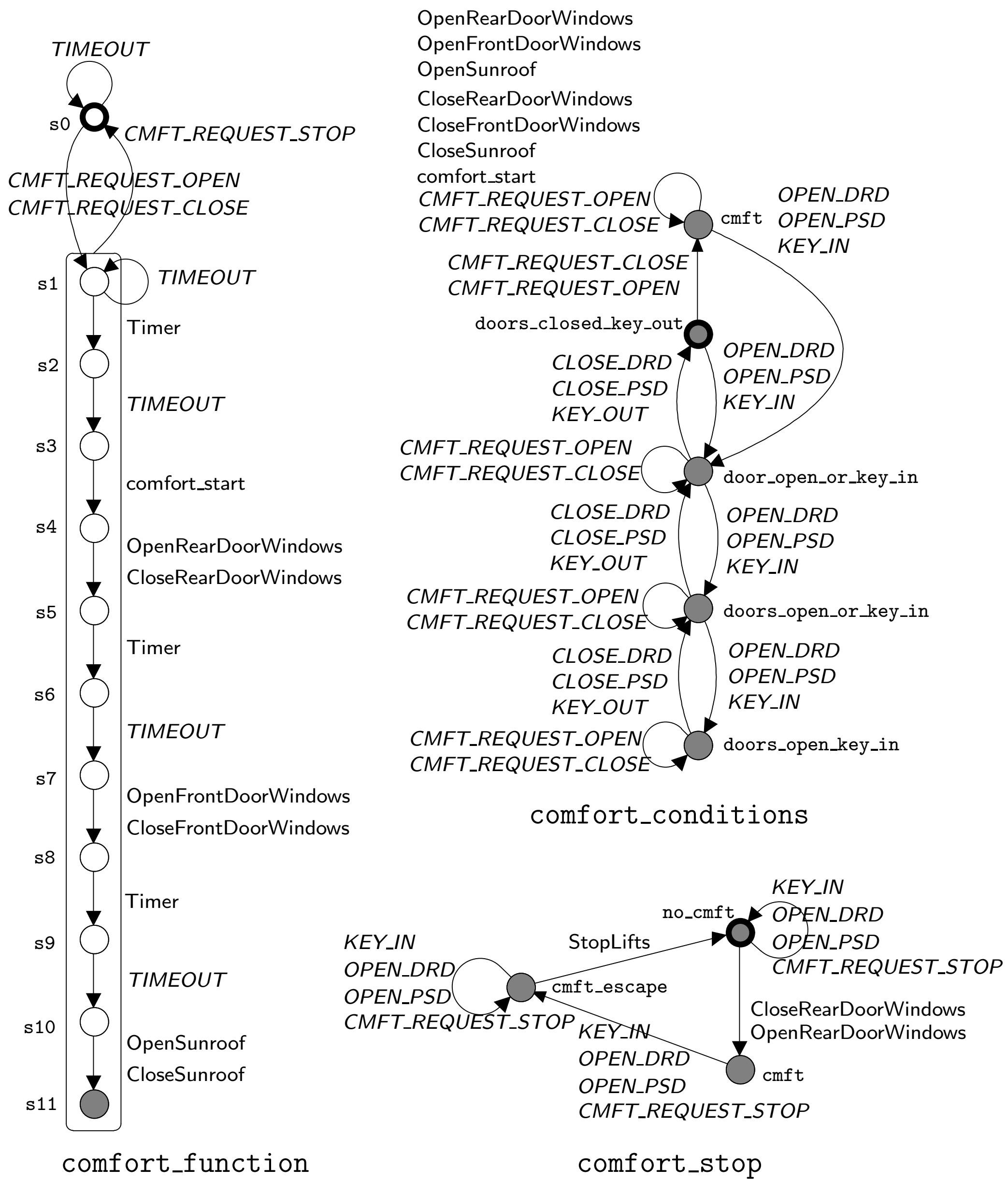

Figure 5.1: Is the synchronous product of these automata Ξ-nonblocking?

	`comfort_function` A_1	`comfort_conditions` A_2	`comfort_stop` A_3
Events contained in Ξ (output events):			
CloseFrontDoorWindows	$\times$	$\times$	
CloseRearDoorWindows	$\times$	$\times$	$\times$
CloseSunroof	$\times$	$\times$	
comfort_start	$\times$	$\times$	
OpenFrontDoorWindows	$\times$	$\times$	
OpenRearDoorWindows	$\times$	$\times$	$\times$
OpenSunroof	$\times$	$\times$	
Timer	$\times$		
StopLifts			$\times$
Events not contained in Ξ (input events):			
CLOSE_DRD		$\times$	
CLOSE_PSD		$\times$	
CMFT_REQUEST_CLOSE	$\times$	$\times$	
CMFT_REQUEST_OPEN	$\times$	$\times$	
CMFT_REQUEST_STOP	$\times$		$\times$
KEY_IN		$\times$	$\times$
KEY_OUT		$\times$	
OPEN_DRD		$\times$	$\times$
OPEN_PSD		$\times$	$\times$
TIMEOUT	$\times$		

Figure 5.2: Relevant events for the automata in figure 5.1.

the synchronous product of the η-partitioned automata in $\mathcal{S}_1$ satisfies a property called (η, Ξ)-nonblocking. An η-partitioned automaton is (η, Ξ)-nonblocking if each state, reachable by a string that ends with the event η, is Ξ-coreachable. In order to check that the synchronous product of the given deterministic automata is Ξ-nonblocking, it suffices to check if the synchronous product of the η-partitioned automata A_1, A_2, and A_3 is (η, Ξ)-nonblocking.

Before computing synchronous products, we can apply certain reduction operations to the η-partitioned automata to simplify them. Figure 5.3 shows the steps performed in the example. Each of the automata A_1, A_2, and A_3 contains events that are only relevant for this automaton and are irrelevant for the other automata. Such events can be identified in figure 5.2, since the η-partitioned automata contain, except for event η, the same relevant events as the corresponding deterministic automata.

Events Timer and *TIMEOUT*, for instance, are only relevant for automaton A_1 shown in figure 5.4. This fact can be used to simplify automaton A_1 by means

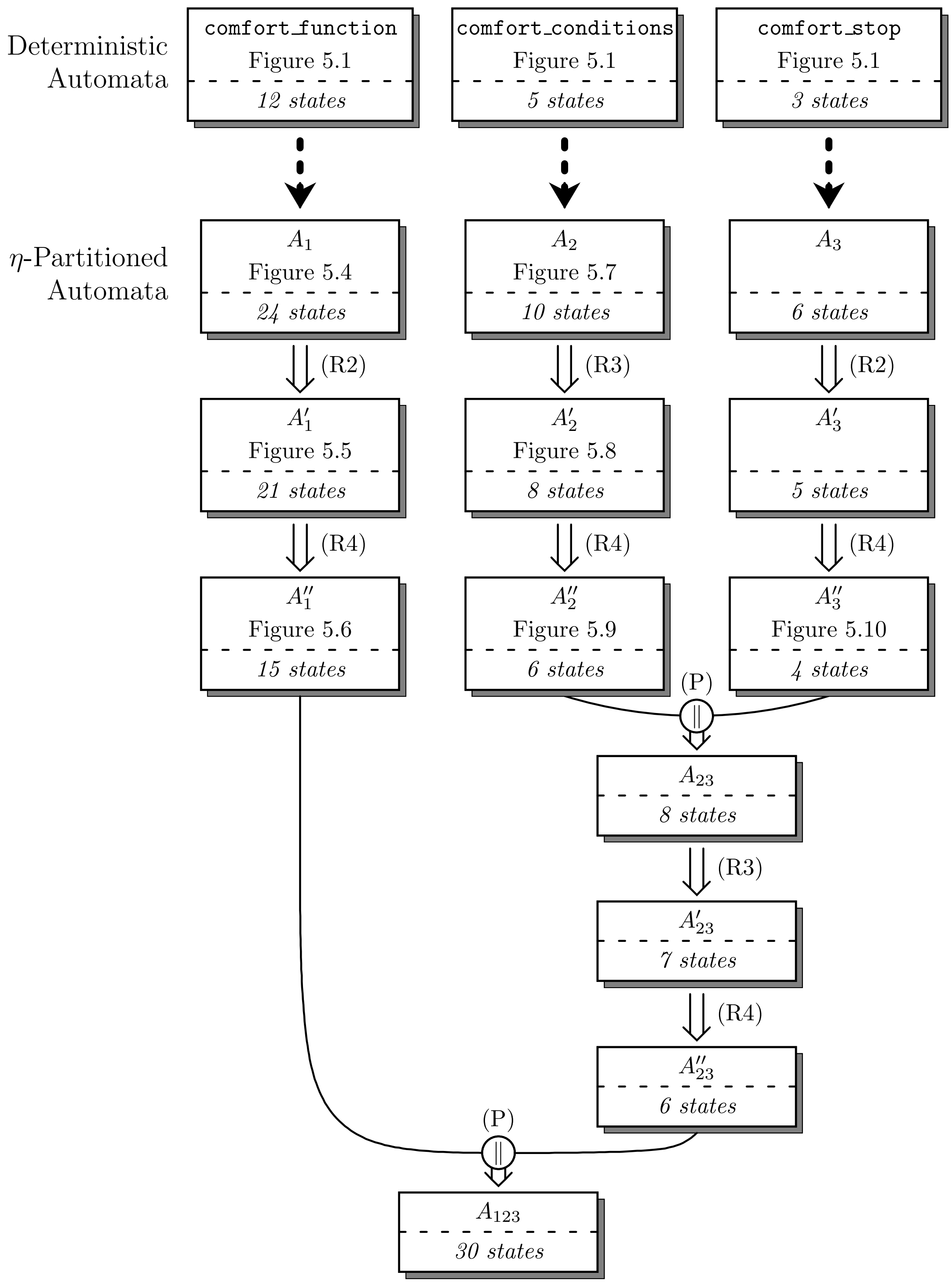

Figure 5.3: Steps of an incremental Ξ-nonblocking check.

of two reduction steps obtaining first automaton A_1' in figure 5.5 and then automaton A_1'' in figure 5.6. Automaton A_1 is substituted by the reduced automata, replacing the set $\mathcal{S}_1$ of η-partitioned automata by

$$\begin{aligned} \mathcal{S}_2 &= \{A_1', A_2, A_3\} \text{ and} \\ \mathcal{S}_3 &= \{A_1'', A_2, A_3\}. \end{aligned}$$

The reduction operations are performed in such a way that $\mathcal{S}_1$ is (η, Ξ)-nonblocking[4] if and only if $\mathcal{S}_2$ is (η, Ξ)-nonblocking, and $\mathcal{S}_2$ is (η, Ξ)-nonblocking if and only if $\mathcal{S}_3$ is (η, Ξ)-nonblocking.

Automaton A_2 in figure 5.7 can also be simplified by means of two reduction steps, obtaining first automaton A_2' in figure 5.8 and then automaton A_2'' in figure 5.9, by taking into account that *CLOSE_DRD*, *CLOSE_PSD*, and *KEY_OUT* are only relevant for this part of the system. Automaton A_3 can be simplified, exploiting that StopLifts is irrelevant for A_1 and A_2, which results in the reduced automaton A_3'' shown in figure 5.10. By replacing in $\mathcal{S}_3$ the automata A_2 and A_3 by the simplified automata A_2'' and A_3'', we obtain the set

$$\mathcal{S}_4 = \{A_1'', A_2'', A_3''\}.$$

Again, the performed reduction steps preserve the property of being (η, Ξ)-nonblocking. Set $\mathcal{S}_4$ is (η, Ξ)-nonblocking if and only if $\mathcal{S}_3$ is (η, Ξ)-nonblocking.

Since no more reduction operations are applicable to single automata, the synchronous product of two automata is computed. We decide to compute the synchronous product of automata A_2'' and A_3''; the resultant automaton is called A_{23} and replaces the two composed automata yielding

$$\mathcal{S}_5 = \{A_1'', A_{23}\}.$$

Again, $\mathcal{S}_5$ is (η, Ξ)-nonblocking if and only if $\mathcal{S}_4$ is (η, Ξ)-nonblocking. Automaton A_{23} can be simplified again before the synchronous product with A_1'' is computed. Events *OPEN_DRD*, *OPEN_PSD*, and *KEY_IN* are relevant for automaton A_{23} and irrelevant for automaton A_1''. This can be exploited by two reduction steps. We obtain

$$\mathcal{S}_6 = \{A_1'', A_{23}''\},$$

in which A_{23} is replaced by the simplified automaton A_{23}''. We have that $\mathcal{S}_6$ is (η, Ξ)-nonblocking if and only if $\mathcal{S}_5$ is (η, Ξ)-nonblocking. Then the synchronous product of automata A_1'' and A_{23}'' is computed and replaces the two composed automata. We finally obtain

$$\mathcal{S}_7 = \{A_{123}\}.$$

[4] A set of automata is (η, Ξ)-nonblocking if the synchronous product of the automata contained in this set is (η, Ξ)-nonblocking.

$\mathcal{S}_7$ is (η, Ξ)-nonblocking if and only if the automata in figure 5.1 are Ξ-nonblocking, since every reduction operation and the synchronous product operation preserves the property of being (η, Ξ)-nonblocking. A check if each η-reachable state in A_{123} is Ξ-coreachable provides us with the desired result. We have proven that the system is Ξ-nonblocking.

The number of states for each automaton computed by the algorithm is given in figure 5.3. Instead of computing 113 reachable states, we have solved the problem incrementally. The largest automaton computed by this algorithm has 30 reachable states. While this result is not really useful for such a small example, applying this methodology to larger models can bring considerable improvements. We may obtain results, even if the reachable state space is too large to be computed explicitly.

In the remainder of this chapter, we provide a formal introduction of the reduction operations and prove their correctness. We start by defining nondeterministic η-partitioned automata and show how they can be derived from deterministic automata. We prove that checking if a deterministic automaton is Ξ-nonblocking is equivalent to checking if the corresponding η-partitioned automaton is (η, Ξ)-nonblocking. Afterwards, each of the reduction operations (R1) to (R4) is discussed in a subsection of its own. It is shown that the reduction operations preserve the property to be (η, Ξ)-nonblocking. In addition, it is shown that the reduction operations preserve another property called (η, Ξ)-nice. This is an invariant which we derive from the assumption that the original system is Ξ-terminating and Ξ-confluent.

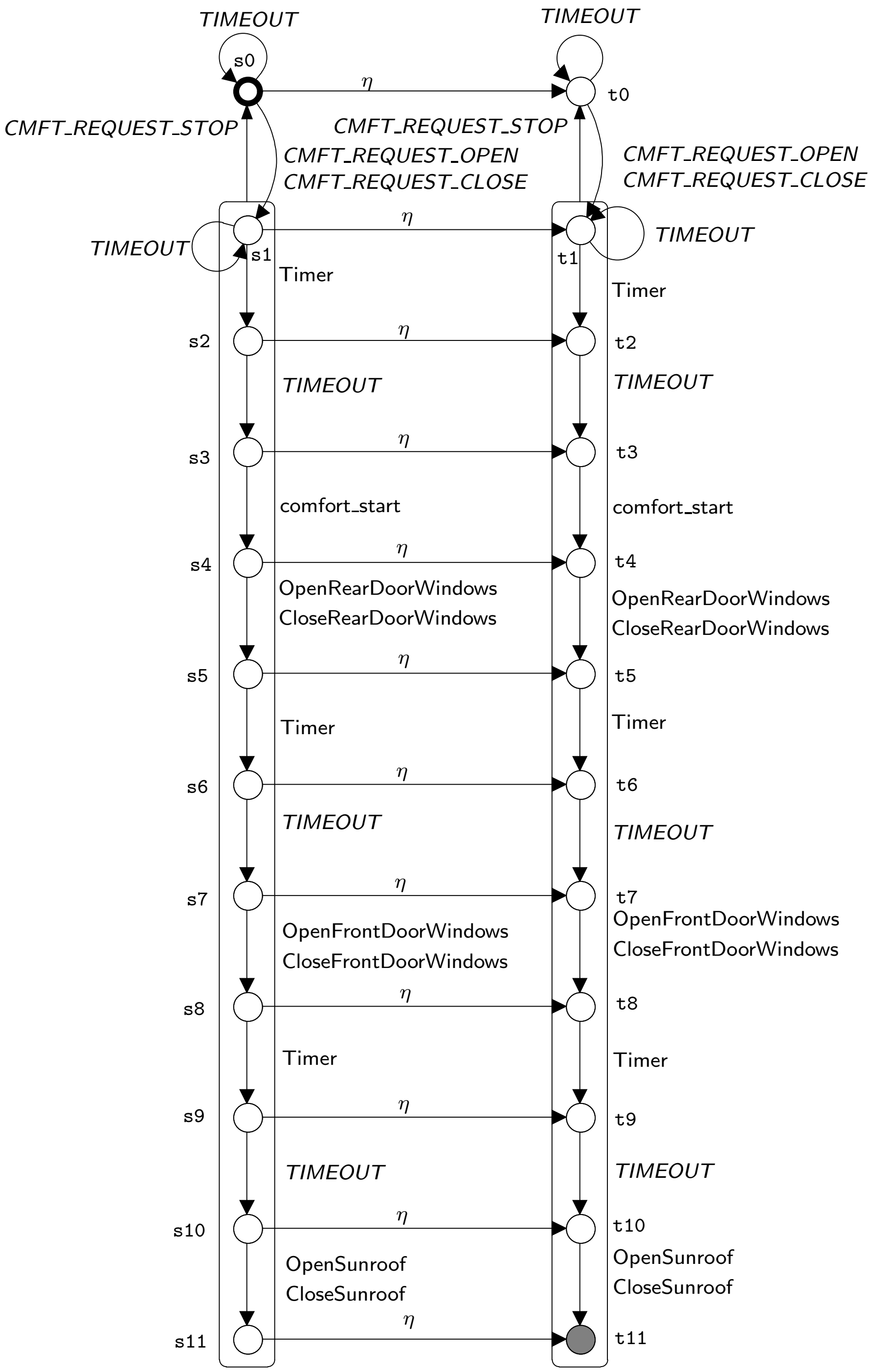

Figure 5.4: Automaton A_1.

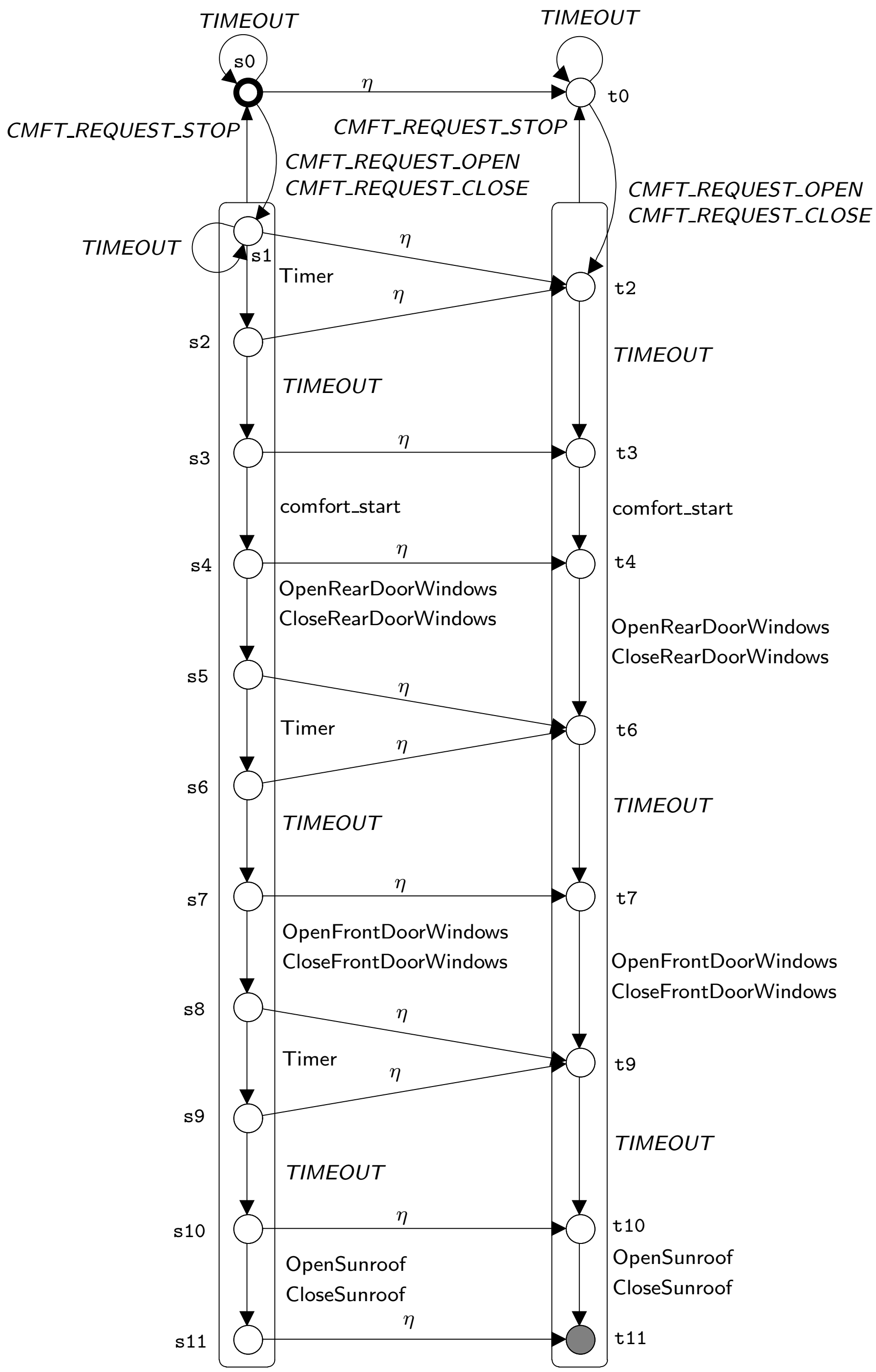

Figure 5.5: Automaton A_1'.

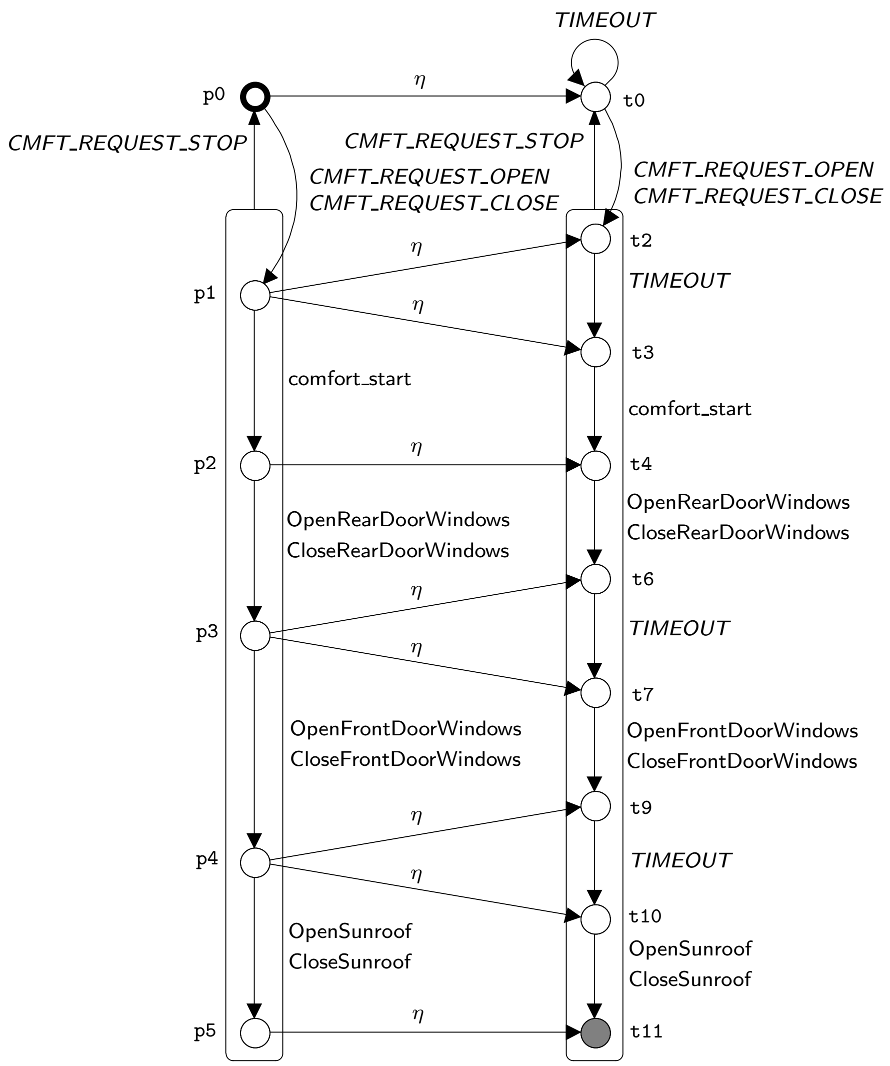

Figure 5.6: Automaton A_1''.

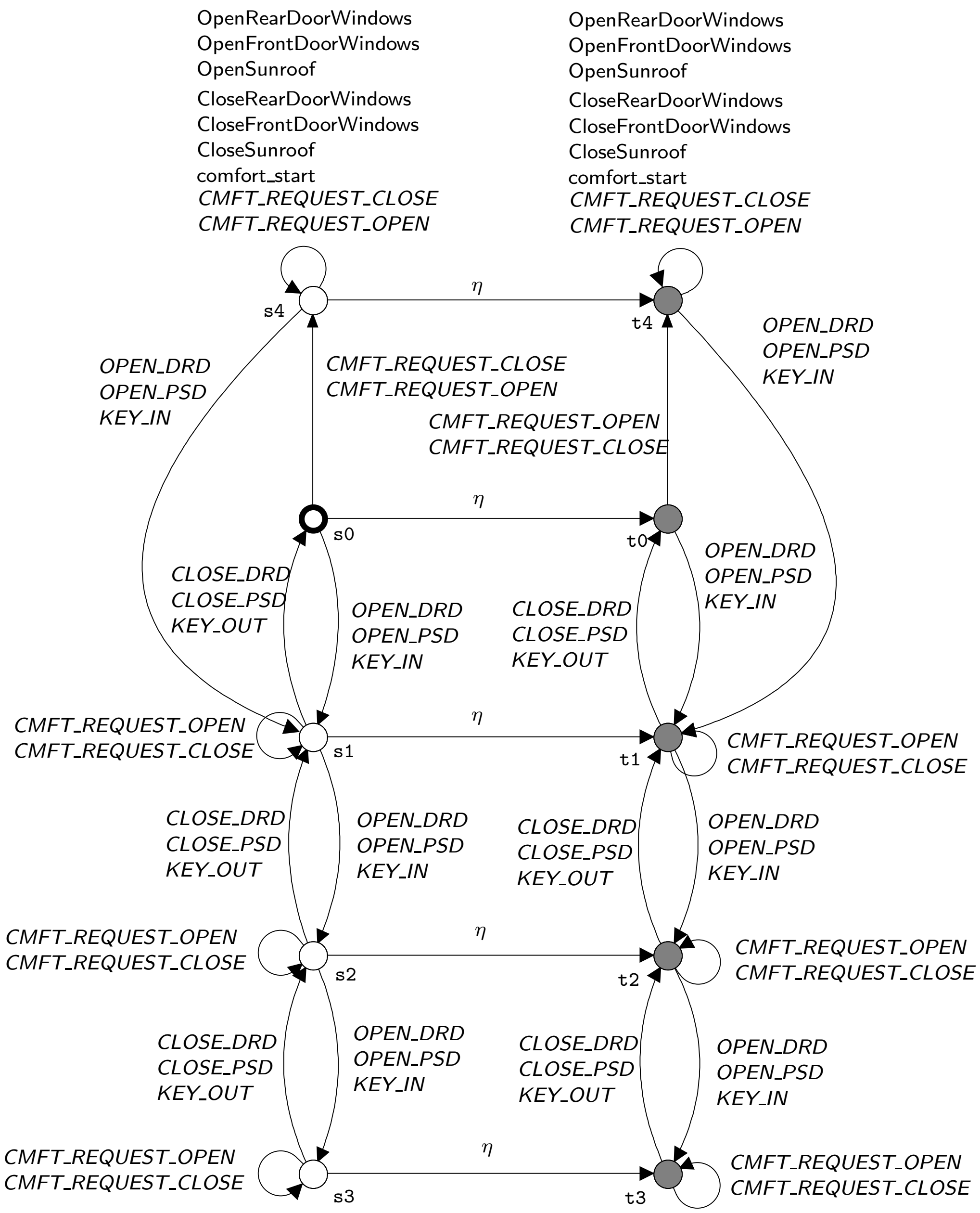

Figure 5.7: Automaton A_2.

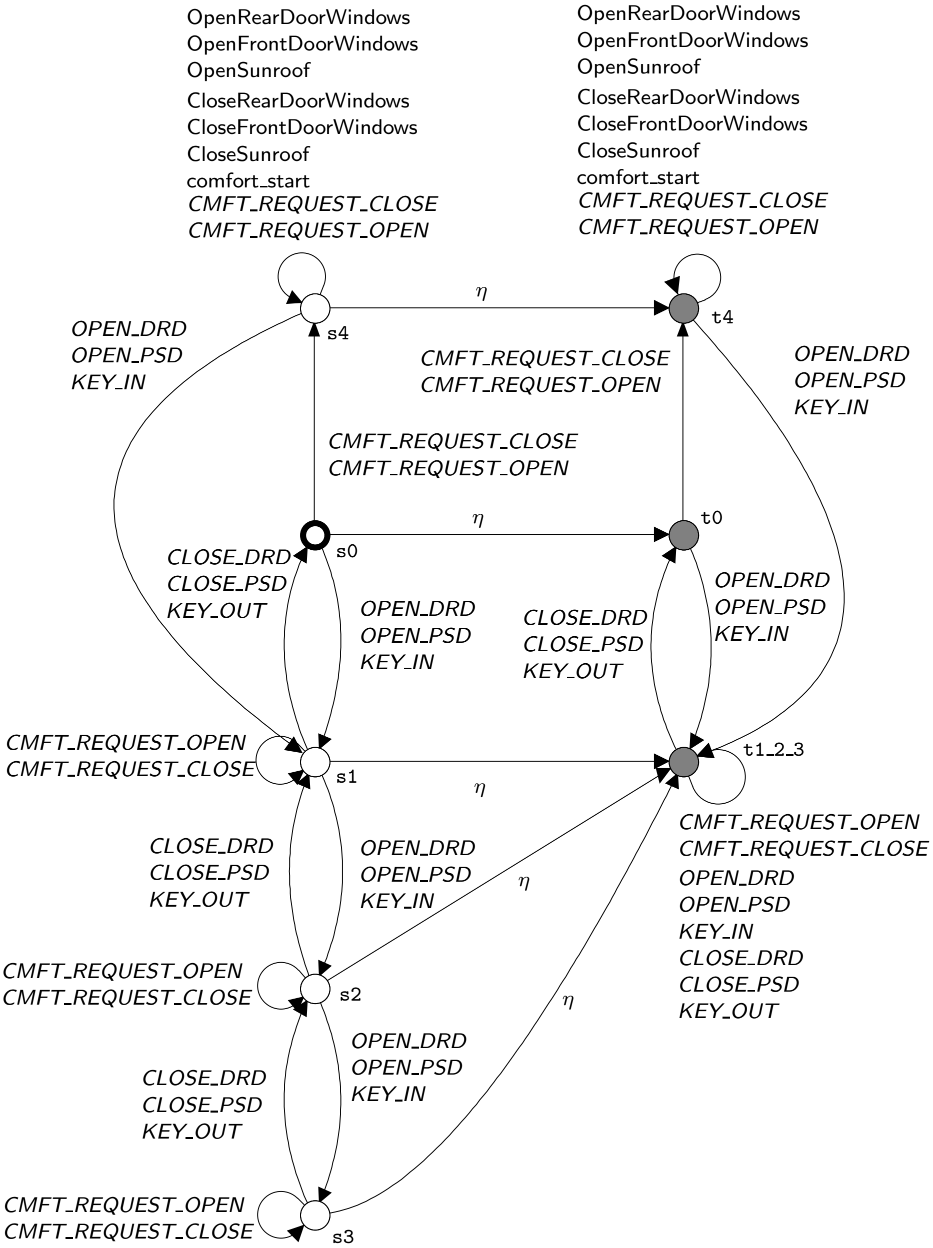

Figure 5.8: Automaton A_2'.

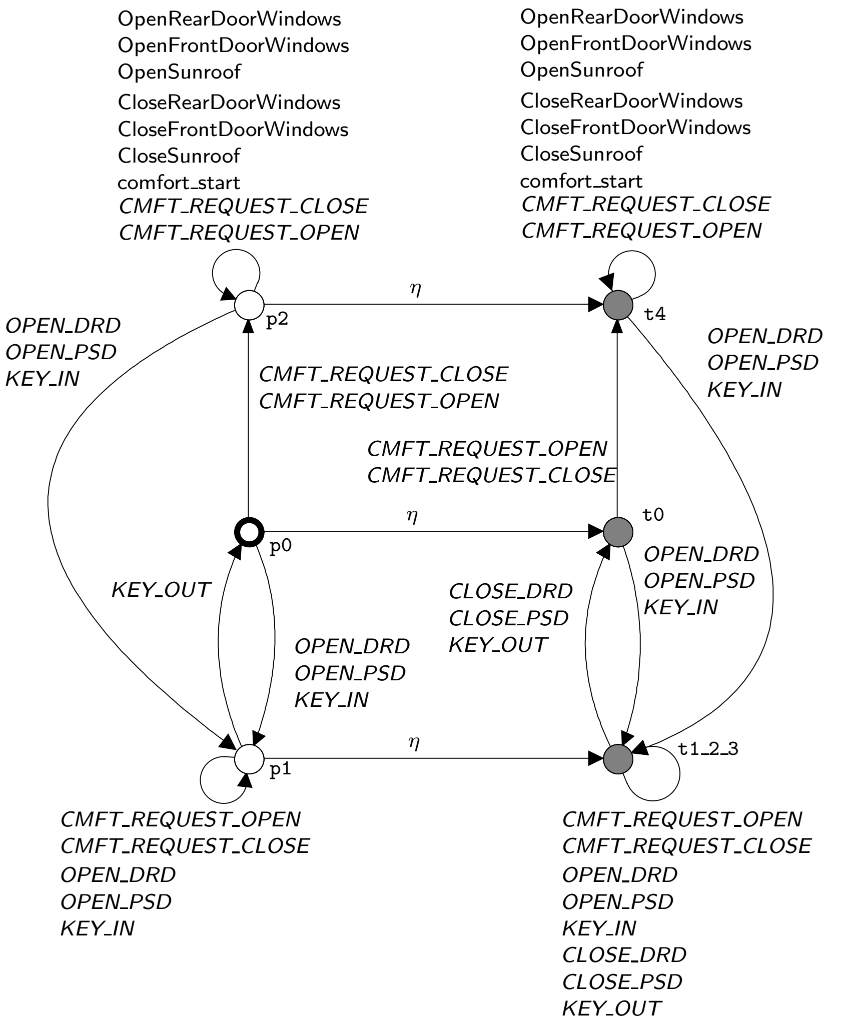

Figure 5.9: Automaton A_2''.

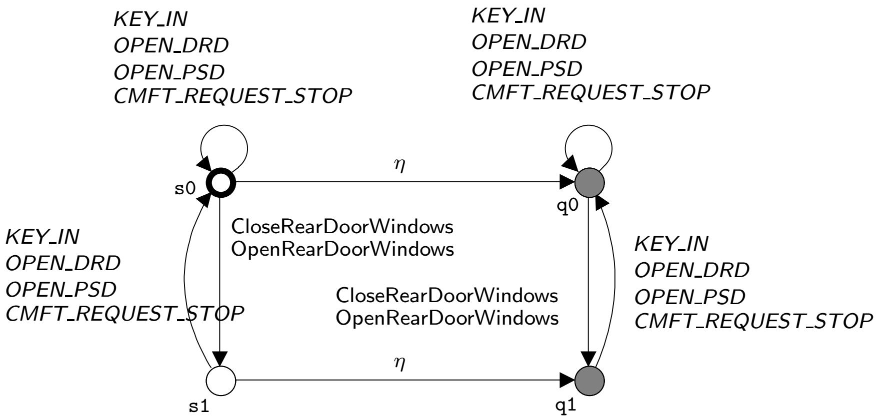

Figure 5.10: Automaton A_3''.

Non-deterministic η-partitioned Automata

Non-deterministic η-partitioned automata are used as abstractions for deterministic automata. In order to check if the synchronous product of deterministic automata is Ξ-nonblocking, we check if the synchronous product of η-partitioned automata is (η, Ξ)-nonblocking. The size of the η-partitioned automata can be reduced by applying reduction operations defined later. Since the number of states of a synchronous product depends critically on the number of states of the constituent automata, we are able to verify much larger systems.

Figure 5.6 on page 73 shows a typical η-partitioned automaton. It consists of two parts: the *reachability* and the *coreachability* part. The reachability part is the subautomaton on the left side. This part is used to represent the reachability information for the coreachability part. Note that the event η is enabled at each state of the reachability part. The coreachability part is the subautomaton on the right side. This part is used to memorise Ξ-controlled paths to marked states. Note that the event η is disabled at each state. Initial states are located in the reachability part. Transitions labelled with the event η start at a state of the reachability part and end in a state of the coreachability part.

Definition 5.2.7 (η-partitioned automaton)
Let A be an automaton, and let $\eta \in \Sigma_A$. A is said to be η-partitioned *if, for all $p, q \in Q_A$,*

(EP1) $p \in J_A$ *implies* $\eta \in \textsc{Elig}_A(p)$,

(EP2) $(p, \sigma, q) \in \Delta_A$ *for some* $\sigma \in \Sigma_A \backslash \{\eta\}$ *implies*
$\eta \in \textsc{Elig}_A(p)$ *if and only if* $\eta \in \textsc{Elig}_A(q)$, *and*

(EP3) $(p, \eta, q) \in \Delta_A$ *implies* $\eta \notin \textsc{Elig}_A(q)$.

The automaton in figure 5.6 satisfies properties (EP1), (EP2), and (EP3). Event η is eligible at the initial state `p0` (EP1), we are not able to leave the (co)reachability part by following transitions not labelled with the event η (EP2), and every transition labelled with event η ends in the coreachability part (EP3).

An η-partitioned automaton A accepts words that contain at most one occurrence of the event η, i.e. given $t \in \Sigma_A^*$,

$$(p, t, q) \in \Delta_A \quad \text{implies} \quad |t|_\eta \leq 1. \tag{5.6}$$

Starting from a state in which the event η is eligible and following transitions not labelled with η, we reach a state in which event η is still eligible, i.e. for $p, q \in Q_A$, and $s \in (\Sigma_A \backslash \{\eta\})^*$ it can be easily proven (by induction on the length of s) that

$$(p, s, q) \in \Delta_A \quad \text{implies} \quad \eta \in \textsc{Elig}_A(p) \Leftrightarrow \eta \in \textsc{Elig}_A(q). \tag{5.7}$$

The synchronous product of two η-partitioned automata is again η-partitioned. This is an immediate consequence of the following more general lemma.

Lemma 5.2.18
Let A be an η-partitioned automaton, and let B be an automaton over the same alphabet that satisfies properties (EP1) and (EP2) of definition 5.2.7. Then $A \,\|\, B$ is an η-partitioned automaton.

Proof. Let A be an η-partitioned automaton over the alphabet Σ, and let B be an automaton over Σ that satisfies properties (EP1) and (EP2) of definition 5.2.7. We want to show that $A \,\|\, B$ satisfies properties (EP1), (EP2), and (EP3) of definition 5.2.7. Let $p_1, p_2 \in Q_A$, and let $q_1, q_2 \in Q_B$.

(EP1) Let $(p_1, q_1) \in J_{A\|B}$. Then we have $p_1 \in J_A$ and $q_1 \in J_B$, and, by property (EP1) for A and B we have $\eta \in \textsc{Elig}_A(p_1)$ and $\eta \in \textsc{Elig}_B(q_1)$, which implies $\eta \in \textsc{Elig}_{A\|B}(p_1, q_1)$.

(EP2) Assume that $((p_1, q_1), \sigma, (p_2, q_2)) \in \Delta_{A\|B}$ for some $\sigma \in \Sigma \backslash \{\eta\}$, i.e. $(p_1, \sigma, p_2) \in \Delta_A$ and $(q_1, \sigma, q_2) \in \Delta_B$. Then

$$\begin{aligned}
\eta \in \textsc{Elig}_{A\|B}(p_1, q_1) &\Leftrightarrow \eta \in \textsc{Elig}_A(p_1) \text{ and } \eta \in \textsc{Elig}_B(q_1) \\
&\Leftrightarrow \eta \in \textsc{Elig}_A(p_2) \text{ and } \eta \in \textsc{Elig}_B(q_2) \\
&\Leftrightarrow \eta \in \textsc{Elig}_{A\|B}(p_2, q_2)
\end{aligned}$$

by property (EP2) for A and B.

(EP3) Let $((p_1, q_1), \eta, (p_2, q_2)) \in \Delta_{A\|B}$. Then we have $(p_1, \eta, p_2) \in \Delta_A$ and, since A is η-partitioned, $\eta \notin \textsc{Elig}_A(p_2)$. Hence, we have $\eta \notin \textsc{Elig}_{A\|B}(p_2, q_2)$. $\square$

In order to define (η, Ξ)-nonblocking automata, we first provide the concept of η-reachable states. An η-reachable state is a state that can be reached by a string ending with event η. Note that such a state must be in the coreachability part of an η-partitioned automaton.

Definition 5.2.8 (η-reachable state)
Let $A = (\Sigma, Q, \Delta, J, M)$ be an η-partitioned automaton. A state $q \in Q$ is said to be η-reachable *in A if there exists $s \in \Sigma^*$ such that $(J, s\eta, q) \in \Delta$.*

We are now able to define the property to be (η, Ξ)-nonblocking. An η-partitioned automaton is (η, Ξ)-nonblocking if each η-reachable state is Ξ-coreachable, i.e. if the state is origin of a Ξ-controlled path to a marked state. Note that the reachability part of the η-partitioned automaton is used for reachability information. The coreachability part is used to record Ξ-controlled paths to marked states.

Definition 5.2.9 ((η, Ξ)-nonblocking automaton)
Let A be an η-partitioned automaton, and let $\Xi \subseteq \Sigma_A$ such that $\eta \notin \Xi$. A is said to be (η, Ξ)-nonblocking *if each η-reachable state in A is Ξ-coreachable in A.*

All reductions defined later will be shown to preserve this property. We will also show that the synchronous product of several automata is (η, Ξ)-nonblocking if and only if the synchronous product of these automata, in which one of the automata is replaced by the reduced one, is (η, Ξ)-nonblocking.

Recall that a DES is Ξ-nonblocking if every word of its language can be continued by a Ξ-controlled word to a word of the marked language. Thus, every reachable state of a deterministic automaton representing a Ξ-nonblocking DES is the origin of a Ξ-controlled path to a marked state. If a Ξ-nonblocking deterministic automaton is η-partitioned, then it is (η, Ξ)-nonblocking, because every η-reachable state is reachable.

Corollary 5.2.19
Let A be a deterministic η-partitioned automaton over the alphabet Σ, and let $\Xi \subseteq \Sigma$. If A is Ξ-nonblocking, then A is (η, Ξ)-nonblocking.

Proof. This is an immediate consequence of lemma 5.1.9. □

An Invariant

We want to apply the incremental algorithm introduced here to a Ξ-terminating and Ξ-confluent system. This can be exploited when reduction operations are defined. We are able to provide more efficient simplifications of the η-partitioned automata, enabling us to verify larger systems. The problem is that the reduction operations do not preserve Ξ-termination and Ξ-confluence. Moreover, it is not clear how these properties are defined for non-deterministic η-partitioned automata. In this section, we introduce an invariant, called (η, Ξ)-nice, which is preserved by the reduction operations. Each Ξ-terminating and Ξ-confluent system satisfies this invariant.

We start by introducing live (η, Ξ)-supervisors. Live (η, Ξ)-supervisors use state information of a non-deterministic automaton to decide which events are enabled. Furthermore, the reachability part of an η-partitioned automaton is not restricted by such a supervisor.

Definition 5.2.10 (live (η, Ξ)-supervisor)
Let $A = (\Sigma, Q, \Delta, J, M)$ be an η-partitioned automaton, and let $\Xi \subseteq \Sigma \backslash \{\eta\}$. A live (η, Ξ)-supervisor *for A is a mapping $\Phi\colon Q \to 2^{\Sigma}$ such that we have for all states $q \in Q$*

- **(LS1)** $\Sigma \backslash \Xi \subseteq \Phi(q)$,
- **(LS2)** $\eta \in \textsc{Elig}_A(q)$ *implies* $\Phi(q) = \Sigma$, *and*
- **(LS3)** $\Phi(q) \cap \textsc{Elig}_A(q) \cap \Xi = \emptyset$ *implies* $\textsc{Elig}_A(q) \cap \Xi = \emptyset$.

The controlled system Φ/A (Φ controlling A) is defined as

$$\Phi/A = (\Sigma, Q, \Delta_{\Phi/A}, J, M),$$

for which $(p, \sigma, q) \in \Delta_{\Phi/A}$ if and only if $(p, \sigma, q) \in \Delta$ and $\sigma \in \Phi(p)$.

The set of eligible events at any state in the controlled system is the intersection of the eligible events in the uncontrolled system and the events enabled by the supervisor. Given an η-partitioned automaton A and a live (η, Ξ)-supervisor Φ for A, we have

$$\textsc{Elig}_{\Phi/A}(q) = \textsc{Elig}_A(q) \cap \Phi(q) \tag{5.8}$$

for each state $q \in Q_A$.

At a given state, a live (η, Ξ)-supervisor may not disable all output events if there is at least one event in Ξ eligible in the uncontrolled system. Therefore, the set of eligible output events is empty in the closed-loop system if and only if it is empty in the uncontrolled system. This is proven in the following lemma.

Lemma 5.2.20
Let A be an η-partitioned automaton, let Φ be a live (η, Ξ)-supervisor for A, and let $q \in Q_A$. Then we have

$$\textsc{Elig}_A(q) \cap \Xi = \emptyset \quad \textit{if and only if} \quad \textsc{Elig}_{\Phi/A}(q) \cap \Xi = \emptyset.$$

Proof. Let A be an η-partitioned automaton and let Φ be a live (η, Ξ)-supervisor for A. Let $q \in Q_A$. By equation (5.8), we obtain that $\textsc{Elig}_{\Phi/A}(q) \cap \Xi = \emptyset$ if and only if $\textsc{Elig}_A(q) \cap \Phi(q) \cap \Xi = \emptyset$. By property (LS3) of definition 5.2.10, this is the case if and only if $\textsc{Elig}_A(q) \cap \Xi = \emptyset$. $\square$

Thus, a path in A that is also a path in Φ/A is Ξ-controlled in A if and only if it is Ξ-controlled in Φ/A.

Now we show that a live (η, Ξ)-supervisor does not restrict words starting from an initial state that end with the event η. This means that the reachability part of an η-partitioned automaton is not modified by a live (η, Ξ)-supervisor.

Lemma 5.2.21
Let A be an η-partitioned automaton, and let Φ be a live (η, Ξ)-supervisor for A. For all $q \in Q_A$ and $s \in \Sigma_A^*$ we have

$$(J_A, s\eta, q) \in \Delta_A \quad \textit{implies} \quad (J_{\Phi/A}, s\eta, q) \in \Delta_{\Phi/A}.$$

Proof. Let A be an η-partitioned automaton, let Φ be a live (η, Ξ)-supervisor for A, and let $(J_A, s\eta, q) \in \Delta_A$. Then there exists a path $\pi = q_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} q_n \xrightarrow{\eta} q$ in A such that $q_0 \in J_A$. By equations (5.6) and (5.7) on page 77, we obtain $|\sigma_1 \ldots \sigma_n|_\eta = 0$ and, for all $k \in \{0, \ldots, n\}$, $\eta \in \textsc{Elig}_A(q_k)$, so that $\Phi(q_k) = \Sigma_A$ by (LS2). Hence, for all $i \in \{0, \ldots, n-1\}$, $(q_i, \sigma_{i+1}, q_{i+1}) \in \Delta_{\Phi/A}$ and $(q_n, \eta, q) \in \Delta_{\Phi/A}$. Since $J_A = J_{\Phi/A}$, this proves that $(J_{\Phi/A}, s\eta, q) \in \Delta_{\Phi/A}$. $\square$

If the closed-loop system of an η-partitioned automaton and a live (η, Ξ)-supervisor is (η, Ξ)-nonblocking, then the uncontrolled system, i.e. the η-partitioned automaton itself, is also (η, Ξ)-nonblocking.

Proposition 5.2.22
Let A be an η-partitioned automaton, and let Φ be a live (η, Ξ)-supervisor for A. If Φ/A is (η, Ξ)-nonblocking, then A is (η, Ξ)-nonblocking.

Proof. Let A be an η-partitioned automaton, and let Φ be a live (η, Ξ)-supervisor for A such that Φ/A is (η, Ξ)-nonblocking. Furthermore, let $q \in Q_A$ be η-reachable in A. By lemma 5.2.21 we obtain that q is η-reachable in Φ/A. Since Φ/A is (η, Ξ)-nonblocking, q is Ξ-coreachable in Φ/A, i.e. there exists a Ξ-controlled and marked path π with origin q in Φ/A. Then π is a marked path in A by definition of Φ/A. By lemma 5.2.20, we obtain that π is Ξ-controlled in A. This proves that q is Ξ-coreachable in A. $\square$

The converse implication in proposition 5.2.22 is not always satisfied. We use it as an invariant for the reduction operations introduced later. It states that applying a live (η, Ξ)-supervisor to an automaton preserves the property of being (η, Ξ)-nonblocking. Among others, this enables us to define a reduction operation that simply applies a live (η, Ξ)-supervisor, disabling as much events as possible.

Definition 5.2.11 ((η, Ξ)-nice)
Let A be an η-partitioned automaton, and let $\Xi \subseteq \Sigma_A \backslash \{\eta\}$. A is said to be (η, Ξ)-nice *if*

(N1) *A is (η, Ξ)-nonblocking implies that Φ/A is (η, Ξ)-nonblocking for each live (η, Ξ)-supervisor Φ for A.*

All reduction operations defined later will be shown to preserve (η, Ξ)-nice automata. We will show that if the synchronous product of several automata is (η, Ξ)-nice, then the synchronous product of these automata in which one of the automata is reduced is also (η, Ξ)-nice.

From Deterministic Automata to η-partitioned Automata

We want to represent a deterministic automaton by an η-partitioned automaton in such a way that the deterministic automaton is Ξ-nonblocking if and only if the corresponding η-partitioned automaton is (η, Ξ)-nonblocking. In order to do this, we introduce the automaton A^η_Σ. The synchronous product of A^η_Σ and a deterministic automaton results in an η-partitioned automaton satisfying the desired property. Furthermore, the resultant η-partitioned automaton is (η, Ξ)-nice if the original automaton is Ξ-terminating and Ξ-confluent.

We start by defining the automaton A^η_Σ. A graph representation of this automaton is given in figure 5.11. Note that only relevant events are shown.

Definition 5.2.12 (A^η_Σ)
Let Σ be an alphabet, and let $\eta \in \Sigma$. Define the automaton

$$A^\eta_\Sigma = (\Sigma, \{\mathtt{s0}, \mathtt{s1}\}, \Delta, \{\mathtt{s0}\}, \{\mathtt{s1}\}),$$

such that $\Delta = \{(\mathtt{s0}, \eta, \mathtt{s1})\} \cup \{\, (\mathtt{s0}, \sigma, \mathtt{s0}) \mid \sigma \in \Sigma\backslash\{\eta\}\,\} \cup \{\, (\mathtt{s1}, \sigma, \mathtt{s1}) \mid \sigma \in \Sigma\backslash\{\eta\}\,\}$.

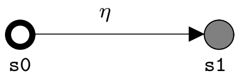

Figure 5.11: Automaton A^{η}_{Σ}.

Consider the example in figure 5.3 again. Let Σ be the set of all events as listed in figure 5.2 together with the new event η. In a first step, the deterministic automata shown in figure 5.1 are doubled by building the synchronous product of each deterministic automaton and A^{η}_{Σ}. The synchronous product A_1 of automaton `comfort_function` and A^{η}_{Σ} is shown in figure 5.4. The synchronous product A_2 of automaton `comfort_condition` and A^{η}_{Σ} is given in figure 5.7. These are η-partitioned automata.

Next, we prove that we always obtain an η-partitioned automaton when we compute a synchronous product in this way. First we show that automaton A^{η}_{Σ} is deterministic, that the language accepted by A^{η}_{Σ} is the set of all strings over the alphabet Σ that contain at most one occurrence of event η, and that the marked language is the set of all strings over Σ that contain exactly one occurrence of event η.

Lemma 5.2.23
Let Σ be an alphabet, and let $\eta \in \Sigma$.
A^{η}_{Σ} is a deterministic η-partitioned automaton such that

$$\begin{aligned} \mathcal{L}\,(A^{\eta}_{\Sigma}) &= \{\, s \in \Sigma^* \mid |s|_\eta \leq 1 \,\}, \\ \mathcal{L}_m\,(A^{\eta}_{\Sigma}) &= \{\, s \in \Sigma^* \mid |s|_\eta = 1 \,\}. \end{aligned}$$

Proof. This follows immediately from definition 5.2.12. □

Now we can show that the synchronous product of A^{η}_{Σ} and a deterministic automaton results in a deterministic η-partitioned automaton.

Lemma 5.2.24
Let G be a deterministic automaton over the alphabet Σ, and let $\eta \in \Sigma$ be not relevant for G. Then $A^{\eta}_{\Sigma} \,\|\, G$ is a deterministic η-partitioned automaton.

Proof. Let G be a deterministic automaton over the alphabet Σ, and let $\eta \in \Sigma$ be not relevant for G. Because A^{η}_{Σ} and G are deterministic, $A^{\eta}_{\Sigma} \,\|\, G$ is also deterministic. Since η is not relevant for G, the automaton G satisfies properties (EP1) and (EP2) of definition 5.2.7 and, since A^{η}_{Σ} is η-partitioned by lemma 5.2.23, we obtain that $A^{\eta}_{\Sigma} \,\|\, G$ is η-partitioned by lemma 5.2.18. □

Now we are able to prove that a deterministic automaton is Ξ-nonblocking if and only if the corresponding η-partitioned automaton, derived by building the synchronous product with A^{η}_{Σ}, is (η, Ξ)-nonblocking.

Proposition 5.2.25
Let G be a deterministic automaton over the alphabet Σ, let $\Xi \subseteq \Sigma$, and let $\eta \in \Sigma$ be not relevant for G.
G is Ξ-nonblocking if and only if $A^{\eta}_{\Sigma} \parallel G$ is (η, Ξ)-nonblocking.

Proof. Let G be a deterministic automaton over the alphabet Σ, let $\Xi \subseteq \Sigma$, let $\eta \in \Sigma$ be not relevant for G, and let $A = A^{\eta}_{\Sigma} \parallel G$.

(Only If) Let G be Ξ-nonblocking. We want to show that A is (η, Ξ)-nonblocking. Let $q \in Q_A$ be η-reachable. Then there exists $s \in \Sigma^*$ such that $s\eta \in \mathcal{L}(A) = \mathcal{L}(A^{\eta}_{\Sigma}) \cap \mathcal{L}(G)$. Since G is Ξ-nonblocking, there exists a Ξ-controlled string t' in $\mathcal{L}(G)/s\eta$ such that $s\eta t' \in \mathcal{L}_m(G)$. Since η is not relevant for G, we have that $t := P_{\Sigma\setminus\{\eta\}}(t')$ is Ξ-controlled in $\mathcal{L}(G)/s\eta$ such that $s\eta t \in \mathcal{L}_m(G)$. Since $s\eta \in \mathcal{L}(A^{\eta}_{\Sigma})$, we have $|s\eta|_{\eta} = 1$ by lemma 5.2.23. Thus, we have $|s\eta t|_{\eta} = 1$. Then we also have $s\eta t \in \mathcal{L}(A^{\eta}_{\Sigma})$, which implies $s\eta t \in \mathcal{L}(A)$. String t is also Ξ-controlled in $\mathcal{L}(A)/s\eta$ because we have $\textsc{Elig}_{\mathcal{L}(A)}(w) \cap \Xi = \textsc{Elig}_{\mathcal{L}(G)}(w) \cap \Xi$ for each $w \in \mathcal{L}(A)$. By lemma 5.1.9, we obtain that q is Ξ-coreachable in A. This proves that A is (η, Ξ)-nonblocking.

(If) Let A be (η, Ξ)-nonblocking, and let $s \in \mathcal{L}(G)$. We need to construct a Ξ-controlled string t in $\mathcal{L}(G)/s$ such that $st \in \mathcal{L}_m(G)$. Consider $s' = P_{\Sigma\setminus\{\eta\}}(s)$. Since η is not relevant for G, we have $\mathcal{L}(G)/s = \mathcal{L}(G)/s'$ and $\mathcal{L}_m(G)/s = \mathcal{L}_m(G)/s'$. Thus, it suffices to find a Ξ-controlled string t in $\mathcal{L}(G)/s'$ such that $s't \in \mathcal{L}_m(G)$. We have $|s'|_{\eta} = 0$ and therefore $s' \in \mathcal{L}(A^{\eta}_{\Sigma})$. Now, $s' \in \mathcal{L}(A^{\eta}_{\Sigma}) \cap \mathcal{L}(G) = \mathcal{L}(A)$. Since A is η-partitioned by lemma 5.2.24, we have $s'\eta \in \mathcal{L}(A)$. Since A is (η, Ξ)-nonblocking, there exists a Ξ-controlled string $t \in \mathcal{L}(A)/s'\eta$ such that $s'\eta t \in \mathcal{L}_m(A) \subseteq \mathcal{L}_m(G)$. Since η is not relevant for G, we have $s't \in \mathcal{L}_m(G)$. It suffices to show that t is Ξ-controlled in $\mathcal{L}(G)/s' = \mathcal{L}(G)/s'\eta$. But this follows since t is Ξ-controlled in $\mathcal{L}(A)/s'\eta$ and by $\textsc{Elig}_{\mathcal{L}(A)}(w) \cap \Xi = \textsc{Elig}_{\mathcal{L}(G)}(w)$ for each $w \in \mathcal{L}(A)$. $\square$

In order to illustrate this result, consider the system

$$G_{\text{comfort}} = \texttt{comfort\_function} \parallel \texttt{comfort\_conditions} \parallel \texttt{comfort\_stop}$$

shown in figure 5.1 on page 66, with the alphabet Σ defined in figure 5.2 on page 67, and let

$$\begin{aligned} A_1 &= A^{\eta}_{\Sigma} \parallel \texttt{comfort\_function}, \\ A_2 &= A^{\eta}_{\Sigma} \parallel \texttt{comfort\_conditions}, \\ A_3 &= A^{\eta}_{\Sigma} \parallel \texttt{comfort\_stop}. \end{aligned}$$

By proposition 5.2.25, in order to check if G_{comfort} is Ξ-nonblocking, we can also check if

$$A_1 \parallel A_2 \parallel A_3 \cong A_\Sigma^\eta \parallel G_{\text{comfort}}$$

is (η, Ξ)-nonblocking. This justifies the first step performed according to figure 5.3 on page 68.

Next, we show that if a deterministic automaton is Ξ-terminating and Ξ-confluent, then the corresponding η-partitioned automaton is (η, Ξ)-nice. Thus, if we check a Ξ-terminating and Ξ-confluent system, then the synchronous product of the corresponding η-partitioned automata is (η, Ξ)-nice, and the reduction operations defined later can exploit this fact.

We first prove the following lemma, which provides conditions under which a deterministic (η, Ξ)-nonblocking automaton is also Ξ-nonblocking.

Lemma 5.2.26
Let $A = (\Sigma, Q, \delta, q_0, M)$ be a deterministic η-partitioned automaton, and let $\Xi \subseteq \Sigma$ such that $\eta \notin \Xi$. If

(BB1) *A is Ξ-terminating,*

(BB2) *A is (η, Ξ)-nonblocking,*

(BB3) *$s_1\eta s_2 \in \mathcal{L}(A)$ if and only if $s_1 s_2 \eta \in \mathcal{L}(A)$, for all $s_1, s_2 \in \Sigma^*$, and*

(BB4) *$\delta(q_0, s_1\eta s_2) = \delta(q_0, s_1 s_2 \eta)$, for all $s_1, s_2 \in \Sigma$ such that $s_1\eta s_2 \in \mathcal{L}(A)$,*

then A is Ξ-nonblocking.

Proof. Let $A = (\Sigma, Q, \delta, q_0, M)$ be a deterministic η-partitioned automaton, and let $\Xi \subseteq \Sigma$ such that $\eta \notin \Xi$. Furthermore, let the properties (BB1), (BB2), (BB3), and (BB4) be satisfied, and let $s \in \mathcal{L}(A)$. We consider two cases.

Case 1: $|s|_\eta = 0$. Consider $t' \in \Xi^*$ such that st' is Ξ-complete in $\mathcal{L}(A)$; such a string exists by property (BB1). We have $\eta \in \textsc{Elig}_{\mathcal{L}(A)}(st')$, since A is η-partitioned. Then we have $st'\eta \in \mathcal{L}(A)$. Let $q = \delta(q_0, st'\eta)$. Then q is η-reachable in A. By property (BB2), q is Ξ-coreachable in A. By lemma 5.1.9, there exists a Ξ-controlled string t in $\mathcal{L}(A)/st'\eta$ such that $st'\eta t \in \mathcal{L}_m(A)$. Since $t' \in \Xi^*$ and st' is Ξ-complete in $\mathcal{L}(A)$, we have that $t'\eta t$ is Ξ-controlled in $\mathcal{L}(A)/s$. Therefore, A is Ξ-nonblocking.

Case 2: $|s|_\eta > 0$, i.e. there exist $s_1, s_2 \in (\Sigma\backslash\{\eta\})^*$ such that $s = s_1\eta s_2$, since A is η-partitioned. By property (BB3), $s_1 s_2 \eta \in \mathcal{L}(A)$. Let $q = \delta(q_0, s)$. By property (BB4), $q = \delta(q_0, s_1\eta s_2) = \delta(q_0, s_1 s_2 \eta)$, i.e. q is η-reachable in A. By property (BB2), q is Ξ-coreachable. There exists a Ξ-controlled string t in $\mathcal{L}(A)/s_1 s_2 \eta$ such that $s_1 s_2 \eta t \in \mathcal{L}_m(A)$, by lemma 5.1.9. Now we have $\mathcal{L}(A)/s = \mathcal{L}(A)/s_1\eta s_2 = \mathcal{L}(A)/s_1 s_2 \eta$, i.e. $t \in \mathcal{L}(A)/s$ and $st \in \mathcal{L}_m(A)$. Hence, A is Ξ-nonblocking. □

Now we can prove the desired result.

Proposition 5.2.27
Let G be a deterministic automaton over the alphabet Σ, let $\Xi \subseteq \Sigma$, and let $\eta \in \Sigma \backslash \Xi$ be not relevant for G.
If G is Ξ-terminating and Ξ-confluent, then $A^{\eta}_{\Sigma} \,\|\, G$ is (η, Ξ)-nice.

Proof. Let G be a deterministic automaton over the alphabet Σ, let $\eta \in \Sigma$ be not relevant for G, and let $\Xi \subseteq \Sigma \backslash \{\eta\}$. Furthermore, let G be Ξ-terminating and Ξ-confluent, let $A = A^{\eta}_{\Sigma} \,\|\, G$, and write $A = (\Sigma, Q, \delta, q_0, M)$.

First we claim that A is Ξ-terminating and Ξ-confluent. Since G is Ξ-terminating we have that A is Ξ-terminating by lemma 5.2.3. Then, by lemma 5.2.6, in order to show that A is Ξ-confluent it suffices to show that each pair $(\xi_1, \xi_2) \in \Xi \times \Xi$ is Ξ-joinable in A. Let $\xi_1, \xi_2 \in \Xi$. Since G is Ξ-confluent, (ξ_1, ξ_2) is Ξ-joinable in G. Furthermore, (ξ_1, ξ_2) is $\emptyset$-joinable in A^{η}_{Σ}. Since Ξ is not relevant for A^{η}_{Σ}, (ξ_1, ξ_2) is Ξ-joinable in A by proposition 5.2.12.

Now let A be (η, Ξ)-nonblocking. Next, we claim that A is Ξ-nonblocking. Since η is not relevant for G and by definition 5.2.12, we obtain that $s_1 \eta s_2 \in \mathcal{L}(A)$ if and only if $s_1 s_2 \eta \in \mathcal{L}(A)$, for all $s_1, s_2 \in \Sigma^*$, and $\delta(q_0, s_1 \eta s_2) = \delta(q_0, s_1 s_2 \eta)$, for all $s_1, s_2 \in \Sigma^*$ such that $s_1 s_2 \eta \in \mathcal{L}(A)$. By lemma 5.2.26, we obtain that A is Ξ-nonblocking.

Now let Φ be a live (η, Ξ)-supervisor for A. Define

$$S\colon \Sigma^* \to 2^{\Sigma}; s \mapsto \begin{cases} \Phi(\delta(q_0, s)), & \text{if } \delta(q_0, s) \text{ is defined;} \\ \Sigma, & \text{otherwise.} \end{cases}$$

S is well-defined, since A is deterministic. Furthermore, S is a live Ξ-supervisor for A such that $\mathcal{L}(S/A) = \mathcal{L}(\Phi/A)$ and $\mathcal{L}_m(S/A) = \mathcal{L}_m(\Phi/A)$. Since A is Ξ-terminating, Ξ-confluent and Ξ-nonblocking, we have by theorem 4.5.2 that S/A is Ξ-nonblocking. Since $\mathcal{L}(S/A) = \mathcal{L}(\Phi/A)$ and $\mathcal{L}_m(S/A) = \mathcal{L}_m(\Phi/A)$ this proves that Φ/A is Ξ-nonblocking and thus, by corollary 5.2.19, (η, Ξ)-nonblocking. □

In the example in figure 5.3 we consider a Ξ-terminating and Ξ-confluent system. Let A_1, A_2, A_3 be the derived η-partitioned automata as defined above. By applying proposition 5.2.27 we obtain that

$$A_1 \,\|\, A_2 \,\|\, A_3 \cong A^{\eta}_{\Sigma} \,\|\, G_{\text{comfort}}$$

is (η, Ξ)-nice.

In the following sections, we introduce reduction operations that can be applied to such η-partitioned automata. Reduction operations (R1), (R2), and (R3) reduce the coreachability part, and reduction operation (R4) is responsible for the reachability part. All reduction operations exploit the fact that certain events, called *hidden events* and denoted by Υ, are only relevant for the automaton to be reduced and are irrelevant for all other automata of the system. Reduction operation (R1) removes transitions labelled with output events starting in a state in which a hidden output event is eligible. Reduction operation (R2) considers states in which exactly one output event is eligible. If this event is hidden, the state can be removed. Hidden input events are considered in reduction operation (R3). An equivalence relation on states is defined and equivalent states are merged. Projections to non-hidden events are considered in reduction operation (R4), in which the reachability part is reduced.

(R1) Removing Transitions

In this section, we introduce reduction operation (R1), which deletes certain transitions in the coreachability part of an η-partitioned automaton. Transitions labelled with output events that originate from a state in which a hidden output event is eligible are removed.

As a motivating example, consider once more automaton `comfort_function` in figure 5.1 on page 66. Assume we want to know if the synchronous product of automata `comfort_function` and `comfort_stop` in figure 5.1 is Ξ-nonblocking. Recall that Ξ is the set of output events defined in figure 5.2. The event set

$$\Upsilon \;=\; \{\mathsf{CloseFrontDoorWindows}, \mathsf{OpenSunroof}, \mathsf{CloseSunroof}\}$$

is hidden for automaton `comfort_stop`; that is, Υ is only relevant for automaton `comfort_function`.

First we compute the corresponding η-partitioned automaton A_1 in figure 5.4 as explained on page 81. Here, the hidden event CloseFrontDoorWindows is eligible in state `t7`. The corresponding Ξ-transition remains enabled when the synchronous product with automaton `comfort_stop` is computed. Assuming a Ξ-confluent and Ξ-terminating system, this transition can be taken without losing the possibility to reach a marked state by means of Ξ-controlled paths, since all Ξ-paths starting from a common state finally reach the same target state. Therefore, all other transitions originating from state `t7` can be disabled.

Reduction operation (R1) removes only Ξ-transitions; the other transitions are removed by reduction operation (R2) introduced later. In automaton A_1, we can remove the transition labelled with event OpenFrontDoorWindows originating from state `t7`. If there is more than one hidden event eligible at a certain state, we can choose one and remove the others. Therefore, at state `t10` of automaton A_1, we may either remove the transition labelled with OpenSunroof or the transition labelled with CloseSunroof.

This idea is now formalised. First we define Υ-selective (η, Ξ)-supervisors. An Υ-selective (η, Ξ)-supervisor is a supervisor which only disables Ξ-transitions if there is a hidden event eligible. Furthermore, at least one of the hidden Υ-transitions must not be disabled, i.e. must be eligible in the closed-loop system.

Definition 5.2.13 (Υ-selective (η, Ξ)-supervisor)
Let A be an η-partitioned automaton, let Φ be a live (η, Ξ)-supervisor for A, and let $\Upsilon \subseteq \Xi$. Φ is said to be Υ-selective *for A if we have for all states $q \in Q_A$ that $\Phi(q) \cap \textsc{Elig}_A(q) \cap \Upsilon = \emptyset$ implies $\textsc{Elig}_A(q) \subseteq \Phi(q)$.*

Reduction operation (R1) is defined as the application of an Υ-selective (η, Ξ)-supervisor to the automaton to be reduced. In the example above, the mapping $\Phi \colon Q_{A_1} \to 2^{\Sigma_{A_1}}$ defined as

$$\Phi(x) = \begin{cases} \Sigma_{A_1} \backslash \{\mathsf{OpenFrontDoorWindows}\}, & \text{if } x = \mathtt{t7}; \\ \Sigma_{A_1} \backslash \{\mathsf{OpenSunroof}\}, & \text{if } x = \mathtt{t10}; \\ \Sigma_{A_1}, & \text{otherwise.} \end{cases}$$

is an Υ-selective (η, Ξ)-supervisor for A_1. The closed-loop system of A_1 and Φ is a possible result of reduction operation (R1) applied to A_1.

In the following, we show the correctness of this reduction operation. We show that a synchronous product of two η-partitioned automata is (η, Ξ)-nonblocking if and only if the synchronous product of these automata in which one of the automata is reduced by applying reduction operation (R1) is (η, Ξ)-nonblocking. Furthermore, the synchronous product in which one of the automata is reduced is (η, Ξ)-nice if the synchronous product of the original automata is (η, Ξ)-nice.

We start by proving that the closed-loop of a live (η, Ξ)-supervisor and an η-partitioned automaton is (η, Ξ)-nice if the automaton itself is (η, Ξ)-nice. Later we extend this result to the synchronous product of two automata.

Lemma 5.2.28
Let A be an η-partitioned automaton and Φ be a live (η, Ξ)-supervisor for A. If A is (η, Ξ)-nice, then Φ/A is (η, Ξ)-nice.

Proof. Let A be an η-partitioned automaton and let Φ be a live (η, Ξ)-supervisor for A. Let A be (η, Ξ)-nice. Assume that Φ/A is (η, Ξ)-nonblocking and let Ψ be a live (η, Ξ)-supervisor for Φ/A. We obtain that A is (η, Ξ)-nonblocking by proposition 5.2.22. Define

$$(\Psi \wedge \Phi) \colon Q_A \to 2^{\Sigma};\ (\Psi \wedge \Phi)(q) \mapsto \Psi(q) \cap \Phi(q).$$

It can be easily verified that $\Psi \wedge \Phi$ is a live (η, Ξ)-supervisor for A. Furthermore, we have $(\Psi \wedge \Phi)/A = \Psi/(\Phi/A)$, since

$$\begin{aligned} (p, \sigma, q) \in \Delta_{(\Psi \wedge \Phi)/A} \quad &\Leftrightarrow \quad \sigma \in \Psi(q),\ \sigma \in \Phi(q),\ \text{and } (p, \sigma, q) \in \Delta_A \\ &\Leftrightarrow \quad (p, \sigma, q) \in \Delta_{\Psi/(\Phi/A)}. \end{aligned}$$

Since A is (η, Ξ)-nice, $\Psi/(\Phi/A)$ is (η, Ξ)-nonblocking. $\square$

The following lemma shows that the closed-loop system of an (η, Ξ)-nice and (η, Ξ)-nonblocking automaton and a live (η, Ξ)-supervisor is (η, Ξ)-nonblocking. Furthermore, if the original automaton is (η, Ξ)-nice and the closed-loop system is (η, Ξ)-nonblocking, then the original automaton is (η, Ξ)-nonblocking. This proves that original and reduced automaton are in fact (η, Ξ)-nonblocking equivalent if the original automaton is (η, Ξ)-nice.

Lemma 5.2.29
Let A be an η-partitioned automaton, let Φ be a live (η, Ξ)-supervisor for A, and let A be (η, Ξ)-nice.
A is (η, Ξ)-nonblocking if and only if Φ/A is (η, Ξ)-nonblocking.

Proof. The statement follows directly from proposition 5.2.22 and by the definition of (η, Ξ)-nice. ☐

So far, we have only considered a single automaton. Since we want to apply the introduced reduction to an automaton that is part of the synchronous product of several automata, we provide results regarding this. Applying an Υ-selective supervisor to a single automaton can be described by the application of a live (η, Ξ)-supervisor to the synchronous product.

Lemma 5.2.30
Let A and B be η-partitioned automata over the alphabet Σ, let $\Xi \subseteq \Sigma \backslash \{\eta\}$, and let $\Upsilon \subseteq \Xi$ such that Υ is not relevant for B. Let Φ be an Υ-selective live (η, Ξ)-supervisor for A. Then there exists a live (η, Ξ)-supervisor Ψ for $A \parallel B$ such that

$$\Psi/(A \parallel B) = (\Phi/A) \parallel B.$$

Proof. Let A and B be η-partitioned automata over the alphabet Σ, let $\Xi \subseteq \Sigma \backslash \{\eta\}$, and let $\Upsilon \subseteq \Xi$ such that Υ is not relevant for B. Let Φ be an Υ-selective live (η, Ξ)-supervisor for A. Define

$$\Psi \colon Q_A \times Q_B \to 2^{\Sigma};\ (p, q) \mapsto \Phi(p).$$

It can be easily verified that properties (LS1) and (LS2) of definition 5.2.10 are satisfied for Ψ. In order to show property (LS3), let $\Psi(p, q) \cap \textsc{Elig}_{A\parallel B}(p, q) \cap \Xi = \emptyset$ for some $(p, q) \in Q_A \times Q_B$. We want to show that $\textsc{Elig}_{A\parallel B}(p, q) \cap \Xi = \emptyset$. We have

$$\emptyset = \Psi(p, q) \cap \textsc{Elig}_{A\parallel B}(p, q) \cap \Xi = \Phi(p) \cap \textsc{Elig}_A(p) \cap \textsc{Elig}_B(q) \cap \Xi. \tag{5.9}$$

Since $\Upsilon \subseteq \Xi$, we have

$$\emptyset = \Phi(p) \cap \textsc{Elig}_A(p) \cap \textsc{Elig}_B(q) \cap \Upsilon.$$

Since Υ is not relevant for B, we have $\Upsilon \subseteq \textsc{Elig}_B(q)$. Therefore,

$$\begin{aligned} \emptyset &= \Phi(p) \cap \textsc{Elig}_A(p) \cap \textsc{Elig}_B(q) \cap \Upsilon \\ &= \Phi(p) \cap \textsc{Elig}_A(p) \cap \Upsilon. \end{aligned}$$

Since Φ is Υ-selective this implies $\textsc{Elig}_A(p) \subseteq \Phi(p)$. Thus, we have

$$\begin{aligned} \textsc{Elig}_{A\|B}(p,q) \cap \Xi &= \textsc{Elig}_A(p) \cap \textsc{Elig}_B(q) \cap \Xi \\ &= \Phi(p) \cap \textsc{Elig}_A(p) \cap \textsc{Elig}_B(q) \cap \Xi \\ &= \emptyset \end{aligned}$$

by equation 5.9. Therefore, Ψ is a live (η, Ξ)-supervisor.

Now it suffices to show that $\Psi/(A \,\|\, B) = (\Phi/A) \,\|\, B$. This is the case since

$$\begin{aligned} & ((p_1, q_1), \sigma, (p_2, q_2)) \in \Delta_{\Psi/(A\|B)} \\ \Leftrightarrow\ & \sigma \in \Psi(p_1, q_1) \text{ and } ((p_1, q_1), \sigma, (p_2, q_2)) \in \Delta_{A\|B} \\ \Leftrightarrow\ & \sigma \in \Phi(p_1),\ (p_1, \sigma, p_2) \in \Delta_A, \text{ and } (q_1, \sigma, q_2) \in \Delta_B \\ \Leftrightarrow\ & (p_1, \sigma, p_2) \in \Delta_{\Phi/A} \text{ and } (q_1, \sigma, q_2) \in \Delta_B \\ \Leftrightarrow\ & ((p_1, q_1), \sigma, (p_2, q_2)) \in \Delta_{(\Phi/A)\|B}. \end{aligned}$$

□

Now we provide the desired result that, in order to check if the synchronous product of several automata is (η, Ξ)-nonblocking, it is sufficient to consider the synchronous product of these automata, in which one of the automata is substituted by its reduced automaton. Here automata are reduced by means of an Υ-selective (η, Ξ)-supervisor, for which $\Upsilon \subseteq \Xi$ is a set of events that is only relevant for the reduced automaton. Furthermore, the original system must be (η, Ξ)-nice.

Proposition 5.2.31

Let A and B be η-partitioned automata over the alphabet Σ, let $\Xi \subseteq \Sigma$, and let $\Upsilon \subseteq \Xi$ such that Υ is not relevant for B. Let $A \,\|\, B$ be (η, Ξ)-nice and let Φ be an Υ-selective live (η, Ξ)-supervisor for A. Then the following assertions hold.

(1) *$A \,\|\, B$ is (η, Ξ)-nonblocking if and only if $(\Phi/A) \,\|\, B$ is (η, Ξ)-nonblocking.*

(2) *$(\Phi/A) \,\|\, B$ is (η, Ξ)-nice.*

Proof. Let A and B be η-partitioned automata over the alphabet Σ, let $\Xi \subseteq \Sigma \backslash \{\eta\}$, and let $\Upsilon \subseteq \Xi$ such that Υ is not relevant for B. Let $A \,\|\, B$ be (η, Ξ)-nice, and let Φ be an Υ-selective live (η, Ξ)-supervisor for A. By lemma 5.2.30, there exists a live (η, Ξ)-supervisor Ψ for $A \,\|\, B$ such that $\Psi/(A \,\|\, B) = \Phi/A \,\|\, B$. By lemma 5.2.29, we obtain that $A \,\|\, B$ is (η, Ξ)-nonblocking if and only if $\Psi/(A \,\|\, B)$ is (η, Ξ)-nonblocking. This proves the first assertion. By lemma 5.2.28, we obtain that $\Psi/(A \,\|\, B)$ is (η, Ξ)-nice, namely (2) holds. □

(R2) Removing States

Reduction operation (R2) removes states from the coreachability part of an η-partitioned automaton in which exactly one output event is eligible that is hidden. If there is more than one such event eligible, the corresponding transition can be removed by reduction operation (R1).

In order to motivate reduction (R2), consider the example described in figure 5.3 again. Event Timer is relevant for automaton A_1 in figure 5.4, but irrelevant for automata A_2 and A_3. State `t1` is in the coreachability part of A_1, i.e. the subgraph at the right side, which is used to represent Ξ-controlled paths. Since event Timer is an output event, all Ξ-controlled paths originating from `t1` must eventually be continued by the transition labelled with event Timer. Since this event is not relevant for other automata, it cannot be disabled by other automata. Therefore, we can delete state `t1` and redirect incoming transitions to state `t2`. By doing the same for states `t5` and `t8`, we obtain automaton A'_1 in figure 5.5.

In order to define this reduction formally, we characterise states for which the reduction is applicable.

Definition 5.2.14 ((υ, Ξ)-reducible state)
Let A be an η-partitioned automaton over the alphabet Σ, let $\Xi \subseteq \Sigma$, and let $\upsilon \in \Xi$. A state $r \in Q_A$ is said to be (υ, Ξ)-reducible *in A if*

(RE1) $\text{ELIG}_A(r) \cap \Xi = \{\upsilon\}$,

(RE2) $(r, \upsilon, q) \in \Delta_A$ *for some* $q \in Q_A$ *implies* $q \neq r$, *and*

(RE3) $\eta \notin \text{ELIG}_A(r)$.

A state is (υ, Ξ)-reducible if the set of eligible output events consists exactly of one event (RE1) that must not be self-looped (RE2). Furthermore, such a state must be contained in the coreachability part (RE3).

State `t1` in automaton A_1 is (Timer, Ξ)-reducible. At this state, Timer is the only eligible output event (RE1), the transition labelled with Timer is not self-looped (RE2), and state `t1` is in the coreachability part (RE3).

Such reducible states can be eliminated as follows. Let r be the state to be removed, and let q be the target state of the transition originating from r and labelled with the output event that is only relevant for the automaton to be reduced. Then any transition ending in r is redirected to state q. Afterwards, state r is deleted from the automaton.

Definition 5.2.15
Let A be an η-partitioned automaton over the alphabet Σ, let $\Xi \subseteq \Sigma$, let $\upsilon \in \Xi$, and let $r \in Q_A$ be (υ, Ξ)-reducible. Define

$$\text{RED}_{r,\upsilon}(A) = (\Sigma, Q', \Delta', J_A, M_A \backslash \{r\}),$$

where

$$\begin{array}{rcl} Q' & = & Q_A \backslash \{r\} \\ \Delta' & = & (\Delta_A \cap (Q' \times \Sigma \times Q')) \cup \\ & & \{\, (p,\sigma,q) \in Q' \times \Sigma \times Q' \mid (p,\sigma,r),(r,\upsilon,q) \in \Delta_A \,\}. \end{array}$$

Reduction operation (R2) computes $\text{RED}_{r,\upsilon}(A)$ for a given η-partitioned automaton A, state r, and event υ. In the example in figures 5.4 and 5.5, we have

$$A_1' = \text{RED}_{\mathsf{t8},\mathsf{Timer}}(\text{RED}_{\mathsf{t5},\mathsf{Timer}}(\text{RED}_{\mathsf{t1},\mathsf{Timer}}(A_1))).$$

Given an η-partitioned automaton A, alphabet Ξ, and $r \in Q_A$ as above, let $A' = \text{RED}_{r,\upsilon}(A)$. Then we have

$$\text{ELIG}_A(q) = \text{ELIG}_{A'}(q) \tag{5.10}$$

for each $q \in Q_{A'}$.

In the following, we provide results enabling us to show that the introduced reduction behaves as desired: the property of being (η,Ξ)-nice is preserved, and the original system is (η,Ξ)-nonblocking if and only if the reduced system is (η,Ξ)-nonblocking.

In the next lemmas, we analyse the behaviour of the reduction operation when applied to an automaton that is part of a synchronous product. We begin with two auxiliary results stating that reduction (R2) preserves the existence of Ξ-controlled paths within a synchronous product. The first lemma shows that each Ξ-coreachable state in a system, in which one automaton is substituted by the result of reduction operation (R2), is also Ξ-coreachable in the original system.

Lemma 5.2.32
Let A and B be η-partitioned automata over the alphabet Σ, let $\Xi \subseteq \Sigma\backslash\{\eta\}$, let $\upsilon \in \Xi$ be not relevant for B, let $r \in Q_A$ be (υ,Ξ)-reducible, let $A' = \text{RED}_{r,\upsilon}(A)$, and let $o \in Q_{A'\|B}$.
If there exists a marked Ξ-controlled path in $A' \| B$ with origin o, then there exists a marked Ξ-controlled path in $A \| B$ with origin o.

Proof. Let A and B be η-partitioned automata over the alphabet Σ, let $\Xi \subseteq \Sigma\backslash\{\eta\}$, let $\upsilon \in \Xi$ be not relevant for B, let $r \in Q_A$ be (υ,Ξ)-reducible, and let $A' = \text{RED}_{r,\upsilon}(A)$.

We show by induction on $n \in \mathbb{N}_0$ that for all marked Ξ-controlled paths π in $A' \| B$ such that $|\pi| = n$ there exists a marked Ξ-controlled path in $A \| B$ with origin $s_o(\pi)$.

Consider $|\pi| = 0$, i.e. $\pi = o_0$ for some $o_0 \in Q_{A'\|B}$. Since $Q_{A'} \subseteq Q_A$, we have $o_0 \in Q_{A\|B}$, thus π is a path in $A \| B$. Furthermore, π is marked in $A \| B$ since

$M_{A'} \subseteq M_A$. Since π is Ξ-controlled in $A' \parallel B$, we have $\text{ELIG}_{A'\parallel B}(o_0) \cap \Xi = \emptyset$. By equation (5.10), we obtain that $\text{ELIG}_{A\parallel B}(o_0) \cap \Xi = \emptyset$. Therefore, $\pi = o_0$ is a marked and Ξ-controlled path in $A \parallel B$.

Now let

$$\pi = o_0 \xrightarrow{\sigma_1} o_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_{n+1}} o_{n+1}$$

be a marked Ξ-controlled path in $A' \parallel B$ of length $n+1$ for some $n \in \mathbb{N}_0$. Then

$$o_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_{n+1}} o_{n+1}$$

is a marked Ξ-controlled path in $A' \parallel B$ of length n. By the inductive assumption we obtain a marked Ξ-controlled path

$$o_1 \xrightarrow{\sigma'_2} o'_2 \xrightarrow{\sigma'_3} \cdots \xrightarrow{\sigma'_m} o'_m$$

in $A \parallel B$. Consider two cases.

Case 1: $(o_0, \sigma_1, o_1) \in \Delta_{A\parallel B}$. Then

$$o_0 \xrightarrow{\sigma_1} o_1 \xrightarrow{\sigma'_2} o'_2 \xrightarrow{\sigma'_3} \cdots \xrightarrow{\sigma'_m} o'_m$$

is a marked path in $A \parallel B$ with origin $s_o(\pi)$. Since π is Ξ-controlled in $A' \parallel B$, we have $\sigma_1 \in \Xi$ or $\Xi \cap \text{ELIG}_{A\parallel B}(o_0) = \Xi \cap \text{ELIG}_{A'\parallel B}(o_0) = \emptyset$. Thus, this path is Ξ-controlled.

Case 2: $(o_0, \sigma_1, o_1) \notin \Delta_{A\parallel B}$. Let $o_0 = (p_0, q_0)$ and $o_1 = (p_1, q_1)$. Recall that $(o_0, \sigma_1, o_1) \in \Delta_{A'\parallel B}$, i.e. $(p_0, \sigma_1, p_1) \in \Delta_{A'}$ and $(q_0, \sigma_1, q_1) \in \Delta_B$. By definition of $A' = \text{RED}_{r,\upsilon}(A)$, we have $(p_0, \sigma_1, r) \in \Delta_A$ and $(r, \upsilon, p_1) \in \Delta_A$. Since υ is not relevant for B, we have $(q_1, \upsilon, q_1) \in \Delta_B$. Therefore, $o_0 \xrightarrow{\sigma_1} (r, q_1) \xrightarrow{\upsilon} o_1$ is a path in $A \parallel B$. Then

$$\pi' = o_0 \xrightarrow{\sigma_1} (r, q_1) \xrightarrow{\upsilon} o_1 \xrightarrow{\sigma'_2} o'_2 \xrightarrow{\sigma'_3} \cdots \xrightarrow{\sigma'_m} o'_m$$

is a marked path in $A \parallel B$. By equation (5.10) and since π is Ξ-controlled in $A' \parallel B$, we have $\sigma_1 \in \Xi$ or $\Xi \cap \text{ELIG}_{A\parallel B}(o_0) = \Xi \cap \text{ELIG}_{A'\parallel B}(o_0) = \emptyset$. Thus, π' is a marked Ξ-controlled path in $A \parallel B$ with origin $s_o(\pi)$. $\square$

The next lemma shows that each Ξ-coreachable state in the closed-loop system of an (η, Ξ)-supervisor and the synchronous product of two automata is also Ξ-coreachable in the closed-loop system of the corresponding (η, Ξ)-supervisor and the synchronous product of these automata in which one of the automata is substituted by the result of reduction operation (R2).

Lemma 5.2.33
Let A and B be η-partitioned automata over the alphabet Σ, let $\Xi \subseteq \Sigma \backslash \{\eta\}$, let

$\upsilon \in \Xi$ be not relevant for B, let $r \in Q_A$ be (υ, Ξ)-reducible, let $A' = \textsc{Red}_{r,\upsilon}(A)$, and let $o \in Q_{A'\|B}$. Let Φ be a live (η, Ξ)-supervisor for $A \| B$. Let

$$\Psi\colon Q_{A'\|B} \to 2^{\Sigma}; \quad o \mapsto \Phi(o).$$

If there exists a marked Ξ-controlled path in $\Phi/(A \| B)$ with origin o, then there exists a marked Ξ-controlled path in $\Psi/(A' \| B)$ with origin o.

Proof. Let A and B be η-partitioned automata over the alphabet Σ, let $\Xi \subseteq \Sigma\backslash\{\eta\}$, let $\upsilon \in \Xi$ be not relevant for B, let $r \in Q_A$ be (υ, Ξ)-reducible, and let $A' = \textsc{Red}_{r,\upsilon}(A)$. Let Φ be a live (η, Ξ)-supervisor for $A \| B$, and let

$$\Psi\colon Q_{A'\|B} \to 2^{\Sigma}; \quad o \mapsto \Phi(o).$$

We show by induction on $n \in \mathbb{N}_0$ that for all marked Ξ-controlled paths π in $\Phi/(A \| B)$ such that $s_o(\pi) \in Q_{A'\|B}$ and $|\pi| = n$ there exists a marked Ξ-controlled path in $\Psi/(A' \| B)$ with origin $s_o(\pi)$.

Consider $|\pi| = 0$, i.e. $\pi = o$ for some $o \in Q_{A'\|B}$. We have $o \in M_{A\|B}$ and $\textsc{Elig}_{A\|B}(o) \cap \Xi = \emptyset$, since Φ is live. Since $M_{A'} = M_A \cap Q_{A'}$ and by equation (5.10), we obtain $o \in M_{A'\|B}$ and $\textsc{Elig}_{A'\|B}(o) \cap \Xi = \emptyset$. Thus, o is a marked Ξ-controlled path in $\Psi/(A' \| B)$.

Now let

$$\pi = o_0 \xrightarrow{\sigma_1} o_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_{n+1}} o_{n+1}$$

be a marked Ξ-controlled path in $\Phi/(A \| B)$ of length $n+1$ for some $n \in \mathbb{N}_0$ such that $o_0 \in Q_{A'\|B}$. Consider two cases.

Case 1: $o_1 \in Q_{A'\|B}$, i.e. $(o_0, \sigma_1, o_1) \in \Delta_{A'\|B}$. The path

$$o_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_{n+1}} o_{n+1}$$

is a marked Ξ-controlled path in $\Phi/(A \| B)$ of length n. By the inductive assumption, we obtain a marked Ξ-controlled path

$$o_1 \xrightarrow{\sigma'_2} o'_2 \xrightarrow{\sigma'_3} \cdots \xrightarrow{\sigma'_m} o'_m$$

in $\Psi/(A' \| B)$. By equation (5.10), we have $\textsc{Elig}_{A\|B}(o_0) \cap \Xi = \textsc{Elig}_{A'\|B}(o_0) \cap \Xi$. Since $o_0 \xrightarrow{\sigma_1} o_1$ is Ξ-preferring in $A' \| B$,

$$o_0 \xrightarrow{\sigma_1} o_1 \xrightarrow{\sigma'_2} o'_2 \xrightarrow{\sigma'_3} \cdots \xrightarrow{\sigma'_m} o'_m$$

is a marked Ξ-controlled path in $A' \| B$. Since $\sigma_1 \in \Phi(o_0) = \Psi(o_0)$, we obtain that it is a marked Ξ-controlled path in $\Psi/(A' \| B)$ with origin $s_o(\pi)$.

Case 2: $o_1 \notin Q_{A'\|B}$, i.e. $o_1 = (r, q_1)$ for some $q_1 \in Q_B$. Note that $\textsc{Elig}_A(r) \cap \Xi = \{\upsilon\}$, i.e. $\textsc{Elig}_{A\|B}(o_1) \cap \Xi = \{\upsilon\}$, since υ is not relevant for B. Then we have

$n > 1$ and $\sigma_2 = \upsilon$, since π is Ξ-complete. Let $o_0 = (p_0, q_0)$, and let $o_2 = (p_2, q_2)$. Then we have $(p_0, \sigma_1, r) \in \Delta_A$ and $(r, \upsilon, p_2) \in \Delta_A$. By definition of A', this implies $(p_0, \sigma_1, p_2) \in \Delta_{A'}$. Note that $q_1 = q_2$ and therefore $(q_0, \sigma_1, q_2) \in \Delta_B$, since υ is not relevant for B. Therefore, we have $(o_0, \sigma_1, o_2) \in \Delta_{A' \| B}$. The path

$$o_2 \xrightarrow{\sigma_3} \cdots \xrightarrow{\sigma_{n+1}} o_{n+1}$$

is marked and Ξ-controlled in $\Phi/(A \| B)$ and of length $n-1$. By the inductive assumption we obtain a marked Ξ-controlled path

$$o_2 \xrightarrow{\sigma'_3} o'_3 \xrightarrow{\sigma'_4} \cdots \xrightarrow{\sigma'_m} o'_m$$

in $\Psi/(A' \| B)$. Since $o_0 \xrightarrow{\sigma_1} o_1$ is Ξ-preferring in $A' \| B$ and $\textsc{Elig}_{A\|B}(o_0) \cap \Xi = \textsc{Elig}_{A'\|B}(o_0) \cap \Xi$,

$$o_0 \xrightarrow{\sigma_1} o_2 \xrightarrow{\sigma'_3} o'_3 \xrightarrow{\sigma'_4} \cdots \xrightarrow{\sigma'_m} o'_m$$

is a marked Ξ-controlled path in $A' \| B$. Since $\sigma_1 \in \Phi(o_0) = \Psi(o_0)$, we obtain that it is a marked Ξ-controlled path in $\Psi/(A' \| B)$ with origin $s_o(\pi)$. $\square$

By considering an (η, Ξ)-supervisor that does not disable any transition, we can conclude from the previous lemma that any Ξ-coreachable state of a system is also Ξ-coreachable in this system in which one automaton is simplified by applying reduction operation (R2).

Corollary 5.2.34
Let A and B be η-partitioned automata over the alphabet Σ, let $\Xi \subseteq \Sigma\backslash\{\eta\}$, let $\upsilon \in \Xi$ be not relevant for B, let $r \in Q_A$ be (υ, Ξ)-reducible, let $A' = \textsc{Red}_{r,\upsilon}(A)$, and let $o \in Q_{A'\|B}$.
If there exists a marked Ξ-controlled path in $A \| B$ with origin o, then there exists a marked Ξ-controlled path in $A' \| B$ with origin o.

Now we can show that a system is (η, Ξ)-nonblocking if and only if the system in which one automaton is simplified by means of reduction operation (R2) is (η, Ξ)-nonblocking.

Proposition 5.2.35
Let A and B be η-partitioned automata over the alphabet Σ, let $\Xi \subseteq \Sigma\backslash\{\eta\}$, let $\upsilon \in \Xi$ be not relevant for B, and let $r \in Q_A$ be (υ, Ξ)-reducible in A.

$A \| B$ is (η, Ξ)-nonblocking
if and only if
$\textsc{Red}_{r,\upsilon}(A) \| B$ is (η, Ξ)-nonblocking.

Proof. Let A and B be η-partitioned automata over the alphabet Σ, let $\Xi \subseteq \Sigma \backslash \{\eta\}$ let $\upsilon \in \Xi$ be not relevant for B, let $r \in Q_A$ be (υ, Ξ)-reducible in A, and let $A' = \textsc{Red}_{r,\upsilon}(A)$.

(If) Assume that $A' \| B$ is (η, Ξ)-nonblocking. Let $o \in Q_{A\|B}$ be η-reachable in $A \| B$. Consider two cases.

Case 1: $o \in Q_{A'\|B}$. Then o is also η-reachable in $A' \| B$. Since $A' \| B$ is (η, Ξ)-nonblocking, there exists a marked Ξ-controlled path in $A' \| B$ with origin o. By lemma 5.2.32, we obtain a marked Ξ-controlled path in $A \| B$ with origin o.

Case 2: $o \notin Q_{A'\|B}$. Then $o = (r, q)$ for some $q \in Q_B$. By definition 5.2.14, there exists $p \in Q_A$ such that $p \neq r$ and $(r, \upsilon, p) \in \Delta_A$. Since υ is not relevant for B, $(q, \upsilon, q) \in \Delta_B$. Thus, we have $((r, q), \upsilon, (p, q)) \in \Delta_{A\|B}$. Since o is η-reachable in $A \| B$, there exists $o_\eta = (p_\eta, q_\eta) \in Q_{A\|B}$ such that $(o_\eta, \eta, o) \in \Delta_{A\|B}$, i.e. $(p_\eta, \eta, r) \in \Delta_A$ and $(q_\eta, \eta, q) \in \Delta_B$. By definition of A', o_η is reachable in $A' \| B$. Furthermore, we have $(p_\eta, \eta, r) \in \Delta_A$ and $(r, \upsilon, p) \in \Delta_A$. By definition of A', this implies $(p_\eta, \eta, p) \in \Delta_{A'}$. Thus, we have $((p_\eta, q_\eta), \eta, (p, q)) = (o_\eta, \eta, (p, q)) \in \Delta_{A'\|B}$, i.e. (p, q) is η-reachable in $A' \| B$. Since $A' \| B$ is η-nonblocking, there exists a marked Ξ-controlled path in $A' \| B$ with origin (p, q). By lemma 5.2.32, there exists a marked Ξ-controlled path

$$o_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} o_n$$

in $A \| B$ such that $o_0 = (p, q)$. Then

$$(r, q) \xrightarrow{\upsilon} o_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} o_n$$

is a marked Ξ-controlled path in $A \| B$ with origin o, thus $A \| B$ is (η, Ξ)-nonblocking.

(Only If) Assume that $A \| B$ is (η, Ξ)-nonblocking. Let o be η-reachable in $A' \| B$. Since $Q_{A'} \subseteq Q_A$ and by property (RE3) of definition 5.2.14, we have that o is η-reachable in $A \| B$. Since $A \| B$ is (η, Ξ)-nonblocking, there exists a marked Ξ-controlled path in $A \| B$ with origin o. By corollary 5.2.34, there exists a marked Ξ-controlled path in $A' \| B$ with origin o, thus $A' \| B$ is (η, Ξ)-nonblocking. $\square$

Next, we show that if a system is (η, Ξ)-nice, then the system in which one automaton is simplified by applying reduction operation (R2) is also (η, Ξ)-nice, i.e. the property of being (η, Ξ)-nice is an invariant for this reduction.

Proposition 5.2.36
Let A and B be η-partitioned automata over the alphabet Σ, let $\Xi \subseteq \Sigma \backslash \{\eta\}$, let $\upsilon \in \Xi$ be not relevant for B, and let $r \in Q_A$ be (υ, Ξ)-reducible in A.
If $A \| B$ is (η, Ξ)-nice, then $\textsc{Red}_{r,\upsilon}(A) \| B$ is (η, Ξ)-nice.

Proof. Let A and B be η-partitioned automata over the alphabet Σ, let $\Xi \subseteq \Sigma \backslash \{\eta\}$, let $\upsilon \in \Xi$ be not relevant for B, let $r \in Q_A$ be (υ, Ξ)-reducible in A, let $A' = \textsc{Red}_{r,v}(A)$, and let $A \parallel B$ be (η,Ξ)-nice. Assume that $A' \parallel B$ is (η,Ξ)-nonblocking, and let Ψ be a live (η,Ξ)-supervisor for $A' \parallel B$. Consider the live (η,Ξ)-supervisor Φ for $A \parallel B$ defined as follows:

$$\begin{aligned} \Phi\colon Q_{A\parallel B} &\rightarrow 2^{\Sigma} \\ (p,q) &\mapsto \begin{cases} \Psi(p,q), & \text{if } p \neq r \\ \Sigma, & \text{otherwise.} \end{cases} \end{aligned}$$

By proposition 5.2.35, we know that $A \parallel B$ is (η,Ξ)-nonblocking. Since $A \parallel B$ is (η,Ξ)-nice, $\Phi/(A \parallel B)$ is (η,Ξ)-nonblocking. Let $o \in Q_{A'\parallel B}$ be η-reachable in $A' \parallel B$. By definition of A', o is also η-reachable in $A \parallel B$. Since Φ is an (η,Ξ)-supervisor, this implies that o is η-reachable in $\Phi/(A \parallel B)$. Since $\Phi/(A \parallel B)$ is (η,Ξ)-nonblocking, there exists a marked Ξ-controlled path in $\Phi/(A \parallel B)$ with origin o. By lemma 5.2.33, there exists a marked and Ξ-controlled path in $\Psi/(A' \parallel B)$ with origin o. This proves that that $\Psi/(A' \parallel B)$ is (η,Ξ)-nonblocking, i.e. $A' \parallel B$ is (η,Ξ)-nice. □

(R3) Merging Equivalent States

The next reduction operation, called (R3), simplifies again the coreachability part of an η-partitioned automaton. We identify sets of states for which a marked state can be reached by the same set of Ξ-controlled strings, taking into account that some events are irrelevant for all other automata of a system. These equivalent states are merged, leading to an automaton with fewer states.

Consider automaton A_2 in figure 5.7 from the example in figure 5.3. Events *CLOSE_DRD*, *CLOSE_PSD*, and *KEY_OUT* are not relevant for automata A_1 and A_3, i.e. they are hidden for A_1 and A_3. States `t1`, `t2`, and `t3` on the coreachability part of automaton A_2 are equivalent, when hidden events cannot be observed. All transitions labelled with an observable event starting at on of these states also end in one of these states. Event *OPEN_DRD* is not eligible at state `t3`, but state `t2`, in which *OPEN_DRD* is eligible, can be reached by means of a hidden event. The hidden event cannot be observed, and an observer cannot tell the difference whether *OPEN_DRD* is eligible or not. State `t0` can be reached by a hidden event from state `t1`. But state `t0` can also be reached by a sequence of hidden events from states `t2` and `t3`. Again, an observer cannot tell the difference whether a hidden event or a sequence of hidden events has occurred. Merging the equivalent states `t1`, `t2`, and `t3` leads to the automaton A_2' in figure 5.8.

This reduction operation can also be described using an equivalence relation on the state set of the automaton to be reduced. In the example above, we have $A_2' = A_2/{\sim_2}$, and

$$\begin{aligned} \sim_2 \;=\; & \{(q,q) \mid q \in Q_{A_2}\} \cup \\ & \{(\mathtt{t1},\mathtt{t2}),(\mathtt{t1},\mathtt{t3}),(\mathtt{t2},\mathtt{t3}),(\mathtt{t2},\mathtt{t1}),(\mathtt{t3},\mathtt{t1}),(\mathtt{t3},\mathtt{t2})\}. \end{aligned}$$

Not all equivalence relations can be used for a reduction. The information if an output event is eligible at a certain state must be preserved. By considering Ξ-restricted paths, we ensure that each output event that is eligible on a Ξ-restricted path is also eligible at the origin of this path.

Definition 5.2.16 (Ξ-restricted path)
Let A be an automaton, let $\Xi \subseteq \Sigma_A$, and let $\pi = p_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} p_n$ be a path in A. π is said to be Ξ-restricted *in A if for each $k \in \{1, \ldots, n\}$*

$$\textsc{Elig}_A(p_k) \cap \Xi \subseteq \textsc{Elig}_A(p_0).$$

Consider automaton A_1' in figure 5.5 from the example in figure 5.3. Even though event *TIMEOUT* is hidden with respect to A_1', states `t2` and `t3` cannot be treated as equivalent. In order to understand this problem, consider the synchronous product of this automaton and automaton `oops` in figure 5.12. Assume that event *oops* is an input event, i.e. *oops* $\notin \Xi$. It is possible to reach the marked state by means of Ξ-controlled paths starting from state $(\mathtt{t2}, \mathtt{q1})$, but not when starting from state $(\mathtt{t3}, \mathtt{q1})$, since event comfort_start is preferred leading to state $(\mathtt{t4}, \mathtt{q2})$, from which a marked state cannot be reached. The path $\mathtt{t2} \xrightarrow{\textit{TIMEOUT}} \mathtt{t3}$ is not Ξ-restricted.

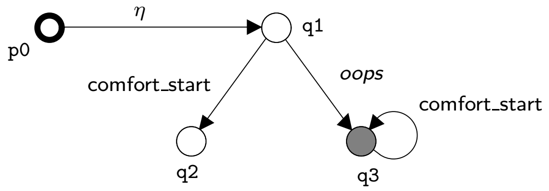

Figure 5.12: Automaton `oops`.

Using the concept of Ξ-restricted paths, we now define a state reduction equivalence. This equivalence relation is used to define reduction operation (R3). Let Υ be a set of events that are only relevant for the automaton to be reduced and that neither contain η nor output events. States are equivalent if they can always be continued in the same way, as far as the occurrence of events not contained in Υ, transitions to other equivalence classes, and marking are concerned. Ξ-restricted Υ-paths can be used in order to reach states from which such a continuation is possible.

Definition 5.2.17 ((Υ, Ξ)-state reduction equivalence)
Let A be an η-partitioned automaton over the alphabet Σ, let $\Xi \subseteq \Sigma \backslash \{\eta\}$, and let $\Upsilon \subseteq \Sigma \backslash (\{\eta\} \cup \Xi)$. An equivalence relation $\sim \subseteq Q_A \times Q_A$ is said to be an (Υ, Ξ)-state reduction equivalence *for A if for all $p, q \in Q_A$ such that $p \sim q$, the following conditions are satisfied:*

(SR1) $\eta \in \textsc{Elig}_A(p)$ *implies* $p = q$.

(SR2) *If* $(p, \xi, E) \in \Delta_A$ *for some* $\xi \in \Xi$ *and* $E \in Q_A/\sim$*, then* $(q, \xi, E) \in \Delta_A$.

(SR3) *If* $(p, \upsilon, E) \in \Delta_A$ *for some* $\upsilon \in \Upsilon$ *and* $E \in Q_A/\sim$*, then there exists a* Ξ*-restricted* Υ*-path in* A *such that* $s_o(\pi) = q$ *and* $s_e(\pi) \in E$.

(SR4) *If* $(p, \sigma, E) \in \Delta_A$ *for some* $\sigma \in \Sigma\backslash(\Upsilon \cup \Xi)$ *and* $E \in Q_A/\sim$*, then there exists a* Ξ*-restricted* Υ*-path in* A *such that* $s_o(\pi) = q$ *and* $(s_e(\pi), \sigma, E) \in \Delta_A$.

(SR5) *If* $p \in M_A$*, then there exists a marked and* Ξ*-restricted* Υ*-path* π *in* A *such that* $s_o(\pi) = q$.

Condition (SR1) ensures that only the coreachability part of the η-partitioned automaton is reduced, i.e. η must not be eligible in different states that are equivalent. Equivalent states must have the same eligible output events, and the corresponding transitions must take us to the same equivalence classes (SR2). Equivalence classes reachable by a transition labelled with a hidden event must be reachable by means of Ξ-restricted Υ-paths from equivalent states (SR3). Equivalence classes reachable by any other event must be reachable from any equivalent state by first following a Ξ-restricted Υ-path and then following a transition labelled with this event (SR4). If one of the equivalent states is marked, then a marked state must be reachable by means of Ξ-restricted Υ-paths from any of the equivalent states (SR5). The equivalence relation $\sim_2$ defined for the example on page 96 is a (Υ, Ξ)-state reduction equivalence.

Reduction operation (R3) computes the quotient of an η-partitioned automaton with respect to an (Υ, Ξ)-state reduction equivalence. Recall that Υ is a set of events that are only relevant for the automaton to be reduced, and $\Upsilon \cap \Xi = \emptyset$. In the following, we show that reduction operation (R3) can be used when checking a system if it is Ξ-nonblocking.

In the remainder of this section, we show that the property to be (η, Ξ)-nice is an invariant for reduction (R3), and we show that a system is (η, Ξ)-nonblocking if and only if the reduced system is (η, Ξ)-nonblocking. We start by providing some basic results.

Because of (SR2), for a given η-partitioned automaton A and (Υ, Ξ)-state reduction equivalence $\sim$ for A, we have

$$\textsc{Elig}_A(q) \cap \Xi = \textsc{Elig}_{A/\sim}([q]) \cap \Xi. \tag{5.11}$$

for each state q in A.

Next, we show that each path in the original automaton is also a path in the reduced automaton when considering the equivalence classes.

Lemma 5.2.37
Let A *be an* η*-partitioned automaton, let* $\sim$ *be an* (Υ, Ξ)*-state reduction equivalence for* A*, and let*

$$\pi = p_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} p_n$$

be a path in A. Then

$$\tilde{\pi} = [p_0] \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} [p_n]$$

is a path in $A/\sim$. Furthermore,

(1) *π is Ξ-preferring in A if and only if $\tilde{\pi}$ is Ξ-preferring in $\tilde{A}$ and*

(2) *π is Ξ-complete in A if and only if $\tilde{\pi}$ is Ξ-complete in $A/\sim$.*

Proof. Let A be an η-partitioned automaton, let $\sim$ be an (Υ, Ξ)-state reduction equivalence for A, and let $\pi = p_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} p_n$ be a path in A. Write $\tilde{A} = A/\sim$. By lemma 3.1.5, $[p_0] \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} [p_n]$ is a path in $\tilde{A}$. For all $p \in Q_A$, we know that $\textsc{Elig}_A(p) \cap \Xi = \textsc{Elig}_{\tilde{A}}([p]) \cap \Xi$ by equation (5.11). Therefore, π is Ξ-preferring (Ξ-complete) in A if and only if $\tilde{\pi}$ is Ξ-preferring (Ξ-complete) in $\tilde{A}$. □

A path in the reduced automaton is not necessarily a valid path in the original automaton. However, there exists a corresponding path in the original automaton that is Ξ-preferring (Ξ-complete) if the original path is Ξ-preferring (Ξ-complete).

Lemma 5.2.38
Let A be an η-partitioned automaton over the alphabet Σ, and let $\sim$ be an (Υ, Ξ)-state reduction equivalence for A. Then, for each path

$$\tilde{\pi} = E_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} E_n$$

in $A/\sim$ and for each $q_0 \in E_0$, there exists a path

$$\pi = q_0 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_m} q_m$$

in A such that $q_m \in E_n$,

(1) *$\tilde{\pi}$ is Ξ-preferring implies that π is Ξ-preferring,*

(2) *$\tilde{\pi}$ is Ξ-complete implies that π is Ξ-complete, and*

(3) *$n > 0$ and $\sigma_n \notin \Upsilon$ implies $m > 0$ and $\tau_m = \sigma_n$.*

Proof. Let A be an η-partitioned automaton over the alphabet Σ, and let $\sim$ be an (Υ, Ξ)-state reduction equivalence for A. Write $\tilde{A} = A/\sim$.

We first show by induction on the length n of a path $\tilde{\pi}$ in $\tilde{A}$ with origin E_0 and end E_n that, given state $q_0 \in E_0$, there exists a path π in A with origin q_0 and end contained in E_n such that $\tilde{\pi}$ is Ξ-preferring in $\tilde{A}$ implies that π is Ξ-preferring in A, i.e. property (1) is satisfied.

Base case ($n = 0$). Assume that $E_0 \in Q_{\tilde{A}}$ is a path in $\tilde{A}$ and let $q_0 \in E_0$. We have that q_0 is a path in A with origin and end $q_0 \in E_0$. Furthermore, q_0 is Ξ-preferring in A.

Inductive step ($n \to n+1$). Let $\tilde{\pi} = E_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_{n+1}} E_{n+1}$ be a path in $\tilde{A}$, and let $q_0 \in E_0$. Consider $\tilde{\pi}_1 = E_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} E_n$. By the inductive assumption, we obtain a path π_1 in A with origin q_0 and end $q \in E_n$ that is Ξ-preferring in A if $\tilde{\pi}$ is Ξ-preferring in $\tilde{A}$. Since $(E_n, \sigma_{n+1}, E_{n+1}) \in \Delta_{\tilde{A}}$, there exists $p \in E_n$ such that $(p, \sigma_{n+1}, E_{n+1}) \in \Delta_A$. We have $p \sim q$. Consider three cases:

Case 1: $\sigma_{n+1} \in \Xi$. We have $(q, \sigma_{n+1}, E_{n+1}) \in \Delta_A$ by (SR2). Let $q_e \in E_{n+1}$ such that $(q, \sigma_{n+1}, q_e) \in \Delta_A$, i.e. $\pi_2 := q \xrightarrow{\sigma_{n+1}} q_e$ is a path in A. Then $\pi := \pi_1\pi_2$ is a path in A with origin q_0 and end $q_e \in E_{n+1}$. Assume that $\tilde{\pi}$ is Ξ-preferring in $\tilde{A}$. Then we have that π_1 is also Ξ-preferring in A. Since $\sigma_{n+1} \in \Xi$, we obtain that π is Ξ-preferring in A.

Case 2: $\sigma_{n+1} \in \Upsilon$. We obtain a Ξ-restricted Υ-path π_2 in A with origin q and end $q_e \in E_{n+1}$ by (SR3). Then $\pi := \pi_1\pi_2$ is a path in A with origin q_0 and end $q_e \in E_{n+1}$. Now assume that $\tilde{\pi}$ is Ξ-preferring in $\tilde{A}$. Then we have that π_1 is Ξ-preferring in A. It suffices to show that π_2 is Ξ-preferring in A. Since $\sigma_{n+1} \notin \Xi$, we have $\textsc{Elig}_{\tilde{A}}(E_n) \cap \Xi = \emptyset$, because π is Ξ-preferring. Since $q \in E_n$, we obtain $\textsc{Elig}_A(q) \cap \Xi = \textsc{Elig}_{\tilde{A}}(E_n) \cap \Xi = \emptyset$ by equation (5.11). Since π_2 is Ξ-restricted, we have $\textsc{Elig}_A(q') \cap \Xi \subseteq \textsc{Elig}_A(q) \cap \Xi = \emptyset$ for all states q' that occur in π_2. Therefore, π is Ξ-preferring in A.

Case 3: $\sigma_{n+1} \in \Sigma\backslash(\Upsilon \cup \Xi)$. By (SR4), we obtain a Ξ-restricted Υ-path π_2 with origin q such that $(s_e(\pi_2), \sigma_{n+1}, E_{n+1}) \in \Delta_A$. Let $q_e \in E_{n+1}$ such that $(s_e(\pi_2), \sigma_{n+1}, q_e) \in \Delta_A$. Then $\pi := \pi_1\pi_2(s_e(\pi_2), \sigma_{n+1}, q_e)$ is a path in A with origin q_0 and end $q_e \in E_{n+1}$. The proof that $\tilde{\pi}$ is Ξ-preferring in $\tilde{A}$ implies that π is Ξ-preferring in A is equivalent to the one given in case 2.

The induction shows property (1). Property (2) follows from equation (5.11); property (3) follows immediately from the construction above (case 1 and 3); $\square$

The following lemma extends the above result to consider marked paths. If there exists a Ξ-controlled and marked path in the reduced automaton, then there exist Ξ-controlled and marked paths starting from any state of the origin equivalence class.

Lemma 5.2.39
Let A be an η-partitioned automaton, and let $\sim$ be an (Υ, Ξ)-state reduction equivalence for A. For all Ξ-controlled and marked paths $\tilde{\pi}$ in $A/\sim$ and all $p_o \in Q_A$ such that $p_o \in s_o(\tilde{\pi})$ there exists a Ξ-controlled and marked path π in A with origin p_o.

Proof. Let A be an η-partitioned automaton, and let $\sim$ be an (Υ, Ξ)-state reduction equivalence for A. Write $\tilde{A} = A/\sim$. Let $\tilde{\pi}$ be a Ξ-controlled and marked path in $\tilde{A}$, and let $p_o \in Q_A$ such that $p_o \in s_o(\tilde{\pi})$. Write $E_e = s_e(\tilde{\pi})$. By lemma 5.2.38, we obtain a Ξ-controlled path π_1 in A with origin p_o such that $p := s_e(\pi_1) \in E_e$. There exits $q_m \in M_A$ such that $q_m \in E_e$, since $\tilde{\pi}$ is marked. Then we have $q_m \sim p$, and by (SR5), we obtain a Ξ-restricted and marked Υ-path π_2 in A such that $s_o(\pi_2) = p$. Let $\pi = \pi_1\pi_2$. This is a marked path in A such that $s_o(\pi) = p_o$. It suffices to show that π_2 is Ξ-controlled. Since $\tilde{\pi}$ is Ξ-controlled we have $\textsc{Elig}_{\tilde{A}}(E_e) \cap \Xi = \emptyset$. Since $p \in E_e$ we have $\textsc{Elig}_A(p) \cap \Xi = \textsc{Elig}_{\tilde{A}}(E_e) \cap \Xi = \emptyset$ by equation (5.11). Since π_2 is Ξ-restricted, we have $\textsc{Elig}_A(p') \cap \Xi \subseteq \textsc{Elig}_A(p) \cap \Xi = \emptyset$ for all states p' that occur in π_2. Thus, π_2 is Ξ-controlled in A. □

Now we are able to prove that an η-partitioned automaton is (η, Ξ)-nonblocking if and only if the reduced automaton is (η, Ξ)-nonblocking.

Lemma 5.2.40
Let A be an η-partitioned automaton, and let $\sim$ be an (Υ, Ξ)-state reduction equivalence for A.
A is (η, Ξ)-nonblocking if and only if $A/{\sim}$ is (η, Ξ)-nonblocking.

Proof. Let A be an η-partitioned automaton, let $\sim$ be an (Υ, Ξ)-state reduction equivalence for A, and let $\tilde{A} = A/{\sim}$.

(Only If) Assume that A is (η, Ξ)-nonblocking. We need to show that $\tilde{A}$ is (η, Ξ)-nonblocking. In order to do this, let $E \in Q_{\tilde{A}}$ be η-reachable in $\tilde{A}$. Note that $\eta \notin \Upsilon$. There exists a state $p \in E$ that is η-reachable in A by lemma 5.2.38. Since A is (η, Ξ)-nonblocking, there exists a Ξ-controlled and marked path in A that starts in p. We obtain a Ξ-controlled and marked path in $\tilde{A}$ which starts in E by lemma 5.2.37.

(If) Assume that $\tilde{A}$ is (η, Ξ)-nonblocking. We want to show that A is (η, Ξ)-nonblocking. Let $q \in Q_A$ be η-reachable in A. By lemma 3.1.5 we obtain that $[q]$ is η-reachable in $\tilde{A}$. Since $\tilde{A}$ is (η, Ξ)-nonblocking, we obtain a Ξ-controlled and marked path in $\tilde{A}$ originating in $[q]$. By lemma 5.2.39, we obtain a Ξ-controlled and marked path in A originating in q. □

Next, we show that the reduced automaton is (η, Ξ)-nice if the original automaton is (η, Ξ)-nice.

Lemma 5.2.41
Let A be an η-partitioned automaton, let $\sim$ be an (Υ, Ξ)-state reduction equivalence for A, and let A be (η, Ξ)-nice. Then $A/{\sim}$ is (η, Ξ)-nice.

Proof. Let A be an η-partitioned automaton over the alphabet Σ, let $\sim$ be an (Υ, Ξ)-state reduction equivalence for A, let $\tilde{A} = A/{\sim}$, and let A be (η, Ξ)-nice. Assume that $\tilde{A}$ is (η, Ξ)-nonblocking, and let Ψ be a live (η, Ξ)-supervisor for $\tilde{A}$. By lemma 5.2.40, we have that A is (η, Ξ)-nonblocking. Define

$$\Phi \colon Q_A \to 2^{\Sigma}; \quad q \mapsto \Psi([q]).$$

We claim that Φ is a live (η, Ξ)-supervisor for A. Property (LS1) follows immediately from the definition of Φ, since Ψ is a live (η, Ξ)-supervisor. Let $q \in Q_A$ such that $\eta \in \textsc{Elig}_A(q)$ and hence $\eta \in \textsc{Elig}_{\tilde{A}}([q])$. Since Ψ is a live (η, Ξ)-supervisor we have $\Phi(q) = \Psi([q]) = \Sigma$ which proves property (LS2). Now let

$q \in Q_A$ such that $\Phi(q) \cap \textsc{Elig}_A(q) \cap \Xi = \emptyset$. By equation (5.11), we obtain $\Psi([q]) \cap \textsc{Elig}_{\tilde{A}}([q]) \cap \Xi = \Phi(q) \cap \textsc{Elig}_A(q) \cap \Xi = \emptyset$. Since Ψ is a live (η, Ξ)-supervisor for $\tilde{A}$, we then have $\textsc{Elig}_{\tilde{A}}([q]) \cap \Xi = \emptyset$, so that $\textsc{Elig}_A(q) \cap \Xi = \emptyset$ which proves (LS3). Therefore, Φ is a live (η, Ξ)-supervisor for A.

Since A is nice, Φ/A is (η, Ξ)-nonblocking. Now we show that $\Psi/\tilde{A}$ is (η, Ξ)-nonblocking. In order to do this, let $E \in Q_{\Psi/\tilde{A}} = Q/{\sim}$ be η-reachable in $\Psi/\tilde{A}$. Then E is η-reachable in $\tilde{A}$ by lemma 5.2.21. Note that $\eta \notin \Upsilon$. There exists $q \in E$ such that q is η-reachable in A by lemma 5.2.38. Since Φ/A is (η, Ξ)-nonblocking, there exists a Ξ-controlled and marked path $p_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} p_n$ in Φ/A. By lemma 5.2.37, we have that $\tilde{\pi} = [p_0] \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} [p_n]$ is a Ξ-controlled and marked path in $\tilde{A}$. It suffices to show that, for all $i \in \{0, \ldots, n-1\}$, we have $\sigma_{i+1} \in \Psi([p_i])$, so that $\tilde{\pi}$ is a Ξ-controlled and marked path in $\Psi/\tilde{A}$. But this is the case because $\sigma_{i+1} \in \Phi(p_i) = \Psi([p_i])$. $\square$

So far, we have only considered a single automaton. In the following, we prove that the shown properties can be applied to a system consisting of several automata that are reduced separately. We show that applying a reduction to a single automaton can be represented as a reduction on the product automaton.

Lemma 5.2.42
Let A and B be η-partitioned automata over the same alphabet Σ. Let $\Xi \subseteq \Sigma \backslash \{\eta\}$, let $\Upsilon \subseteq \Sigma$ such that $\Upsilon \cap \Xi = \emptyset$ and Υ is not relevant for B, and let $\sim_A$ be an (Υ, Ξ)-state reduction equivalence for A. Then there exists an (Υ, Ξ)-state reduction equivalence $\sim_{A\|B}$ for $A \,\|\, B$ such that

$$(A/{\sim_A}) \,\|\, B \;\cong\; (A \,\|\, B)/{\sim_{A\|B}}\,.$$

Proof. Let A and B be η-partitioned automata over the same alphabet Σ, let $\Xi \subseteq \Sigma \backslash \{\eta\}$, let $\Upsilon \subseteq \Sigma$ such that $\Upsilon \cap \Xi = \emptyset$ and Υ is not relevant for B, and let $\sim_A$ be an (Υ, Ξ)-state reduction equivalence for A. Write $\tilde{A} = A/{\sim_A}$.

Define

$$\sim_{A\|B} = \{\, ((p_1, q), (p_2, q)) \in Q_{A\|B} \times Q_{A\|B} \mid p_1 \sim_A p_2 \,\}.$$

We claim that $\sim_{A\|B}$ is an (Υ, Ξ)-state reduction equivalence for $A \,\|\, B$. Let $o, o' \in Q_{A\|B}$ such that $o \sim_{A\|B} o'$. There exist $p, p' \in Q_A$, $q \in Q_B$ such that $o = (p, q)$, $o' = (p', q)$, and $p \sim_A p'$.

(SR1) Let $\eta \in \textsc{Elig}_{A\|B}(o)$. Hence, we have $\eta \in \textsc{Elig}_A(p)$ and, since $\sim_A$ is a state reduction equivalence, $p = p'$, so that $o = o'$.

(SR2) Let $\xi \in \Xi$ and $E \in Q_{A\|B}/{\sim_{A\|B}}$ such that $(o, \xi, E) \in \Delta_{A\|B}$. There exist $(p_\xi, q_\xi) \in E$ such that $(p, \xi, p_\xi) \in \Delta_A$ and $(q, \xi, q_\xi) \in \Delta_B$. Since $\sim_A$ is an (Υ, Ξ)-state reduction equivalence for A, $(p', \xi, [p_\xi]) \in \Delta_A$, i.e. there exists $p'_\xi \in [p_\xi]$ such that $(p', \xi, p'_\xi) \in \Delta_A$. Hence, we have $(o', \xi, (p'_\xi, q_\xi)) = ((p', q), \xi, (p'_\xi, q_\xi)) \in \Delta_{A\|B}$, so that $(o', \xi, E) \in \Delta_{A\|B}$, since $(p'_\xi, q_\xi) \sim_{A\|B} (p_\xi, q_\xi)$.

(SR3) Let $\upsilon \in \Upsilon$ and $E \in Q_{A\|B}/{\sim_{A\|B}}$ such that $(o, \upsilon, E) \in \Delta_{A\|B}$. There exists $o_\upsilon \in E$ such that $(o, \upsilon, o_\upsilon) \in \Delta_{A\|B}$. Since $\upsilon \in \Upsilon$ is not relevant for B, we have $o_\upsilon = (p_\upsilon, q)$ for some $p_\upsilon \in Q_A$. Then we have $(p, \upsilon, p_\upsilon) \in \Delta_A$. Since $\sim_A$ is an (Υ, Ξ)-state reduction equivalence for A and $p \sim_A p'$, we obtain a Ξ-restricted Υ-path $p' \xrightarrow{\upsilon_1} p_1 \xrightarrow{\upsilon_2} \cdots \xrightarrow{\upsilon_n} p_n$ in A such that $p_n \sim_A p_\upsilon$. Since Υ is not relevant for B, we have that $(p', q) \xrightarrow{\upsilon_1} (p_1, q) \xrightarrow{\upsilon_2} \cdots \xrightarrow{\upsilon_n} (p_n, q)$ is a Ξ-restricted Υ-path in $A \,\|\, B$. Furthermore, since $p_n \sim_A p_\upsilon$, we have $o_n = (p_n, q) \sim_{A\|B} (p_\upsilon, q) = o_\upsilon$ and therefore $o_n \in E$.

(SR4) Let $\sigma \in \Sigma \backslash (\Upsilon \cup \Xi)$ and $E \in Q_{A\|B}/{\sim_{A\|B}}$ such that $(o, \sigma, E) \in \Delta_{A\|B}$. Then there exists $o_\sigma \in E$ such that $(o, \sigma, o_\sigma) \in \Delta_{A\|B}$. Let $p_\sigma \in Q_A$ and $q_\sigma \in Q_B$ such that $o_\sigma = (p_\sigma, q_\sigma)$. Then $(p, \sigma, p_\sigma) \in \Delta_A$ and $(q, \sigma, q_\sigma) \in \Delta_B$. Since $\sim_A$ is an (Υ, Ξ)-state reduction equivalence for A and $p \sim_A p'$, we obtain a Ξ-restricted Υ-path $p' \xrightarrow{\upsilon_1} p_1 \xrightarrow{\upsilon_2} \cdots \xrightarrow{\upsilon_n} p_n$ such that $(p_n, \sigma, [p_\sigma]) \in \Delta_A$. Since Υ is not relevant for B, we have that $(p', q) \xrightarrow{\upsilon_1} (p_1, q) \xrightarrow{\upsilon_2} \cdots \xrightarrow{\upsilon_n} (p_n, q)$ is a Ξ-restricted Υ-path in $A \,\|\, B$. Furthermore, since $(p_n, \sigma, [p_\sigma]) \in \Delta_A$ and $(q, \sigma, q_\sigma) \in \Delta_B$, $((p_n, q), \sigma, E) \in \Delta_{A\|B}$.

(SR5) Let $o \in M_{A\|B}$. Then $p \in M_A$ and $q \in M_B$. Since $\sim_A$ is an (Υ, Ξ)-state reduction equivalence for A and $p \sim_A p'$, we obtain a Ξ-restricted and marked Υ-path $p' \xrightarrow{\upsilon_1} p_1 \xrightarrow{\upsilon_2} \cdots \xrightarrow{\upsilon_n} p_n$ in A. Since Υ is not relevant for B, we have that $(p', q) \xrightarrow{\upsilon_1} (p_1, q) \xrightarrow{\upsilon_2} \cdots \xrightarrow{\upsilon_n} (p_n, q)$ is a Ξ-restricted and marked Υ-path in $A \,\|\, B$.

Write $\widetilde{A \,\|\, B} = (A \,\|\, B)/{\sim_{A\|B}}$. It suffices to show that $\tilde{A} \,\|\, B \cong \widetilde{A \,\|\, B}$. Let

$$\varphi\colon Q_{\tilde{A}\|B} \to Q_{\widetilde{A\|B}}; \quad ([p], q) \mapsto [(p, q)].$$

We show that φ is a bijective function which satisfies properties (EQ1), (EQ2), and (EQ3) in definition 3.1.5 on page 17.

φ is well-defined: Let $p, p' \in Q_A$ such that $p \sim_A p'$ and let $q \in Q_B$. By definition of $\sim_{A\|B}$ it follows immediately that $(p, q) \sim_{A\|B} (p', q)$ and therefore $\varphi([p], q) = [(p, q)] = [(p', q)] = \varphi([p'], q)$.

φ is injective: Let $p_1, p_2 \in Q_A$ and $q_1, q_2 \in Q_B$ such that $\varphi([p_1], q_1) = \varphi([p_2], q_2)$, i.e. $[(p_1, q_1)] = [(p_2, q_2)]$. Then we have $(p_1, q_1) \sim_{A\|B} (p_2, q_2)$, i.e. $p_1 \sim_A p_2$ and $q_1 = q_2$. Therefore, $([p_1], q_1) = ([p_2], q_2)$.

φ is surjective: Let $E \in Q_{A\|B}/{\sim_{A\|B}}$. Then there exists $o \in Q_{A\|B}$ such that $[o] = E$. Let $p \in Q_A$ and $q \in Q_B$ such that $o = (p, q)$. Then we have $\varphi([p], q) = [(p, q)] = [o] = E$.

(EQ1): Let $o_1 = (E_1, q_1) \in Q_{\tilde{A}\|B}$, let $o_2 = (E_2, q_2) \in Q_{\tilde{A}\|B}$, and let $\sigma \in \Sigma$. We have $(o_1, \sigma, o_2) \in \Delta_{\tilde{A}\|B}$

$$\begin{aligned}
&\Leftrightarrow \quad (E_1, \sigma, E_2) \in \Delta_{\tilde{A}} \text{ and } (q_1, \sigma, q_2) \in \Delta_B \\
&\Leftrightarrow \quad \text{there exist } p_1 \in E_1, p_2 \in E_2 \text{ with } (p_1, \sigma, p_2) \in \Delta_A, \text{ and } (q_1, \sigma, q_2) \in \Delta_B \\
&\Leftrightarrow \quad \text{there exist } p_1 \in E_1, p_2 \in E_2 \text{ such that } ((p_1, q_1), \sigma, (p_2, q_2)) \in \Delta_{A\|B} \\
&\Leftrightarrow \quad (\varphi(o_1), \sigma, \varphi(o_2)) \in \Delta_{\widetilde{A\|B}}.
\end{aligned}$$

The last step follows by
$(\varphi(o_1), \sigma, \varphi(o_2)) = (\varphi([p_1], q_1), \sigma, \varphi([p_2], q_2)) = ([(p_1, q_1)], \sigma, [(p_2, q_2)]) \in \Delta_{\widetilde{A\|B}}$
if $p_1 \in E_1$ and $p_2 \in E_2$.

(EQ2): Let $o = (E, q) \in Q_{\tilde{A}\|B}$. We have

$$\begin{aligned} o \in J_{\tilde{A}\|B} &\Leftrightarrow E \in J_{\tilde{A}} \text{ and } q \in J_B \\ &\Leftrightarrow \text{there exists } p \in E \text{ such that } p \in J_A, \text{ and } q \in J_B \\ &\Leftrightarrow \text{there exists } p \in E \text{ such that } (p, q) \in J_{A\|B} \\ &\Leftrightarrow \text{there exists } p \in E \text{ such that } [(p, q)] \in J_{\widetilde{A\|B}} \\ &\Leftrightarrow \varphi(o) \in J_{\widetilde{A\|B}}, \text{ since } \varphi(o) = \varphi([p], q) = [(p, q)] \text{ if } p \in E. \end{aligned}$$

(EQ3): The proof for (EQ3) is equivalent to the proof for (EQ2), but we use marked states instead of initial states. □

From the results provided so far, we can now conclude that the introduced reduction can in fact be used when checking for (η, Ξ)-nonblocking: the property to be (η, Ξ)-nice is an invariant, and reduced and original automata are (η, Ξ)-nonblocking equivalent.

Proposition 5.2.43
Let A and B be η-partitioned automata over the alphabet Σ, let $\Xi \subseteq \Sigma$, let $\Upsilon \subseteq \Sigma$ such that $\Upsilon \cap \Xi = \emptyset$ and Υ is not relevant for B, and let $\sim$ be an (Υ, Ξ)-state reduction equivalence for A. Furthermore, let $A \,\|\, B$ be (η, Ξ)-nice. The following assertions hold:

(1) *$A \,\|\, B$ is (η, Ξ)-nonblocking*
if and only if
$(A/{\sim}) \,\|\, B$ is (η, Ξ)-nonblocking.

(2) *$(A/{\sim}) \,\|\, B$ is (η, Ξ)-nice.*

Proof. Let A and B be η-partitioned automata over the alphabet Σ, let $\Xi \subseteq \Sigma$, let $\Upsilon \subseteq \Sigma$ such that $\Upsilon \cap \Xi = \emptyset$ and Υ is not relevant for B, and let $\sim$ be an (Υ, Ξ)-state reduction equivalence for A. Furthermore, let $A \,\|\, B$ be (η, Ξ)-nice. By lemma 5.2.42, there exists an (Υ, Ξ)-state reduction equivalence $\sim_{A\|B}$ for $A \,\|\, B$ such that $(A/{\sim}) \,\|\, B \cong (A \,\|\, B)/{\sim_{A\|B}}$. Property (1) follows by lemma 5.2.40, and property (2) follows by lemma 5.2.41. □

(R4) Reachability Reductions

The reachability part of an η-partitioned automaton can be reduced by using projections. If there exists another automaton with the same coreachability part and a reachability part in which all η-reachable states are reachable by the same set of words by only considering events that are relevant for other automata of the system, then we can use the new automaton instead.

Consider the example in figure 5.3 again. The event set

$$\Upsilon = \{\mathit{CLOSE\_DRD}, \mathit{CLOSE\_PSD}, \mathit{KEY\_OUT}\}$$

is not relevant for automata A_1 and A_3. This enables us to replace automaton A'_2 shown in figure 5.8 by automaton A''_2 in figure 5.9. The coreachability parts of both automata are equal. The set of words, for which an η-reachable state can be reached, projected to events that are not hidden, are equal in both automata. For instance, the word

$$\mathit{OPEN\_DRD}\ \mathit{OPEN\_PSD}\ \mathit{CLOSE\_DRD}\ \mathit{CLOSE\_PSD}\ \eta$$

takes us to state `t0` in automaton A'_2. There must exist a similar word in automaton A''_2 that also leads to state `t0`. In automaton A''_2, the word

$$\mathit{OPEN\_DRD}\ \mathit{OPEN\_PSD}\ \mathit{KEY\_OUT}\ \eta$$

leads to state `t0`. Events *CLOSE_DRD*, *CLOSE_PSD*, and *KEY_OUT* are hidden, thus the projection to events that are not hidden is in both cases the word

$$\mathit{OPEN\_DRD}\ \mathit{OPEN\_PSD}\ \eta.$$

This property must be satisfied for all words leading to an η-reachable state. This kind of reduction can also be applied to automaton A'_1 in figure 5.5 by exploiting that $\Upsilon = \{\mathsf{Timer}, \mathit{TIMEOUT}\}$ is only relevant for A'_1, resulting in automaton A''_1 in figure 5.6.

This idea is now formalised. An η-partitioned automaton is less (Υ, η)-reachable than another η-partitioned automaton if both coreachability parts are equal and every η-reachable state of the first automaton can be reached by a similar word in the second automaton.

Definition 5.2.18 (less (Υ, η)-reachable automaton)
Let A and A' be η-partitioned automata over the same alphabet Σ, let $\Upsilon \subseteq \Sigma$, and let $Q_c = \{q \in Q_{\text{reach}}(A) \mid \eta \notin \textsc{Elig}_A(q)\}$. The automaton A is said to be less (Υ, η)-reachable *than A' if*

- **(LR1)** $Q_c \subseteq Q_{A'}$,
- **(LR2)** $\Delta_A \cap (Q_c \times \Sigma_A \times Q_A) \subseteq \Delta_{A'}$,
- **(LR3)** $M_A \cap Q_c \subseteq M_{A'}$, *and*
- **(LR4)** *for each $q \in Q_A$ and for each $s \in \Sigma^*$ such that $(J_A, s\eta, q) \in \Delta_A$ there exists $t \in \Sigma^*$ such that $P_{\Sigma\setminus\Upsilon}(s) = P_{\Sigma\setminus\Upsilon}(t)$ and $(J_{A'}, t\eta, q) \in \Delta_{A'}$.*

A and A' are (Υ, η)-reachability equivalent *if A is less (Υ, η)-reachable than A' and A' is less (Υ, η)-reachable than A.*

Conditions (LR1), (LR2), and (LR3) ensure that the coreachable part of automaton A is a subautomaton of the coreachable part of A'. Condition (LR4) states that each η-reachable state in A is η-reachable in A'. Furthermore, for each string in A leading to this state there exists a corresponding string in A'.

Note that, if A is less (Υ, η)-reachable than A', then for each $q \in Q_{\text{reach}}(A)$ such that $\eta \notin \textsc{Elig}_A(q)$, we have

$$\textsc{Elig}_A(q) \subseteq \textsc{Elig}_{A'}(q). \tag{5.12}$$

Consider a system consisting of several η-partitioned automata. Now consider one of these automata, called A, and an event set Υ that is only relevant for this automaton. By replacing A by an η-partitioned automaton A' that is (Υ, η)-reachability equivalent we obtain a system that is (η, Ξ)-nonblocking equivalent to the original system. This kind of reduction is called (R4). The automata in figures 5.8 and 5.9 are (Υ, η)-reachability equivalent. Thus, the second automaton is a possible result of reduction (R4) applied to the first automaton.

Like in the case of reduction operations (R1) and (R3), we first show the desired properties for a single automaton and then prove that the results can also be applied to a system consisting of several automata in which a single automaton is reduced. First we show that an automaton is (η, Ξ)-nonblocking if and only if the reduced automaton is (η, Ξ)-nonblocking. This follows from the following lemma, stating that an automaton is (η, Ξ)-nonblocking if it is equivalent to an (η, Ξ)-nonblocking automaton.

Lemma 5.2.44
Let A and A' be η-partitioned automata over the same alphabet Σ, let $\Upsilon, \Xi \subseteq \Sigma$, and let A and A' be (Υ, η)-reachability equivalent.
If A is (η, Ξ)-nonblocking, then A' is (η, Ξ)-nonblocking.

Proof. Let A and A' be η-partitioned automata over the same alphabet Σ, let $\Upsilon, \Xi \subseteq \Sigma$, and let A and A' be (Υ, η)-reachability equivalent. Write $Q_c = \{\, q \in Q_{\text{reach}}(A) \mid \eta \notin \textsc{Elig}_A(q) \,\}$. Assume that A is (η, Ξ)-nonblocking, and let $p_0 \in Q_{A'}$ be η-reachable in A'. By property (LR4) of definition 5.2.18, we obtain that p_0 is η-reachable in A. Since A is (η, Ξ)-nonblocking, there exists a Ξ-controlled and marked path $\pi = p_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} p_n$ in A. We show that this is a Ξ-controlled and marked path in A'. Since A is η-partitioned, we have for each $i \in \{0, \ldots, n\}$ that $p_i \in Q_c$. By property (LR1), we obtain $p_i \in Q_{A'}$ for each $i \in \{0, \ldots, n\}$. Now let $i \in \{0, \ldots, n-1\}$. We have $(p_i, \sigma_{i+1}, p_{i+1}) \in \Delta_A$ such that $p_i \in Q_c$. By property (LR2), we obtain $(p_i, \sigma_{i+1}, p_{i+1}) \in \Delta_{A'}$. Since π is marked, $p_n \in M_A$. By property (LR3), we obtain $p_n \in M_{A'}$. Furthermore, π is Ξ-controlled by equation (5.12). Thus, π is a Ξ-controlled and marked path in A', i.e. A' is (η, Ξ)-nonblocking. $\square$

Now we show that if an η-partitioned automaton is (η, Ξ)-nice, then the reduced automaton is also (η, Ξ)-nice.

Lemma 5.2.45
Let A and A' be η-partitioned automata over the same alphabet Σ, let $\Upsilon, \Xi \subseteq \Sigma$, and let A and A' be (Υ, η)-reachability equivalent.
If A is (η, Ξ)-nice, then A' is (η, Ξ)-nice.

Proof. Let A and A' be η-partitioned automata over the same alphabet Σ, let $\Upsilon, \Xi \subseteq \Sigma$, and let A and A' be (Υ, η)-reachability equivalent. Assume that A is (η, Ξ)-nice and A' is (η, Ξ)-nonblocking. By lemma 5.2.44, we obtain that A is (η, Ξ)-nonblocking. Let Ψ be a live (η, Ξ)-supervisor for A'. Define

$$\begin{aligned} \Phi \colon Q_A &\to 2^{\Sigma} \\ q &\mapsto \begin{cases} \Psi(q), & \text{if } \eta \notin \textsc{Elig}_A(q); \\ \Sigma, & \text{otherwise.} \end{cases} \end{aligned}$$

Since Ψ is a live (η, Ξ)-supervisor for A' and, by equation (5.12), $\textsc{Elig}_A(q) = \textsc{Elig}_{A'}(q)$ for each $q \in Q_{\text{reach}}(A)$ such that $\eta \notin \textsc{Elig}_A(q)$, Φ is a live (η, Ξ)-supervisor for A. Since A is (η, Ξ)-nice, we obtain that Φ/A is (η, Ξ)-nonblocking.

It suffices to show that Ψ/A' is (η, Ξ)-nonblocking. Let $p \in Q_{\Psi/A'}$ be η-reachable in Ψ/A'. Then p is η-reachable in A'. Since A' is less (Υ, η)-reachable than A, p is η-reachable in A. By lemma 5.2.21, this implies that p is η-reachable in Φ/A. Recall that Φ/A is (η, Ξ)-nonblocking, i.e. there exists a Ξ-controlled and marked path in Φ/A with origin p. By definition of Φ and since A and A' are (Υ, η)-reachability equivalent, this is a Ξ-controlled and marked path in Ψ/A'. □

In the following lemma we prove that applying the reduction to a single automaton can be represented as a reduction on the synchronous product automaton.

Lemma 5.2.46
Let A, A', and B be η-partitioned automata over the same alphabet Σ, let $\Upsilon \subseteq \Sigma$, let A be less (Υ, η)-reachable than A', and let Υ be not relevant for B.
Then $A \parallel B$ is less (Υ, η)-reachable than $A' \parallel B$.

Proof. Let A, A', and B be η-partitioned automata over the same alphabet Σ, let $\Upsilon \subseteq \Sigma$, let A be less (Υ, η)-reachable than A', and let Υ be not relevant for B. Write $Q^c_{A\|B} = \{\, o \in Q_{\text{reach}}(A \parallel B) \mid \eta \notin \textsc{Elig}_{A\|B}(o) \,\}$ and $Q^c_A = \{\, q \in Q_{\text{reach}}(A) \mid \eta \notin \textsc{Elig}_A(q) \,\}$.

First we prove that $(p, q) \in Q^c_{A\|B}$ implies $p \in Q^c_A$. Let $(p, q) \in Q^c_{A\|B}$. Since $A \parallel B$ is η-partitioned, there exist $s, w \in \Sigma^*$ such that $(J_{A\|B}, s\eta w, (p, q)) \in \Delta_{A\|B}$. Then we have $(J_A, s\eta w, p) \in \Delta_A$. Since A is η-partitioned, $\eta \notin \textsc{Elig}_A(p)$. Furthermore, p is reachable in A. Therefore, $p \in Q^c_A$. Next, we show that properties (LR1), (LR2), (LR3), and (LR4) are satisfied.

(LR1): Let $(p,q) \in Q^c_{A\|B}$. Then we have $p \in Q^c_A$ and $q \in Q_B$. Since A is less (Υ,η)-reachable than A', we also have $p \in Q_{A'}$. Therefore, $(p,q) \in Q_{A'\|B}$.

(LR2): Let $((p,q),\sigma,(p',q')) \in \Delta_{A\|B} \cap (Q^c_{A\|B} \times \Sigma \times Q_{A\|B})$. Then we have $(p,q) \in Q^c_{A\|B}$, and thus $p \in Q^c_A$. Furthermore, $(p,\sigma,p') \in \Delta_A$ and $(q,\sigma,q') \in \Delta_B$. Since A is less (Υ,η)-reachable than A', we have $(p,\sigma,p') \in \Delta_A \cap (Q^c_A \times \Sigma \times Q_A) \subseteq \Delta_{A'}$. This proves that $((p,q),\sigma(p',q')) \in \Delta_{A'\|B}$.

(LR3): Let $(p,q) \in M_{A\|B} \cap Q^c_{A\|B}$. Then we have $p \in Q^c_A$, $p \in M_A$, and $q \in M_B$. Since A is less (Υ,η)-reachable than A', we have $p \in M_{A'}$. Thus, we have $(p,q) \in M_{A'\|B}$.

(LR4): Let $(p,q) \in Q_{A\|B}$ and $s \in \Sigma^*$ such that $(J_{A\|B}, s\eta, (p,q)) \in \Delta_{A\|B}$. Then we have $(J_A, s\eta, p) \in \Delta_A$ and $(J_B, s\eta, q) \in \Delta_B$. Since A is less (Υ,η)-reachable than A' there exists a string $t \in \Sigma^*$ such that $P_{\Sigma\setminus\Upsilon}(s) = P_{\Sigma\setminus\Upsilon}(t)$ and $(J_{A'}, t\eta, p) \in \Delta_{A'}$. Since Υ is not relevant for B, $(J_B, s\eta, q) \in \Delta_B$ implies $(J_B, t\eta, q) \in \Delta_B$, so that $(J_{A'\|B}, t\eta, (p,q)) \in \Delta_{A'\|B}$. $\square$

Finally, we state that the introduced reduction behaves as expected: if the system is (η,Ξ)-nice, then the reduced system is (η,Ξ)-nice and (η,Ξ)-nonblocking equivalent to the original system.

Proposition 5.2.47
Let A, A', and B be η-partitioned automata over the alphabet Σ, let $\Upsilon \subseteq \Sigma$ such that A and A' are (Υ,η)-reachability equivalent and Υ is not relevant for B. Furthermore, let $\Xi \subseteq \Sigma\setminus\{\eta\}$ such that $A \,\|\, B$ is (η,Ξ)-nice. The following assertions hold:

(1) *$A \,\|\, B$ is (η,Ξ)-nonblocking*
if and only if
$A' \,\|\, B$ is (η,Ξ)-nonblocking.

(2) *$A' \,\|\, B$ is (η,Ξ)-nice.*

Proof. Let A, A', and B be η-partitioned automata over the alphabet Σ, let $\Upsilon \subseteq \Sigma$ such that A and A' are (Υ,η)-reachability equivalent and Υ is not relevant for B. Furthermore, let $\Xi \subseteq \Sigma\setminus\{\eta\}$ such that $A \,\|\, B$ is (η,Ξ)-nice. Then we have $A \,\|\, B$ and $A' \,\|\, B$ are (Υ,η)-reachability equivalent by lemma 5.2.46. We obtain that $A \,\|\, B$ is (η,Ξ)-nonblocking if and only if $A' \,\|\, B$ is (η,Ξ)-nonblocking by lemma 5.2.44. This shows that the first assertion holds. By lemma 5.2.45, we obtain that $A' \,\|\, B$ is (η,Ξ)-nice, so that the second assertion is proven. $\square$

An Incremental Algorithm

In the previous sections, we have provided a transformation from deterministic automata to η-partitioned automata and several approaches how these automata can be reduced. In this section, the results are summarised and used to outline an incremental algorithm for checking if a system is Ξ-nonblocking.

In the following definition, we summarise the introduced reductions into one reduction relation denoted by $\Rightarrow$.

Definition 5.2.19 (reduction relation $\Rightarrow$)
Let Σ be an alphabet of events, let $\eta \in \Sigma$, and let $\Xi \subseteq \Sigma$ such that $\eta \notin \Xi$. Define the following relations over sets of η-partitioned automata that are defined over the alphabet Σ:

(R1) Let $\{A_1, A_2, \ldots, A_n\} \Rightarrow_{R1} \{\Phi/A_1, A_2, \ldots, A_n\}$ if there exists $\Upsilon \subseteq \Xi$ such that

- *Φ is an Υ-selective live (η, Ξ)-supervisor for A_1 and*
- *for all $k \in \{2, \ldots, n\}$ we have that Υ is not relevant for A_k.*

(R2) Let $\{A_1, A_2, \ldots, A_n\} \Rightarrow_{R2} \{\mathrm{RED}_{r,\upsilon}(A_1), A_2, \ldots, A_n\}$ if

- *$\upsilon \in \Xi$, $r \in Q_{A_1}$ is (υ, Ξ)-reducible, and*
- *for all $k \in \{2, \ldots, n\}$ we have that υ is not relevant for A_k.*

(R3) Let $\{A_1, A_2, \ldots, A_n\} \Rightarrow_{R3} \{A_1/\!\sim, A_2, \ldots, A_n\}$ if there exists $\Upsilon \subseteq \Sigma$ such that $\Upsilon \cap \Xi = \emptyset$,

- *$\sim$ is a (Υ, Ξ)-state reduction equivalence for A_1, and*
- *for all $k \in \{2, \ldots, n\}$ we have that Υ is not relevant for A_k.*

(R4) Let $\{A_1, A_2, \ldots, A_n\} \Rightarrow_{R4} \{A_1', A_2, \ldots, A_n\}$ if there exists $\Upsilon \subseteq \Sigma$ such that

- *A_1 and A_1' are (Υ, η)-reachability equivalent and*
- *for all $k \in \{2, \ldots, n\}$ we have that Υ is not relevant for A_k.*

(P) Furthermore, let $\{A_1, A_2, A_3, \ldots, A_n\} \Rightarrow_P \{A_1 \,\|\, A_2, A_3, \ldots, A_n\}$.

Now define

$$\Rightarrow \;=\; \Rightarrow_{R1} \cup \Rightarrow_{R2} \cup \Rightarrow_{R3} \cup \Rightarrow_{R4} \cup \Rightarrow_P .$$

The following proposition provides the basis for an incremental algorithm. It states that an automaton that is the result of a sequence of synchronous product operations and reductions is (η, Ξ)-nonblocking if and only if the original system is Ξ-nonblocking.

Proposition 5.2.48
Let $G_1, \ldots, G_n$ be deterministic automata over the alphabet Σ, let $\Xi \subseteq \Sigma$, and let $\eta \in \Sigma \backslash \Xi$ be, for all $k \in \{1, \ldots, n\}$, irrelevant for G_k. Furthermore, let $G_1 \,\|\, \cdots \,\|\, G_n$ be Ξ-terminating and Ξ-confluent, and let

$$\{G_1 \,\|\, A^\eta_\Sigma, \ldots, G_n \,\|\, A^\eta_\Sigma\} \overset{*}{\Rightarrow} \{A\}.$$

Then

$$G_1 \parallel \cdots \parallel G_n \text{ is } \Xi\text{-nonblocking}$$
$$\text{if and only if}$$
$$A \text{ is } (\eta, \Xi)\text{-nonblocking.}$$

Proof. Let $G_1, \ldots, G_n$ be deterministic automata over the same alphabet Σ, let $\Xi \subseteq \Sigma$, and let $\eta \in \Sigma \backslash \Xi$ be, for all $k \in \{1, \ldots, n\}$, irrelevant for G_k. Furthermore, let $G_1 \parallel \cdots \parallel G_n$ be Ξ-terminating and Ξ-confluent.

First we show that $(G_1 \parallel A^\eta_\Sigma) \parallel \cdots \parallel (G_n \parallel A^\eta_\Sigma)$ is (η, Ξ)-nice and, furthermore, $G_1 \parallel \cdots \parallel G_n$ is Ξ-nonblocking if and only if $(G_1 \parallel A^\eta_\Sigma) \parallel \cdots \parallel (G_n \parallel A^\eta_\Sigma)$ is (η, Ξ)-nonblocking. But this is proven in propositions 5.2.25 and 5.2.27, since the synchronous product is associative and commutative and $A^\eta_\Sigma \parallel A^\eta_\Sigma \cong A^\eta_\Sigma$.

It suffices to show that, starting with a tuple $(A_1, \ldots, A_n)$ of η-partitioned automata such that $A_1 \parallel \cdots \parallel A_n$ is (η, Ξ)-nice, each $\Rightarrow$-derivation yields a tuple of automata $(B_1, \ldots, B_m)$ such that $B_1 \parallel \cdots \parallel B_m$ is (η, Ξ)-nice, and $B_1 \parallel \cdots \parallel B_m$ is (η, Ξ)-nonblocking if and only if $A_1 \parallel \cdots \parallel A_n$ is (η, Ξ)-nonblocking. For reduction operation (R1) this is shown in proposition 5.2.31, for reduction operation (R2) this is proven in propositions 5.2.35 and 5.2.36, for reduction operation (R3) this is shown in proposition 5.2.43, and proposition 5.2.47 provides the necessary results for reduction operation (R4). Furthermore, computing the synchronous product, i.e. applying operation (P), does not change these properties. □

This result leads directly to an incremental algorithm for checking whether a synchronous product of deterministic automata is Ξ-nonblocking or not. Given a set $\{G_1, \ldots, G_n\}$ of automata to be checked, we first construct the corresponding η-partitioned automata $\{G_1 \parallel A^\eta_\Sigma, \ldots, G_n \parallel A^\eta_\Sigma\}$. Then we search this list for an automaton to which we can apply one of the reductions (R1) through (R4). If there is such an automaton, we apply the appropriate reduction operation, replace the automaton by the result of the reduction, and continue to look for further reductions. If there is no more reduction possible, then we have to select two automata from the list, and replace them by their synchronous product. If only one automaton remains in the list, we can compute the result by checking if each η-reachable state is Ξ-coreachable.

If this algorithm finds that a model is Ξ-blocking, it is also possible to compute an appropriate counter-example. In order to do this, we start with the counter-example obtained for the final automaton. Then we have to backtrack through all the reduction steps performed, and at each step have to re-insert the events projected out at the appropriate positions in the counter-example.

Many improvements can be applied to this algorithm. More reduction operations can be defined. For instance, it is possible to remove some more transitions labelled with events that cannot be observed by other automata. Several heuristics for finding an automaton with which the actual subsystem is incremented can be considered. It is important to select automata in such a way that the reductions can be most efficiently applied. Another source of improvements is that the two parts of an η-partitioned automaton are often quite similar. It could be investigated in reductions that can be performed on both parts such that the doubling operation is only necessary in certain cases.

Chapter 6

Case Studies

In this chapter, the design process of two discrete controllers is presented. It is demonstrated how the properties introduced in this thesis can be integrated in this process. We will see that design errors and incomplete specification can be found by checking if a model is Ξ-terminating, Ξ-confluent, and Ξ-nonblocking. Since these checks can be performed automatically, they provide powerful means of verification.

In section 6.1, we develop a controller for a batch plant, which is often used in literature. The basic steps involved in the design process of a controller are explained in detail. The manageable size of this example allows the presentation of the complete model by means of a single automaton. This enables us to demonstrate and motivate the main problems and concepts developed in this thesis. This example also shows that the suggested approach is applicable to examples of practical interest.

In section 6.2, a more elaborate industrial example is treated. The modelling process of a comfort function and a window locking system of a car is performed. Several design errors are found by applying the new ideas. A fixed model is proposed, which is nonblocking for each derived controller. This example shows that the approach proposed in this thesis can be applied to quite large real-world systems.

6.1 Controlling a Chemical Dosing Unit

The batch plant example considered in this section is taken from [HK94]; a more elaborate version of this model is discussed in [DMWB02]. We model a dosing unit as it is used in chemical batch plants to supply a defined amount of liquid material to subsequent process units. The dosing unit, shown in figure 6.1, consists of a tank, a stirrer, an inlet valve, an outlet valve, and two level sensors.

The task of the controller to be designed is to give open and close commands to the valves in a way that the right amount of liquid is provided. Furthermore,

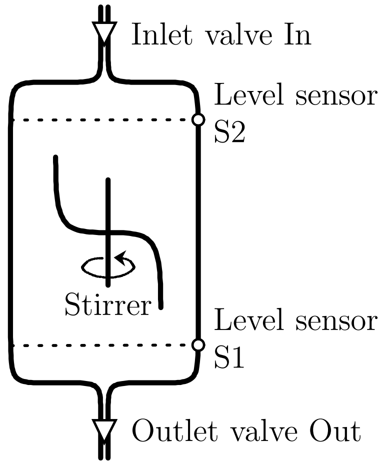

Figure 6.1: The dosing unit.

the fluid must be stirred, since it may gelatinise if it is not in motion. The fluid may become jelly when both valves are closed, the tank is not empty, and the stirrer is not running[1].

Supervisory control theory provides the basis to solve this control problem. Given a plant model and a specification, we are able to compute the least restricted behaviour that satisfies the specification and is nonblocking. We will see in the following that this model is not suitable to generate a controller, i.e. software that performs the control task, from it. Checking the properties introduced in this thesis provides a counter-example which reveals problems in the design that must be refined further. A refined model is developed. From the refined model, which satisfies the desired properties, we generate the controller. The results of this thesis make sure that the generated controller is nonblocking.

In a first step, we develop a plant model of the dosing unit. The automaton in figure 6.2 models the behaviour of the level sensors. We distinguish three states. The tank is empty in state `empty` when both sensors are off. It is partially filled in state `partial` when sensor S1 is on and sensor S2 is off. The tank is full in state `full` when both sensors are on. Initially, the tank is empty. Physically, it is not possible that sensor S2 is on at the same time as sensor S1 is off. State changes are indicated by the uncontrollable events *S1_ON*, *S1_OFF*, *S2_ON*, and *S2_OFF*. The uncontrollable event *JELLY* models the gelatinising process. This event is only eligible when the tank is not empty.

Both valves can be opened or closed. The behaviour of the inlet valve is modelled by the automaton in figure 6.3; the behaviour of the outlet valve is modelled by the automaton in figure 6.4. The events OpenIn and OpenOut for opening the valves and CloseIn and CloseOut for closing the valves are controllable.

[1]In this model, different from the original example, the fluid can become jelly when the tank is full.

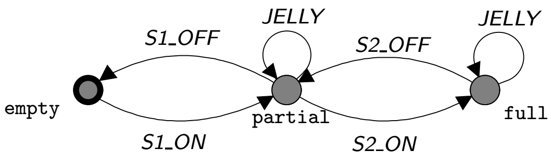

Figure 6.2: The level measuring sensors.

The level sensors can only change their state from off to on, i.e. events *S1_ON* and *S2_ON* are eligible, when the inlet valve is open. Conversely, the level sensors can only change their state from on to off, i.e. events *S1_OFF* and *S2_OFF* are eligible, when the outlet valve is open. The gelatinising process may only take place when the valves are closed. Initially, both valves are closed.

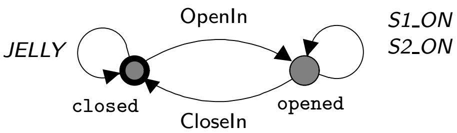

Figure 6.3: The inlet valve.

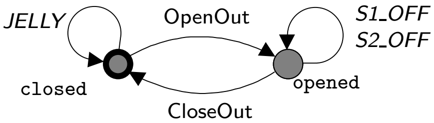

Figure 6.4: The outlet valve.

The stirrer is modelled in figure 6.5. It can be switched on, indicated by the controllable event StirrerOn, and switched off, indicated by the controllable event StirrerOff. Initially, the stirrer is switched off. The fluid may only turn to jelly when the stirrer is off.

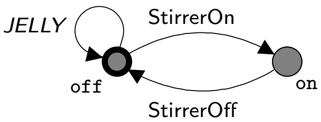

Figure 6.5: The stirrer.

The plant behaviour, i.e. the physically possible behaviour of the system, is described by the synchronous product of the automata in figures 6.2, 6.3, 6.4, and 6.5. Recall that only relevant events are given in the graph representations of the automata.

In a next step, we impose certain restrictions on the plant behaviour by modelling the desired behaviour of the dosing unit. In order to provide the right quantity of substance to a subsequent process, the tank must be filled to the upper level, i.e. until sensor S2 goes on, and then must be discharged until it is empty, i.e. until sensor S1 goes off. This is modelled by the automaton in figure 6.6. Furthermore, it is not allowed to fill the tank partially and then open the outlet valve; neither is it allowed to empty it partially and then open the inlet

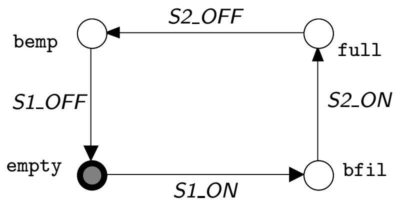

Figure 6.6: Specification: empty-full-empty-cycle.

valve. Furthermore, it must be avoided that both valves are open at the same time or the fluid gelatinises. This is modelled by the automaton in figure 6.7.

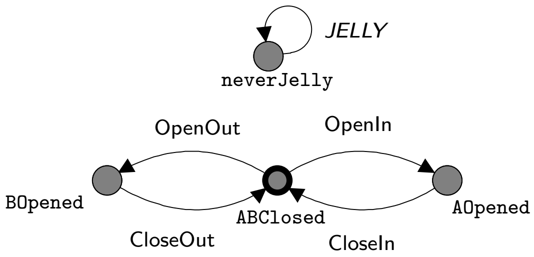

Figure 6.7: Specification: no passing through, never jelly.

The task of the controller to be designed is to fill and empty the tank. In order to formalise this, we mark all states for which the tank is empty. This means that a task is completed when the tank is emptied or, in other words, when an empty-full-empty-cycle is completed.

Synthesising the least restrictive nonblocking supervisor for the given specification provides the model in figure 6.8. If the valves are opened and closed manually, opening and closing can be enabled or disabled by the controller in order to ensure that the specification is satisfied. However, we are interested in a controller which sends open and close commands to the valves as well as on and off commands to the stirrer. We identify the set of output events of the controller as the set of controllable events, i.e. we consider

$$\Xi = \{\mathsf{OpenIn}, \mathsf{OpenOut}, \mathsf{CloseIn}, \mathsf{CloseOut}, \mathsf{StirrerOn}, \mathsf{StirrerOff}\}.$$

But deriving such a controller from the model may result in a blocking system. At the initial state `s0` in the automaton in figure 6.8, the output events OpenIn, OpenOut, and StirrerOn are enabled, i.e. the corresponding commands can be sent by the controller. The controller which always sends an OpenOut command at this state gets stuck in a loop, opening and closing the outlet valve forever and preventing any progress. But even if a controller sends an OpenIn command, the CloseIn command could be sent immediately after this. Obviously, this is not the

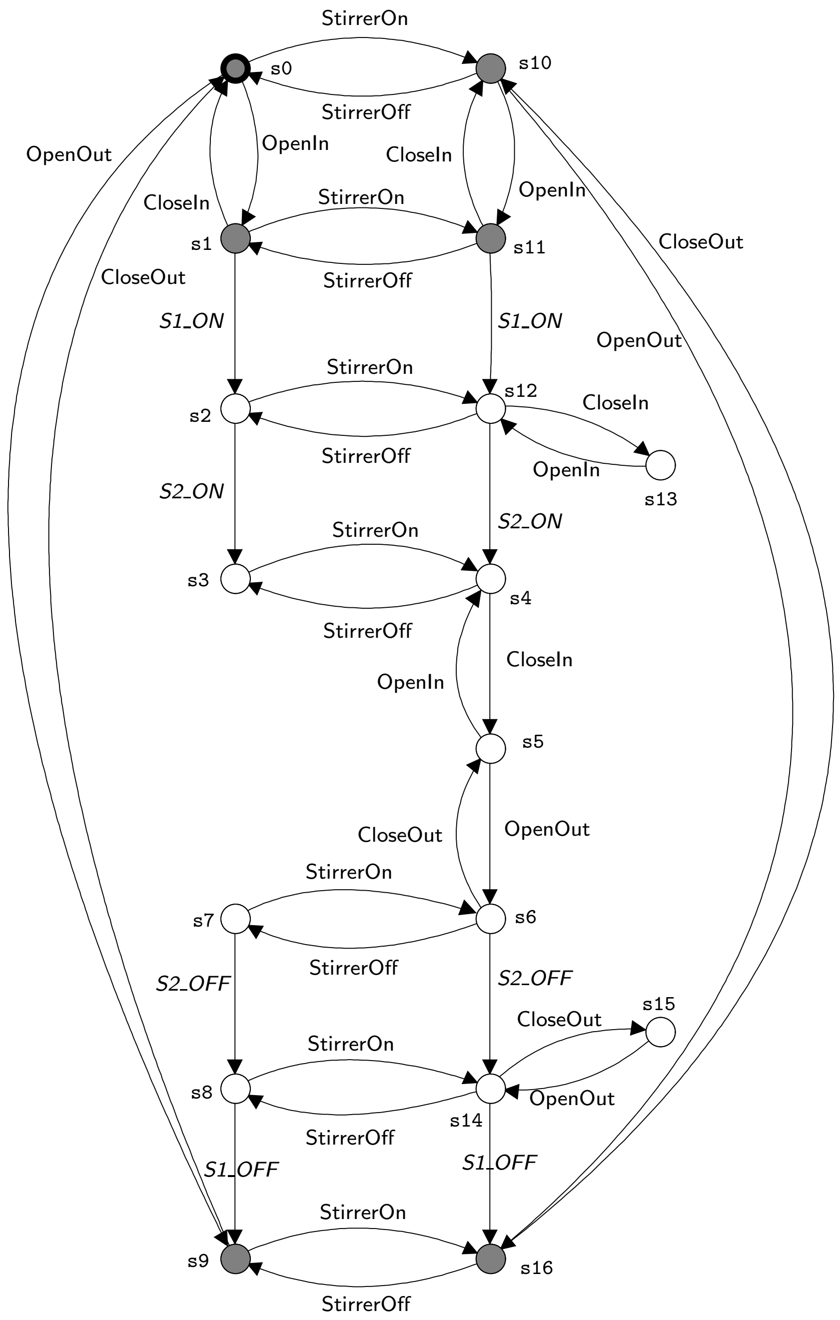

Figure 6.8: A first model.

desired behaviour of our controller. Additional effort is needed in order to obtain a controller which behaves as desired.

The properties introduced in this thesis support the designer in finding these problems. Counter-examples give hints how the model can be refined and the desired controller can be obtained.

The model in figure 6.8 is neither Ξ-terminating nor Ξ-nonblocking. A checking algorithm for Ξ-termination returns an iterative Ξ-pair as a counter-example, for instance $(\varepsilon, \textsf{StirrerOn StirrerOff})$ or $(\varepsilon, \textsf{OpenOut CloseOut})$. Switching the stirrer on and off again and again is not the desired behaviour of the controller. Thus, by checking for Ξ-termination we can find out where the model is still incomplete.

In order to fix these problems, we refine the model by adding more specifications. We model more precisely under which conditions the valves should be opened and closed and the stirrer should be switched on and off. For instance, it is not necessary to open the outlet valve or to switch the stirrer on when the tank is empty. We want to open the inlet valve when the tank is empty, and close it when the tank is full. The outlet valve should only be opened when the tank is full, and closed when the tank is empty, as shown in figure 6.9. Switching the stirrer on is only necessary when the tank is full. In all other cases, the stirrer should be switched off, as shown in figure 6.5.

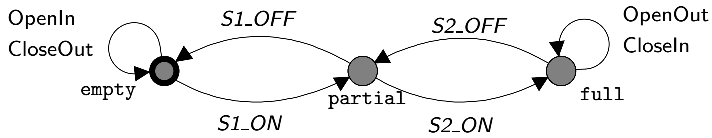

Figure 6.9: Specification: the valves.

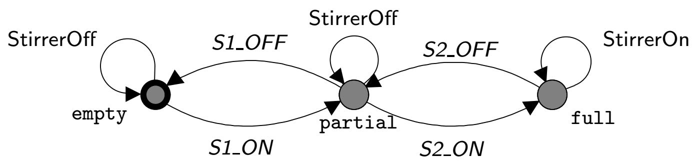

Figure 6.10: Specification: the stirrer.

Synthesising the least restrictive nonblocking supervisor which ensures that the specifications in figures 6.6, 6.7, 6.9 and 6.10 are satisfied provides the refined model in figure 6.11. This system is Ξ-terminating, Ξ-confluent, and Ξ-nonblocking; that is, it satisfies all three properties introduced in this thesis.

Firstly, it does not contain Ξ-loops and is therefore Ξ-terminating. Secondly, whenever two or more output events lead to different states, there exist Ξ-paths leading to a common target state. For instance, at state `s10`, the output events

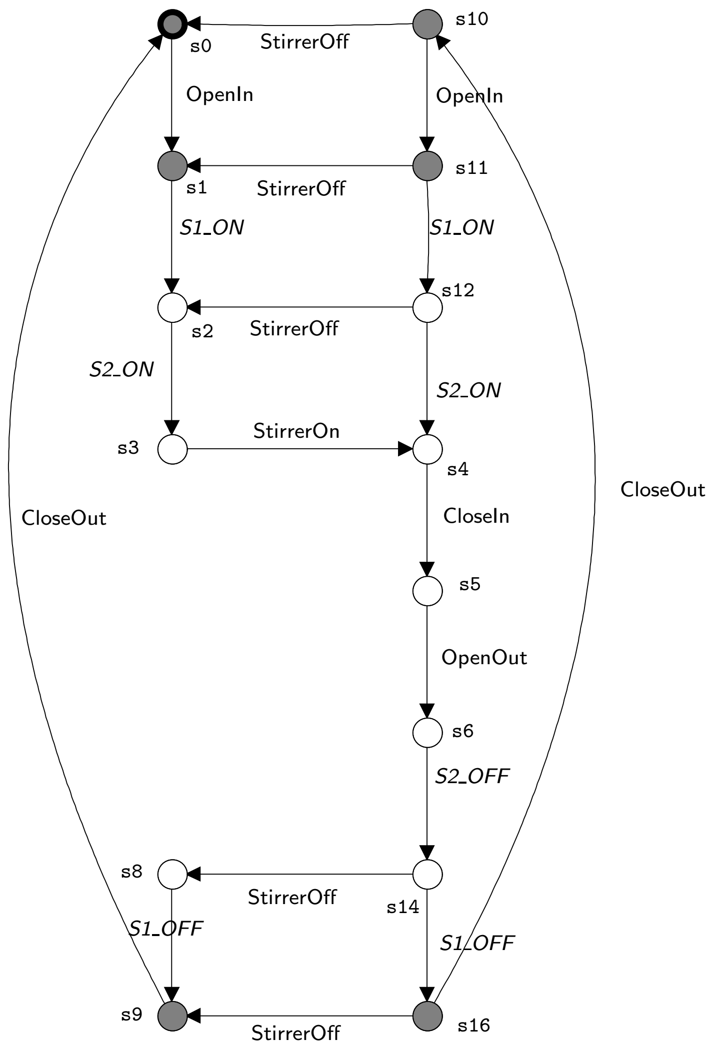

Figure 6.11: Refined model.

StirrerOff and OpenIn are eligible, leading to states `s0` and `s11`. But it is always possible to reach state `s1` by means of output events only: from state `s0` by event OpenIn, from state `s11` by event StirrerOff. The behaviour of the system does not depend on whether a controller switches the stirrer off first and then opens the inlet valve or the other way round. Thus, the system is Ξ-confluent.

Finally, the restricted model is Ξ-nonblocking; it is always possible to complete the task by preferring output events. State `s1` represents the completed task; it is marked and no output event is eligible. This state is reachable from every other state using a Ξ-controlled path, i.e. by a sequence of events in which output events are preferred. Thus, the controller can always complete a cycle of filling and emptying the tank by sending commands and without the need of special intervention from the plant. Note that state `s0` is not suitable for defining a completed task, since an output event is eligible here.

The desired controller can be generated from figure 6.11 as follows. At each state in which an output event is eligible, the controller sends a command corresponding to an eligible event and changes the state based on the chosen command. If there are only input events eligible, the controller waits until an input arrives and changes the state respectively.

Note that there exist states in which more than one output event is eligible. For instance, events CloseOut and StirrerOff are eligible at state s16. We have proven in section 4.5 of this thesis that each controller for a Ξ-terminating, Ξ-confluent, and Ξ-nonblocking model is necessarily nonblocking. Therefore, every controller derived from the model in figure 6.11 is able to complete the current task, i.e. to empty the tank. It does not matter whether we implement the controller which always closes the outlet valve first and switches the stirrer off later or the controller which first switches the stirrer off and then closes the outlet valve.

Note that there are an output and an input event eligible at state s16. Such a state may be affected by the communication problem introduced in section 2.1. In this case, the controller is not affected by the communication problem, since it does not matter in which order the events StirrerOff and *S1_OFF* occur.

6.2 The Window Control System of a Car

In this section, the modelling process for some parts of car electronics is described. This example is taken from a case study done for a component supplier for the automotive industry. We develop a controller for a comfort function and a locking system responsible for the front door windows, the rear door windows, and the sunroof of a car. Several sensors provide the necessary information of the car status. The task of the controller is to generate commands for locking, unlocking, opening, and closing the windows and the sunroof.

An informal task description of the car and the desired controller is given in section 6.2.1. From this description, a model based on supervisory control theory is derived. The relevant car components are modelled in section 6.2.2. The comfort function is formally specified in section 6.2.3, and a model of the locking system is developed in section 6.2.4.

In section 6.2.5, it is checked if this model satisfies the three properties introduced in this thesis, i.e. if the developed model is Ξ-terminating, Ξ-confluent, and Ξ-nonblocking. The counter-examples obtained from these checks point out some subtle errors in the design. Guided by the counter-examples, a corrected model is given.

This application shows that the properties introduced in this thesis provide powerful and helpful guidance during the process of designing a correct controller. They do not only provide the basis for generating nonblocking controllers, but are also able to disclose errors or incomplete specification in the design.

6.2.1 Informal Task Description

Comfort function and locking system of a car are responsible for opening, closing, locking, and unlocking the front door windows, the rear door windows, and the sunroof. Their behaviour depends on the status of the front doors (whether open or closed), the position of the key in the ignition switch (figure 6.12), the buttons requesting a comfort function, and the country configured before the delivery of the car.

Position 0 ('off' position) It is possible to remove the key only in this position.

Position 1 ('on' position) Normal position for driving.

Position 2 ('starter' position) As long as the key is held in this position, the starter is engaged.

Figure 6.12: The ignition switch of the car.

The Comfort Function

The *comfort function* (figure 6.13) provides the possibility to open or close all windows and the sunroof remotely. By using a remote control or buttons on the dashboard, the comfort function can be started. It starts after two seconds delay, provided that the front doors are closed and the ignition key is not inserted. The opening or closing of the windows and the sunroof is initiated successively, starting with the rear door windows and followed, with two seconds delay, by the front door windows. At last, again with two seconds delay, the sunroof is opened or closed. During the whole procedure, the request button has to remain pressed. Releasing the button, opening one of the front doors, or inserting the key into the ignition switch causes the comfort function to be aborted, i.e. the opening or closing is stopped.

The Locking System

The task of the *locking system* is to lock and unlock the windows and the sunroof in a way depending on the country configured before the delivery of the car. When *unlocked*, buttons on the doors, next to the sunroof, and on the dashboard can be used for opening and closing. If the windows and the sunroof are *locked*, these buttons are locked, and opening and closing can only be performed by means of the comfort function.

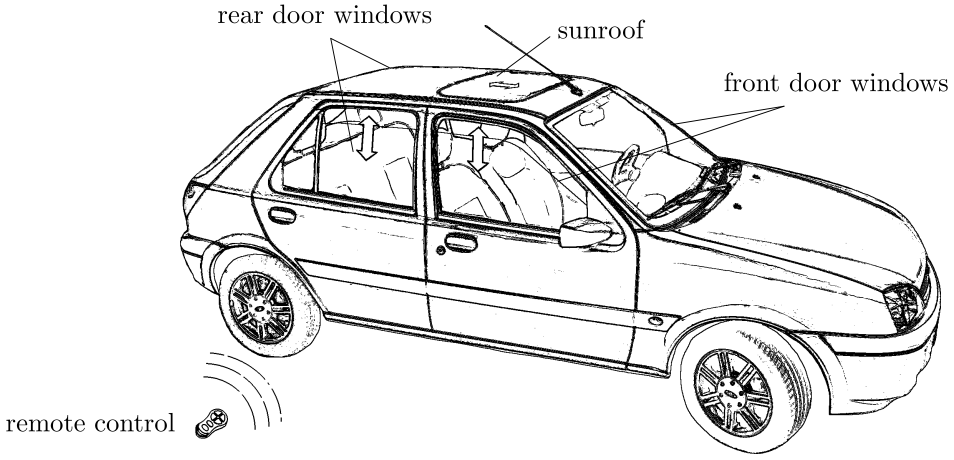

Figure 6.13: The comfort function of the car.

The decision whether the windows and the sunroof should be locked or unlocked is based on the following conditions.

(C1) The key in the ignition switch is in the 'on' position.

(C2) The key in the ignition switch is in the 'starter' position.

(C3) A comfort function is started.

Initially, all windows and the sunroof are locked. They are unlocked when the ignition key is turned to the 'on' position, i.e. condition (P1) is satisfied. As long as the ignition key is in the 'starter' position, i.e. condition (C2) is satisfied, the windows and the sunroof must be locked because of the starter's energy requirements. After turning the key back to the 'on' position, the windows and the sunroof are unlocked again. When a comfort function is started, i.e. condition (C3) is satisfied, the windows and the sunroof are locked and remain locked until the ignition key is turned to the 'on' position, i.e. condition (C1) is satisfied.

Before the delivery of a car, one of the following countries is configured.

- NO COUNTRY: the country is not specified.
- EUROPE: the car is intended for use in Europe.
- USAUS: the car is intended for use in the USA or Australia.

The behaviour of the locking system depends on the country configured. If the country is not specified, the locking system behaves as described above and depends only on conditions (C1), (C2), and (C3). If the car is intended for use in Europe, USA, or Australia, additional conditions are introduced:

(C4) At least one of the front doors was opened and the last of this doors has been closed while no key is in the ignition switch or the ignition key is in the ‘off’ position.

(C5) At least one of the front doors is open.

If the car is intended for use in Europe, the windows and the sunroof are locked and unlocked as described above. In addition, they are also locked when the car is parked, i.e. the last front door has been closed while the ignition key is in the ‘off’ position or not inserted, i.e. condition (C4) is satisfied. Then they remain locked until the ignition key is turned to the ‘on’ position, i.e. condition (C1) is satisfied.

If the car is intended for use in USA or Australia, the windows and the sunroof are locked as long as one of the front doors is open, i.e. condition (C5) is satisfied. If both front doors are closed, the locking system behaves as if no country is specified.

A summary of the behaviour of the locking system is given in figure 6.14. The table shows for each possible country which action is to be taken if the indicated conditions become true.

NO COUNTRY	
(C1)	unlock windows and sunroof
(C2) or (C3)	lock windows and sunroof
EUROPE	
(C1)	unlock windows and sunroof
(C2), (C3), or (C4)	lock windows and sunroof
USAUS	
(C1)	unlock windows and sunroof
(C2) or (C3)	lock windows and sunroof
(C5)	lock as long as (C5) is true

Figure 6.14: The locking system of a car.

6.2.2 The Plant Model

Comfort function and locking system of the car depend on the status of the front doors, the position of the key in the ignition switch, the buttons requesting the comfort function, and the country configured as described in section 6.2.1. Sensors, modelled by means of uncontrollable events listed in figure 6.15, provide the necessary information. The actuators for opening, closing, stopping, locking, and unlocking the windows, controlled by the comfort function and the locking system, are modelled by means of controllable events and are also listed in figure 6.15.

Uncontrollable Events	
CLOSE_DRD	The driver door has been closed.
CLOSE_PSD	The passenger door has been closed.
CMFT_REQUEST_CLOSE	One of the buttons for requesting a comfort function (for closing) has been pressed.
CMFT_REQUEST_OPEN	One of the buttons for requesting a comfort function (for opening) has been pressed.
CMFT_REQUEST_STOP	The button requesting a comfort function has been released.
EUROPE	A European country is configured.
KEY_IN	The key has been inserted into the ignition switch.
KEY_OUT	The key has been removed from the ignition switch.
NO_COUNTRY	No country is configured.
OPEN_DRD	The driver door has been opened.
OPEN_PSD	The passenger door has been opened.
STARTER_OFF	The ignition key has been turned from the 'starter' position to the 'on' position.
STARTER_ON	The ignition key has been turned from the 'on' position to the 'starter' position.
TIMEOUT	A timeout has occurred.
TURN_OFF	The ignition key has been turned from the 'on' position to the 'off' position.
TURN_ON	The ignition key has been turned from the 'off' position to the 'on' position.
USAUS	USA or Australia is configured.
Controllable Events	
CloseFrontDoorWindows	The front door windows start to close.
CloseRearDoorWindows	The rear door windows start to close.
CloseSunroof	The sunroof starts to close.
Lock	Windows and sunroof are locked.
OpenFrontDoorWindows	The front door windows start to open.
OpenRearDoorWindows	The rear door windows start to open.
OpenSunroof	The sunroof starts to open.
ReadCountry	The country configured is read.
Reset	The timer is reset.
StopLifts	Windows and sunroof lifts are stopped.
Timer	The timer is started.
Unlock	Windows and sunroof are unlocked.

Figure 6.15: Car example—events.

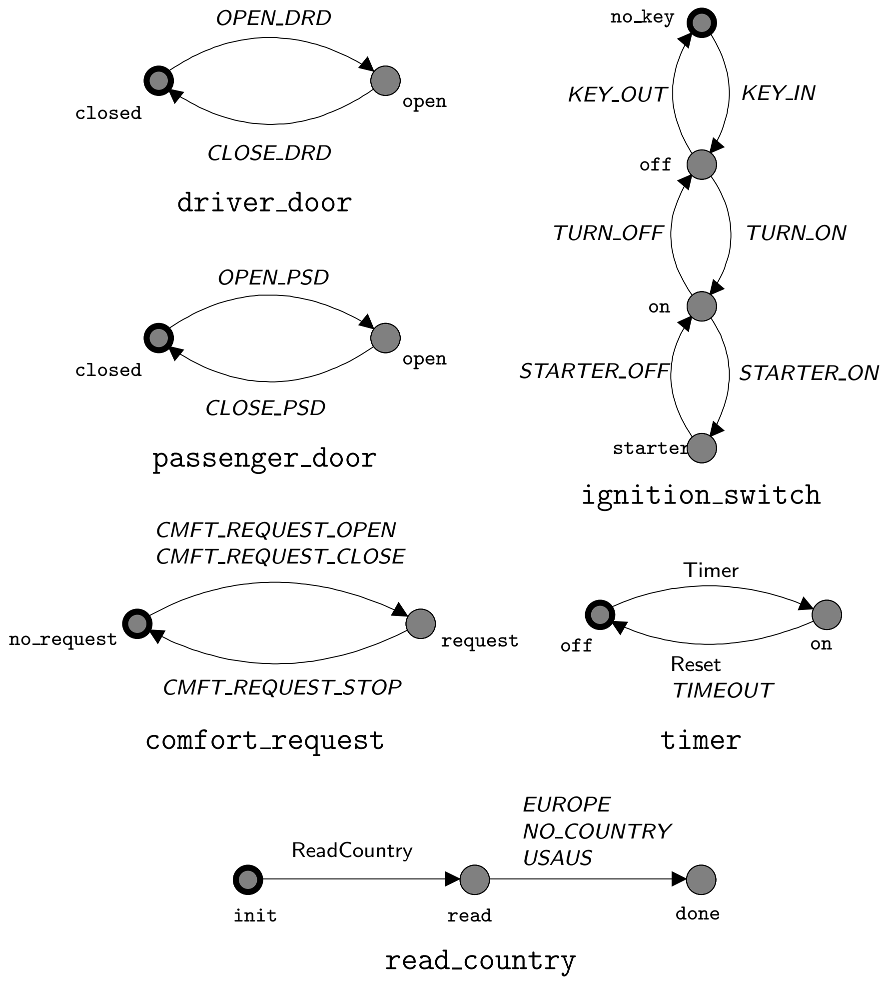

Figure 6.16: Car example—the plant model.

A modular plant model is given in figure 6.16. Automaton `driver_door` models the driver door; automaton `passenger_door` models the passenger door. The doors can be open or closed. Opening a door, which is only possible if the respective door is closed, is modelled by means of the uncontrollable events *OPEN_DRD* and *OPEN_PSD*. Closing a door, which is only possible if the corresponding door is open, is modelled by means of the uncontrollable events *CLOSE_DRD* and *CLOSE_PSD*.

Automaton `ignition_switch` in figure 6.16 models the ignition switch. We distinguish four states: state `no_key`, where no key is inserted, and states `off`, `on`, and `starter` which correspond to the key positions shown in figure 6.12. The key has to be inserted, modelled by the uncontrollable event *KEY_IN*, before it can be turned to the 'on' position, modelled by the uncontrollable event *TURN_ON*. The key must be at the 'on' position, before it can be turned to the 'starter' position, modelled by *STARTER_ON*. Conversely, first the key has to be moved to the 'on' position, modelled by *STARTER_OFF*, then it can be moved to the 'off' position, modelled by *TURN_OFF*, and afterwards the key can be removed from the ignition switch, modelled by *KEY_OUT*.

The comfort function can be requested by using a remote control or by using buttons on the dashboard. In this model, we do not distinguish requests from different locations and assume that there is at most one request at a time. In the automaton `comfort_request` in figure 6.16, the uncontrollable event *CMFT_REQUEST_CLOSE* models a closing request, and the uncontrollable event *CMFT_REQUEST_OPEN* models an opening request. Releasing the button that requests a comfort function is modelled by means of the uncontrollable event *CMFT_REQUEST_STOP*.

We need a timer to model the time delays that occur during a comfort function. Although it is modelled as a plant, the timer may be implemented in software and added to the controller functionality. Automaton `timer` in figure 6.16 provides a timer model. The timer can be started by the controllable event Timer. After starting the timer, an uncontrollable event *TIMEOUT* will occur after two seconds unless the timer is reset modelled by the controllable event Reset.

The locking system depends on the country configured before the delivery of the car. After the first start of the car electronics, this setting is read once. The automaton `read_country` models this behaviour. Reading the country is modelled by the controllable event ReadCountry. Possible values, modelled as uncontrollable events, are *NO_COUNTRY* if none is configured, *EUROPE* if Europe is configured, and *USAUS* if USA or Australia is configured.

Note that most controllable events are not restricted by the plant. They may physically occur at any time and do not influence the status of the plant components. Therefore, most of the controllable events listed in figure 6.15 do not occur in the graph representation of the automata for the plant model in figure 6.16. Furthermore, all states of the plant automata are marked. We introduce markings in order to describe completed tasks when adding specifications.

6.2.3 The Comfort Function

The comfort function provides the possibility to open or close all windows and the sunroof. Its behaviour is described informally in section 6.2.1. A formal specification by means of automata, which is derived from the informal description, is given in figure 6.17.

The sequential opening or closing of the rear door windows, the front door windows, and the sunroof is modelled by means of the automaton `comfort_function`. After a request for closing (event *CMFT_REQUEST_CLOSE*) or opening (event *CMFT_REQUEST_OPEN*),

1. the timer is started (event Timer),
2. the timeout occurs (event *TIMEOUT*),
3. the comfort function is started (internal[2] event comfort_start)
4. the rear door windows are opened (event OpenRearDoorWindows) or closed (event CloseRearDoorWindows),
5. the timer is started (event Timer),
6. the timeout occurs (event *TIMEOUT*),
7. the front door windows are opened (event OpenFrontDoorWindows) or closed (event CloseFrontDoorWindows)
8. the timer is started (event Timer),
9. the timeout occurs (event *TIMEOUT*), and
10. the sunroof is opened (event OpenSunroof) or closed (event CloseSunroof).

This procedure is interrupted at each state by releasing the button requesting the comfort function, which is modelled by event *CMFT_REQUEST_STOP*.

Opening and closing requests of the comfort function are distinguished by the automaton `comfort_open_close` given in figure 6.17. After an opening request (event *CMFT_REQUEST_OPEN*), the rear and front door windows and the sunroof are opened (events OpenRearDoorWindows, OpenFrontDoorWindows, and OpenSunroof). After an closing request (*CMFT_REQUEST_CLOSE*), the rear and front door windows and the sunroof are closed (events CloseRearDoorWindows, CloseFrontDoorWindows, and CloseSunroof).

The comfort function can be only performed if the front doors are closed and there is no key in the ignition switch. Furthermore, if a front door is opened

[2] An internal event is not associated to an actuator or sensor. The internal event comfort_start is introduced to provide a means of communication between comfort function and locking system.

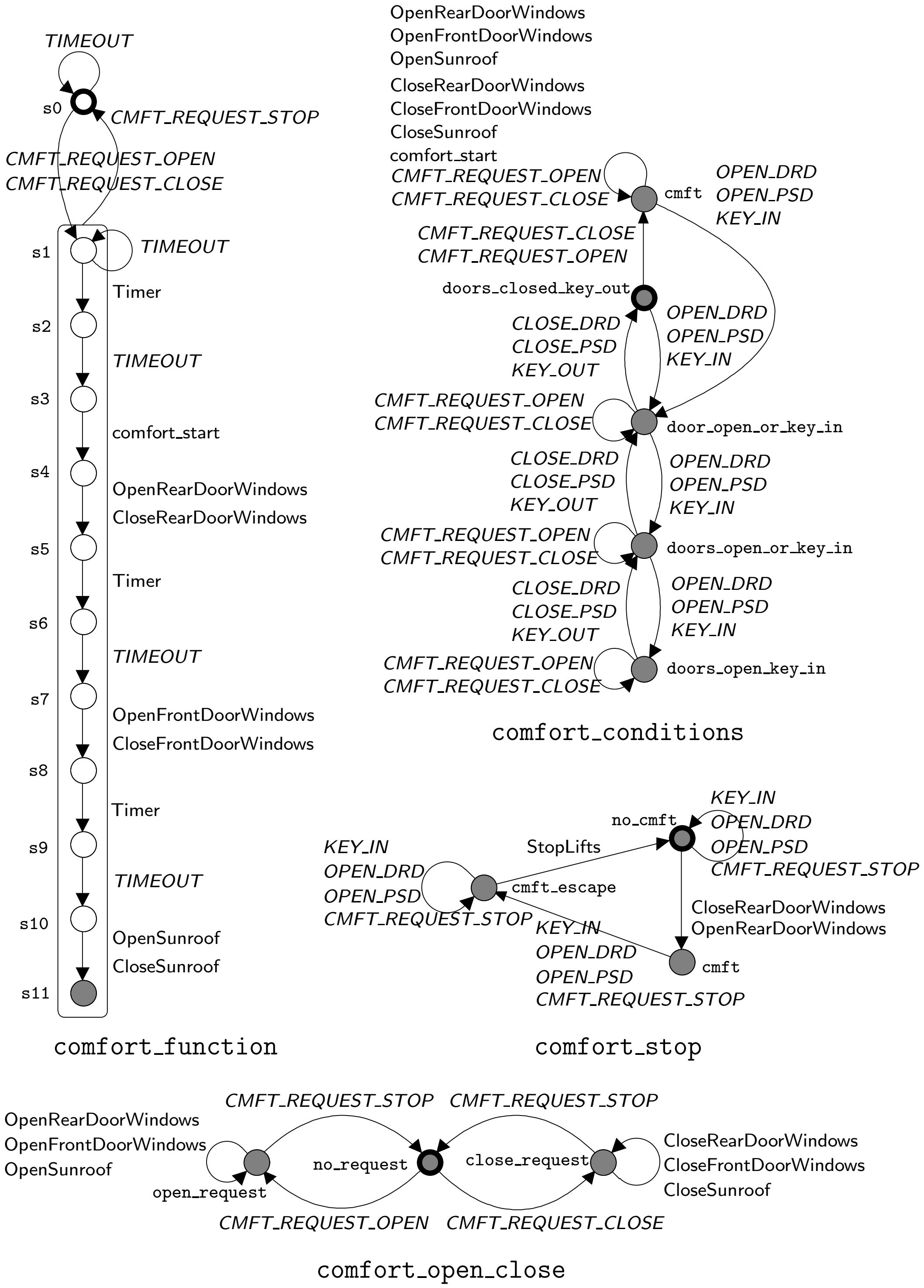

Figure 6.17: Car example—specifications of the comfort function.

or the key is in the ignition switch, a running comfort function is aborted. The automaton `comfort_conditions` specifies these requirements. Only from state `doors_closed_key_out`, where the front doors are closed and there is no key in the ignition switch, state `cmft` can be reached, where the comfort function can be performed, i.e. the events for opening or closing the windows and the sunroof are eligible. These states are left as soon as a door is opened or the key is inserted in the ignition switch.

Automaton `comfort_stop` is responsible for the correct termination of a comfort function. When a comfort function is terminated, either by opening a front door (event *OPEN_DRD* or *OPEN_PSD*), inserting a key into the ignition switch (event *KEY_IN*), or by releasing the button requesting a comfort function (event *CMFT_REQUEST_STOP*), all lifts are stopped (event StopLifts). This is only necessary if a lift has been started.

6.2.4 The Locking System

The locking system is responsible for locking and unlocking the windows and the sunroof as described informally in section 6.2.1. An automata representation derived from this informal description is given in figure 6.18. The controllable events Lock and Unlock model the locking and unlocking of all windows and the sunroof. For the sake of convenience and clarity, we use several controllable events in order to model the locking: Lock.CarOff, Lock.Starter, and Lock.Usaus. The reason why the windows and the sunroof are locked is included in the name after the dot. Note that we do not model the buttons for opening and closing the windows and the sunroof manually. Thus, the result of a locking operation initiated by event Lock is not modelled.

Automaton `locking_system`[3] models locking and unlocking depending on conditions (C1) and (C3) as described in section 6.2.1. The windows and the sunroof can only be unlocked (event Unlock) at state `unlock`, which is reached when the ignition key has been turned to the 'on' position (event *TURN_ON*). When a comfort function is started (event comfort_start), we reach state `lock` in which unlocking (event Unlock) is disabled and locking (event Unlock.CarOff) is enabled.

The windows and the sunroof are also locked when condition (C2) is satisfied, i.e. when the starter is activated by turning the ignition key to the 'starter' position. This is modelled by automaton `locking_system_starter`. After turning the ignition key to the 'starter' position (event *STARTER_ON*), we reach state `lock` in which locking (event Lock.Starter) is enabled and unlocking (event Unlock) is disabled. After deactivating the starter (event *STARTER_OFF*) and reaching state `unlock`, the windows and the sunroof can be unlocked again (event Unlock).

[3] The internal event condition_europe is described later in this section.

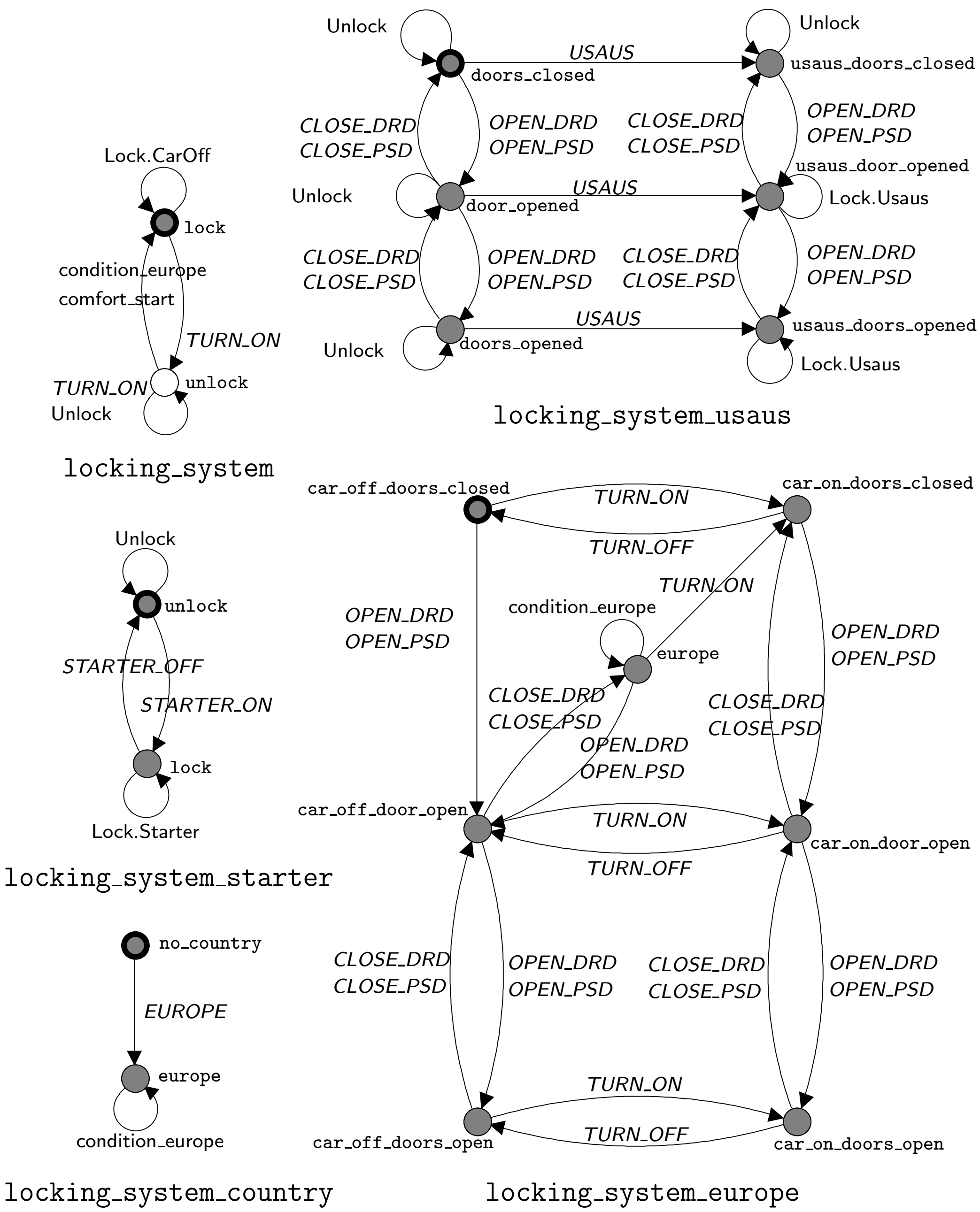

Figure 6.18: Car example—specifications of the locking system.

If USA or Australia is configured, the windows and the sunroof must be locked as long as a front door is open (C5). The automaton `locking_system_usaus` ensures that the controllable event Unlock is only enabled when either USA or Australia has not been selected or both doors are closed. Otherwise, if USA or Australia has been selected and one of the front doors is open, the event Lock.Usaus is enabled.

If a European country is selected, the windows and the sunroof are locked when the last front door has been closed while the ignition key is in the 'off' position or not inserted, as described by condition (C4). In order to model this specification, the internal controllable event condition_europe is introduced, which is not associated to an actuator but describes an internal state change of the controller that is modelled. In automaton `locking_system_europe`, the internal event condition_europe is only enabled at state `europe` when condition (C4) is satisfied. In automaton `locking_system_country`, event condition_europe is only eligible if the uncontrollable event *EUROPE* has occurred once, which indicates that a European country has been configured. In automaton `locking_system`, the occurrence of the event condition_europe causes a state change from state `unlock` to state `lock`, in which the windows and the sunroof are supposed to be locked.

6.2.5 Checking, Completing, and Correcting the Model

The model consisting of all automata introduced so far, i.e. the synchronous product of the automata in figures 6.16, 6.17, and 6.18, describes the desired system behaviour. The specifications in figures 6.17 and 6.18 can be used as a least restrictive supervisor. We are now interested in the generation of a controller that produces controllable events as output and receives uncontrollable events as input. To ensure that the controller is nonblocking, we check the properties introduced in this thesis. Note that the set Ξ of output events is equivalent to the set of controllable events.

Checking the model for Ξ-termination finds several Ξ-loops in the system. Transitions labelled with the output events Lock and Unlock form a loop starting from most states of the model. In particular, a self-loop labelled with the output event Lock.CarOff is defined at the initial state, which is also the shortest counterexample. When deriving a controller from this model, the locking will be initiated again and again, even if the windows and the sunroof are already locked. Beside the fact that this is not the desired behaviour of our controller, a controller which prefers these events may block other output events. In the case that the windows and the sunroof are locked and a comfort function should be started, the controller may choose either Unlock or Timer to occur next. The controller which always prefers Unlock blocks the comfort function.

By adding the automaton `locking_system_lock_unlock` in figure 6.19, we are able to prevent this faulty behaviour. Here, locking and unlocking are enabled alternatingly. When locked, the event Lock must not occur; when unlocked, the

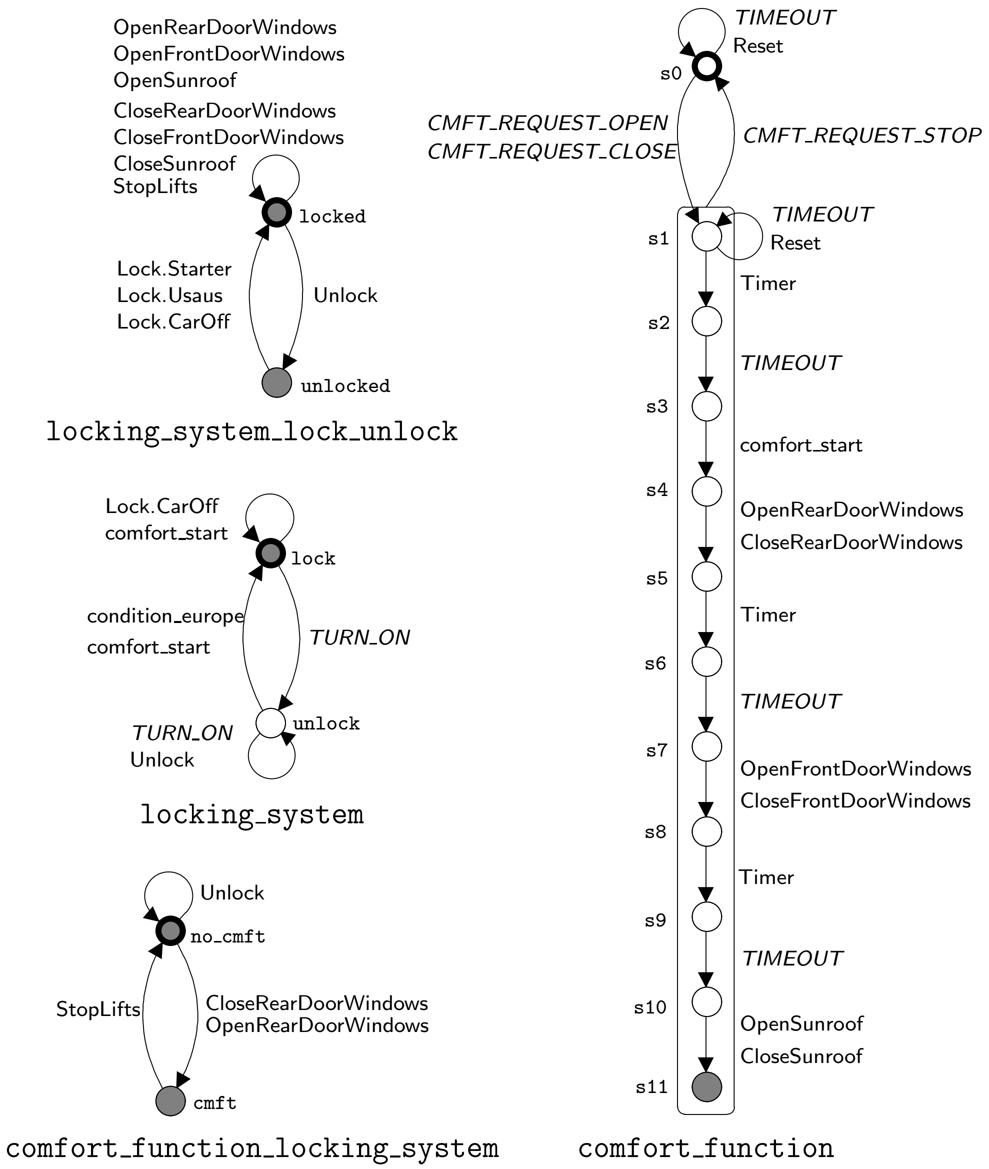

Figure 6.19: Car example—corrections.

event Unlock must not occur. Additionally, we add the forgotten requirement that a comfort function may only be performed when all windows and the sunroof are locked, i.e. in state `locked` of automaton `locking_system_lock_unlock`.

	`comfort_function`	`locking_system`
(initial states)	s0	s0
ReadCountry	s0	s0
EUROPE	s0	s0
KEY_IN	s0	s0
TURN_ON	s0	s1
TURN_OFF	s0	s1
KEY_OUT	s0	s1
CMFT_REQUEST_CLOSE	s1	s1
Timer	s2	s1
TIMEOUT	s3	s1
Non-confluent events:		
comfort_start	s4	s0
condition_europe	s3	s0

Figure 6.20: First Ξ-confluence counter-example.

After this change, the new system is Ξ-loop free, but not Ξ-confluent. A counter-example is given in figure 6.20. The internal events condition_europe and comfort_start lead, starting from the same state, to different states and cannot be joined by means of output events only. Consider automaton `comfort_function` in figure 6.17. The event comfort_start is eligible in state `s3` and causes a state change, whereas event condition_europe is not restricted by this automaton. Thus, if condition_europe is executed, the automaton remains in its actual state. Now consider automaton `locking_system` in figure 6.18. Both events are eligible in state `s1` and cause a state change when executed. If one of the events is executed, the other cannot be executed, since both events are disabled at state `s0`. If event condition_europe is executed, the comfort function cannot be performed. But when the windows and the sunroof are locked, i.e. at state `s0` in automaton `locking_system`, a comfort function should be able to start. We forgot to enable comfort_start in state `s0`. The corrected automaton is given in figure 6.19.

The new model is still not Ξ-confluent. A counter-example is given in figure 6.21. Assume that the key is inserted into the ignition switch and turned to the 'on' position while the comfort function is running. A state can be reached in which the controller can choose between unlocking the windows and the sunroof by sending an Unlock message or stopping the lifts by sending a StopLifts message. These events lead to different states, and it is not possible to reach a common target state by means of output events only. If event Unlock is chosen, the event StopLifts is disabled in the automaton `locking_system_lock_unlock`

	locking_system_lock_unlock
(initial states)	locked
CMFT_REQUEST_CLOSE	locked
Timer	locked
TIMEOUT	locked
comfort_start	locked
KEY_IN	locked
TURN_ON	locked
Non-confluent events:	
StopLifts	locked
Unlock	unlocked

Figure 6.21: Second Ξ-confluence counter-example.

in figure 6.19. Thus, the windows and the sunroof can be unlocked while the comfort function is still running. A possible solution to this problem is to disable event Unlock until the lifts are stopped. This can be done, for instance, by adding the automaton `comfort_function_locking_system` in figure 6.19 to our model.

The resulting model is Ξ-terminating and Ξ-confluent, but Ξ-blocking. Starting from the initial state, we are not able to reach a marked state by means of Ξ-controlled paths only. A marked state could only be reached by performing the comfort function, i.e. opening or closing all windows and the sunroof. But the problem is that a controller may reset the timer as soon as the timer has been started, so the event *TIMEOUT* cannot occur in a Ξ-controlled path and the marked state cannot be reached by means of Ξ-controlled paths. Resetting the timer just after starting it, is not the desired behaviour of our controller. For such a controller it is very unlikely that the comfort function will be performed as desired. In figure 6.19, a corrected version of the automaton `comfort_function` (figure 6.17) is given. In the original automaton, the event Reset does not occur and thus is not restricted by this automaton. In the corrected automaton, resetting the timer is only possible when no comfort function is requested or the comfort function has just been requested and the timer has to be started. We may reach these states with the timer running when a comfort function has been aborted. In this case, we do not want to wait for a timeout and resetting the timer is correct behaviour.

The resulting model is Ξ-terminating, Ξ-confluent, and Ξ-nonblocking. A controller can be derived easily from this model by implementing any live Ξ-supervisor. Based on the analysis results, the system controlled by such a controller is certainly nonblocking, i.e. the task specified by the marking conditions can always be completed.

Chapter 7

Conclusions

In this thesis, we have presented a new approach for the design of discrete controllers, which receive their information from sensors and send commands to actuators, based on supervisory control theory. By introducing Ξ-terminating, Ξ-confluent, and Ξ-nonblocking discrete event systems, we have characterised a class of models that is nonblocking for each controller realisation. A controller derived from such a model cannot prevent the completion of the task by choosing "wrong" commands. Algorithms for checking these properties efficiently have been presented. The proposed design process has been applied to a large industrial example, demonstrating the usability of the approach.

The results of this thesis enable us to derive a nonblocking controller from a model that satisfies the introduced properties, assuming synchronous communication of plant and controller. The results can also be used for systems in which plant and controller are connected via a network by modelling the communication explicitly. Subject to future work is the development of techniques allowing the extraction of controllers that are correct for asynchronous communication without modelling the channel explicitly.

In future work, advantages and limitations of the approach should be studied further by considering more industrial applications. Since discrete event systems are continuously increasing in complexity, more sophisticated algorithms will be required to be able to verify even larger systems in reasonable time. Heuristics for the incremental algorithms should be developed and compared. In particular, the incremental Ξ-nonblocking algorithm has the potential to be extended further by defining more reductions. All algorithms can be improved by adding symbolic and partial order reduction techniques.

In order to increase acceptance in industry, tools have to be developed that guide the modelling process. By pressing a button, the introduced properties could be checked using the incremental algorithms presented in this thesis. If a property is not satisfied, a counter-example can be provided. This leads the designer to a problem in the model. It would be useful to automatically compute several ways of fixing the model. Then the designer can choose the desired solution. Another possibility is to synthesise a system that satisfies the desired properties. Unfortunately, there does not exist a unique solution to this problem. Further research is required in order to identify an appropriate system.

The power of supervisory control, its simplicity and elegance, suggests that it is a convenient basis for tools for the formal modelling and analysis of control software. The approach proposed in this thesis increases the applicability of supervisory theory and is therefore an important step in this direction.

Bibliography

[AH98] J. Ashley and L. E. Holloway. Condition languages and condition systems for modeling ambiguous control specifications. In *Proceedings of 4th International Workshop on Discrete Event Systems*, 1998.

[Ave95] Jürgen Avenhaus. *Reduktionssysteme*. Springer-Verlag, 1995.

[AVW02] A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controllers with partial observation. In F. Cassez, C. Jard, F. Laroussinie, and M. D. Ryan, editors, *MOVEP'2002*, pages 138–172, 2002.

[Bal94] Silvano Balemi. Input/output discrete event processes and communication delays. *Discrete Event Dynamic Systems: Theory and Applications*, 4:41–85, 1994.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BHG+93] S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. F. Franklin. Supervisory control of a rapid thermal multiprocessor. *IEEE Transactions on Automatic Control*, 38(7):1040–1059, July 1993.

[BL98a] G. Barrett and S. Lafortune. A novel framework for decentralized supervisory control with communication. In *Proceedings of IEEE International Conference on Systems, Man, and Cybernetics*, 1998.

[BL98b] G. Barrett and S. Lafortune. On the synthesis of communicating controllers with decentralized information structures for discrete-event systems. In *Proceedings of 37th IEEE Conference on Decision and Control*, 1998.

[BMD00] B. A. Brandin, R. Malik, and P. Dietrich. Incremental system verification and synthesis of minimally restrictive behaviours. In *American Control Conference*, 2000.

[BW92] B. A. Brandin and W. M. Wonham. Supervisory control of timed discrete-event systems. In *Proceedings of 31th IEEE Conference on Decision and Control*, pages 3357–3362, Tucson, AZ, USA, December 1992.

[BW94] B. A. Brandin and W. M. Wonham. Supervisory control of timed discrete-event systems. *IEEE Transactions on Automatic Control*, 39(2):329–342, February 1994.

[CAD99] François Charbonnier, Hassane Alla, and René David. The supervised control of discrete-event dynamic systems. *IEEE Transactions on Control Systems Technology*, 7(2), March 1999.

[CDFV88] Randy Cieslak, C. Desclaux, Ayman S. Fawaz, and Pravin Varaiya. Supervisory control of discrete-event processes with partial observations. *IEEE Transactions on Automatic Control*, 33(3):249–260, March 1988.

[CL99] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, September 1999.

[CM89] H. Cho and S. J. Marcus. On supremal languages of classes of sublanguages that arise in supervisor synthesis problems with partial observation. *Mathmatics of Control, Signals and Systems*, 2(2):47–69, 1989.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. Elsevier Science Publisher (North-Holland), 1990.

[DMWB02] P. Dietrich, R. Malik, W. M. Wonham, and B. A. Brandin. Implementation considerations in supervisory control. In B. Caillaud, P. Darondeau, L. Lavagno, and X. Xie, editors, *Synthesis and Control of Discrete Event Systems*, pages 185–201. Kluwer Academic Publishers, 2002.

[FH98] Martin Fabian and Anders Hellgren. PLC-based implementation of supervisory control for discrete event systems. In *Proceedings of 37th IEEE Conference on Decision and Control*, 1998.

[God91] P. Godefroid. Using partial orders to improve automatic verification methods. In E. M. Clarke and R. P. Kurshan, editors, *Proceedings of 2nd International Conference on Computer Aided Verification, CAV '90*, pages 176–185, Berlin, Heidelberg, 1991. Springer.

[GR87] C. H. Golaszewski and P. J. Ramadge. Control of discrete event processes with forced events. In *Proceedings of 26th IEEE Conference on Decision and Control*, pages 247–251, Los Angeles, CA, USA, December 1987.

[GW00] P. Gohari and W. M. Wonham. Reduced supervisors for timed discrete-event systems. In R. Boel and G. Stremersch, editors, *Discrete Event Systems: Analysis and Control*, SECS 569, pages 119–130. Kluwer Academic Publishers, 2000.

[HCOG00] L. E. Holloway, Andy Callahan, John O'Rear, and Xiaoyi Guan. Spectool: Automated synthesis of control code for discrete event controllers. In R. Boel and G. Stremersch, editors, *Discrete Event Systems: Analysis and Control*, SECS 569, pages 473–476. Kluwer Academic Publishers, 2000.

[HESV93] Ann Hsu, Farokh Eskafi, Sonia Sachs, and Pravin Varaiya. Protocol design for an automated highway system. *Discrete Event Dynamic Systems: Theory and Applications*, 2:183–206, 1993.

[HK94] Hans-Michael Hanisch and Stefan Kowalewski. Algebraic synthesis and verification of discrete supervisory controllers for forbidden path specifications. In *Proceedings of the 4th International Conference on Computer Integrated Manufacturing and Automation Technology*, pages 157–162. IEEE Computer Society Press, October 1994.

[Hop71] J. E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[HWT92] G. Hoffmann and H. Wong-Toi. Symbolic synthesis of supervisory controllers. In *Proceedings of American Control Conference*, June 1992.

[KCM00] R. Kumar, V. Chandra, and S. Mohanty. Automated control synthesis for an assembly line using discrete event system control theory, 2000.

[LBW00] R. J. Leduc, B. A. Brandin, and W. M. Wonham. Hierarchical interface-based non-blocking verification. In *Proceedings of Canadian Conference on Electrical and Computer Engineering*, pages 1–6, May 2000.

[Led96] R. J. Leduc. PLC implementation of a DES supervisor for a manufacturing testbed: An implementation perspective. Master's thesis, Department of Electrical Engineering, University of Toronto, Ontario, Canada, 1996.

[Led02] Ryan James Leduc. *Hierarchical Interface-based Supervisory Control*. PhD thesis, University of Toronto, Ontario, Canada, 2002.

[LW88a] F. Lin and W. M. Wonham. Decentralized control and coordination of discrete event systems. In *Proceedings of 27th IEEE Conference on Decision and Control*, pages 1125–1130, Austin, TX, USA, December 1988.

[LW88b] F. Lin and W. M. Wonham. On observability of discrete-event systems. *Information Sciences*, 44(3):173–198, 1988.

[LW94] F. Lin and W. M. Wonham. Supervisory control of timed discrete event systems under partial observation. *IEEE Transactions on Automatic Control*, 40(3):558–562, 1994.

[LW95] R. J. Leduc and W. M. Wonham. PLC implementation of a DES supervisor for a manufacturing testbed. In *Proceedings of 33rd Allerton Conference on Communication, Control and Computing*, pages 519–528, October 1995.

[Mal02] Petra Malik. Generating controllers from discrete-event models. In F. Cassez, C. Jard, F. Laroussinie, and M. D. Ryan, editors, *MOVEP'2002*, pages 337–342, 2002.

[MCK99] S. R. Mohanty, V. Chandra, and R. Kumar. A computer implementable algorithm for the synthesis of an optimal controller for acyclic discrete event processes. In *Proceedings of 1999 IEEE International Conference on Robotics and Automation*, May 1999.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[MM02] R. Malik and R. Mühlfeld. A case study in verification of UML statecharts: the PROFIsafe protocol. In *Proceedings of the 5th Workshop on Tools for System Design and Verification, FM-TOOLS 2002*, pages 89–93, 2002.

[NSS94] Esko Nuutila and Eljas Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1):9–14, January 1994.

[Per90] Dominique Perrin. Finite automata. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. Elsevier Science Publisher (North-Holland), 1990.

[PKK97] M. H. Prosser, M. Kam, and H. G. Kwatny. Decision fusion and supervisor synthesis in decentralized discrete-event systems. In *Proceedings of American Control Conference*, pages 2251–2255, 1997.

[Ram87] P. J. Ramadge. Some tractable supervisory control problems for discrete event systems. In *International Symposium on MTNS*, Phoenix, AZ, USA, June 1987.

[RSO94] K. Rudie, N. Shimkin, and S. D. O'Young. Timed discrete-event systems: A manufacturing application. In *Proceedings of 28th Conference on Information Sciences and Systems*, pages 374–381, Princeton, NJ, USA, March 1994.

[RW82] P. J. Ramadge and W. M. Wonham. *Supervisory Control of Discrete Event Processes*, volume 39 of *LNCIS*, pages 202–214. Springer-Verlag, Berlin, Germany, 1982.

[RW87a] P. J. Ramadge and W. M. Wonham. Modular feedback logic for discrete event systems. *SIAM Journal of Control and Optimization*, 25(5):1202–1218, September 1987.

[RW87b] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, January 1987.

[RW89] Peter J. G. Ramadge and W. Murray Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[RW92a] K. Rudie and W. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions on Automatic Control*, 37(11):1692–1708, 1992.

[RW92b] K. Rudie and W. M. Wonham. Protocol verification using discrete-event systems. In *Proceedings of 31th IEEE Conference on Decision and Control*, pages 3770–3777, Tucson, AZ, USA, December 1992.

[Tar72] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

[Val91] A. Valmari. A stubborn attack on state explosion. In E. M. Clarke and R. P. Kurshan, editors, *Proceedings of 2nd International Conference on Computer Aided Verification, CAV '90*, pages 156–165, Berlin, Heidelberg, 1991. Springer.

[Var93] Pravin Varaiya. Smart cars on smart roads: Problems of control. *IEEE Transactions on Automatic Control*, 38(2):195–207, February 1993.

[Won99] W. M. Wonham. Notes on control of discrete event systems, 1999. Systems Control Group, Department of Electrical Engineering, University of Toronto, Ontario, Canada; at `http://www.control.utoronto.ca/` under "Research".

[WR87] W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal of Control and Optimization*, 25(3):637–659, May 1987.

[WR88] W. M. Wonham and P. J. Ramadge. Modular supervisor control of discete event systems. *Mathmatics of Control, Signals and Systems*, 1(1):13–30, January 1988.

[WW96] K. C. Wong and W. M. Wonham. Hierarchical control of timed discrete-event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 6(3):275–306, 1996.

[WW98] K. C. Wong and W. M. Wonham. Modular control and coordination of discrete-event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 8(3):247–297, October 1998.

[YL00] Tae-Sic Yoo and Stéphane Lafortune. A general architecture for decentralized supervisory control of discrete-event systems. In R. Boel and G. Stremersch, editors, *Discrete Event Systems: Analysis and Control*, SECS 569, pages 111–118. Kluwer Academic Publishers, 2000.

[Zil02] R. Ziller. Finding bad states during symbolic supervisor synthesis. In *GI/ITG/GMM Workshop: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, Tübingen, Germany, 2002.

[ZW02] Z. H. Zhang and W. M. Wonham. STCT: An efficient algorithm for supervisory control design. In B. Caillaud, P. Darondeau, L. Lavagno, and X. Xie, editors, *Synthesis and Control of Discrete Event Systems*, pages 77–100. Kluwer Academic Publishers, 2002.

Curriculum Vitae of Petra Malik

Name:	Petra Malik
Born:	December 9, 1972 in Merseburg, Germany
Marital Status:	Married
Name of Birth:	Dietrich
Nationality:	German

Education

1979–1991	School attendance in Bad Dürrenberg and Merseburg, Germany.
June 1991	Abitur (secondary school leaving certificate), EOS "Ernst Haeckel", Merseburg, Germany.
1991–1999	Student of Computer Science, University of Kaiserslautern, Germany.
July 1999	Diplom-Informatikerin (Master in computer science), University of Kaiserslautern, Germany.
since 1999	Ph.D. Student at Siemens Corporate Technology, Munich, Germany, and University of Kaiserslautern, Germany.

Employment and Research Stays

Oct.–Nov. 1998	Working student at the Dept. of Software and Engineering, Siemens Corporate Technology, Munich, Germany.
Jan.–Apr. 2001	Visiting scientist at the Systems Control Group, University of Toronto, Canada.
1999–2002	Scholarship from Siemens Corporate Technology, Germany.