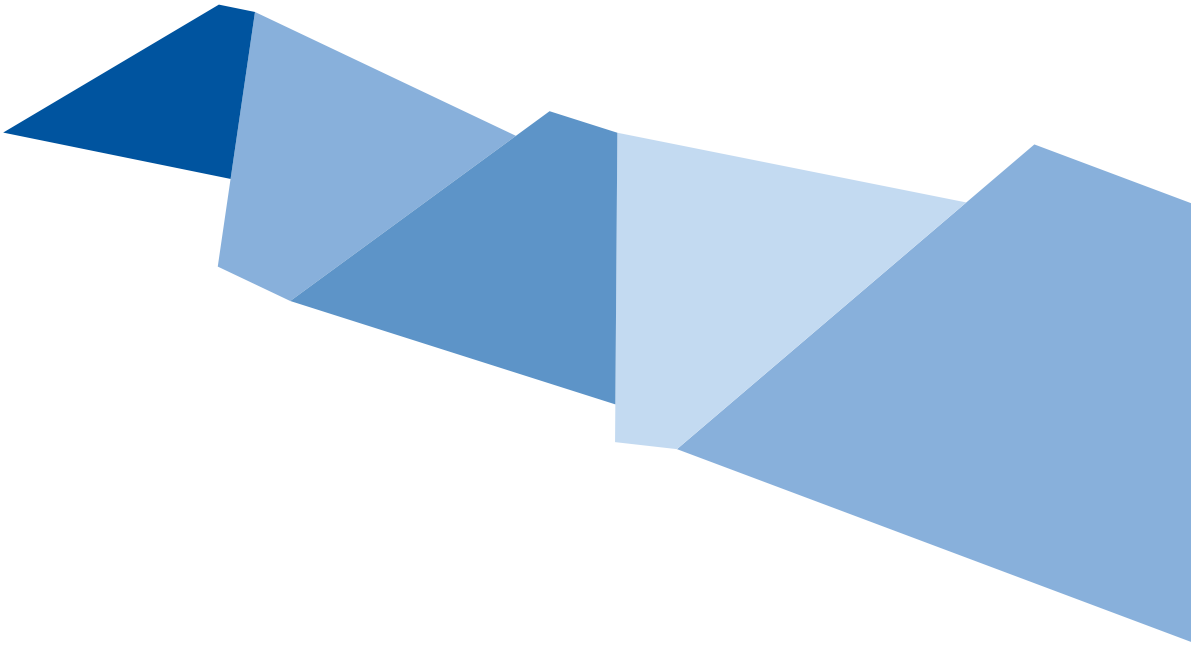


Runtime Supervision of PLC Programs Using Discrete-Event Systems

Florian Göbe

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2019-05
RWTH Aachen University | Department of Computer Science | November 2019

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Runtime Supervision of PLC Programs Using Discrete-Event Systems

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Ingenieurwissenschaften genehmigte Dissertation

vorgelegt von

Florian Göbe
(Master of Science RWTH Aachen University)
aus Duisburg

Berichter: Universitätsprofessor Dr.-Ing. Stefan Kowalewski
Universitätsprofessor Dr.-Ing. Jan Lunze

Tag der mündlichen Prüfung: 7. November 2019

Florian Göbe
Informatik 11 – Embedded Software
goebe@embedded.rwth-aachen.de

Aachener Informatik Bericht AIB-2019-05

Herausgeber: Fachgruppe Informatik
RWTH Aachen University
Ahornstr. 55
52074 Aachen
GERMANY

ISSN 0935-3232

Abstract

The supervisory control theory (SCT) introduced by Ramadge and Wonham is one of the most noted formalisms for the synthesis of solutions in discrete event control. In this dissertation, an approach is elaborated which applies the SCT framework for the supervision of arbitrary existing PLC controller programs. The latter are assumed to be provided manually by the user and hence are not formally guaranteed to respect certain constraints, such as demands on safety, in all possible situations. With the presented approach, these conditions can be formulated in form of discrete-event systems. Using adaptations of the Ramadge and Wonham framework, a supervision layer is generated from these models. It prevents the controller from executing critical actions during runtime, which could eventually lead to the violation of the specified requirements.

In order to address a wide range of realistic use cases, several adaptations and extensions have been introduced to the original framework, such as conditional transitions, templates and event enforcement for the preemption of undesired incidents. A concept for an end-to-end solution from the creation of the requirement models to a ready-to-use safety layer is presented and has been implemented in a tool.

The suitability of the concept has been evaluated in several case studies, some on industrial hardware. Furthermore, the usability of the approach as a whole, the introduced modifications and the tool implementation have been evaluated in a user study.

Zusammenfassung

Bei der von Ramadge und Wonham eingeführten Supervisory Control Theory (SCT) handelt es sich um einen der verbreitesten Ansätze für die Synthese von Lösungen im Bereich diskreter Steuerungen. In dieser Dissertation wurde ein Ansatz erarbeitet, der das SCT-Rahmenwerk zur Überwachung existierender SPS-Steuerungsprogramme einsetzt. Letztere werden als vom Benutzer manuell implementiert angenommen, weswegen ihre Fehlerfreiheit in Bezug auf gewisse Anforderungen, beispielsweise an die funktionale Sicherheit, nicht in allen prinzipiell möglichen Situationen formal sichergestellt werden kann. Im vorgestellten Ansatz können diese Anforderungen als ereignisdiskrete Systeme formalisiert werden. Mittels einer modifizierten Form des Ramadge-Wonham-Rahmenwerkes wird aus diesen Modellen schließlich eine Überwachungsschicht generiert. Diese verhindert frühzeitig solche Aktionen der Steuerung, die schlussendlich zu einer Verletzung der Anforderungen führen könnten.

Um eine möglichst große Bandbreite realistischer Anwendungsfälle abzudecken, wurden mehrere Anpassungen am ursprünglichen Rahmenwerk vorgenommen und Erweiterungen eingeführt, beispielsweise bedingte Transitionen, Vorlagen sowie Ereigniserzwingung zur Verhinderung unerwünschter Vorkommnisse. Das vorgestellte Konzept bildet den gesamten Prozess von der Erstellung der Anforderungsmodelle bis hin zu einer auf der Zielhardware direkt einsetzbaren Sicherheitsschicht ab und wurde in Form eines Werkzeugs prototypisch realisiert.

Die Eignung des Ansatzes zu diesem Zwecke wurde in mehreren Fallstudien demonstriert, in einigen davon auf Industriehardware. Des Weiteren wurde die Anwendbarkeit des Gesamtansatzes, der eingeführten Modifikationen und schlussendlich des Werkzeugs selbst in einer Benutzerstudie evaluiert.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Contribution and Outline . . . 2

2 Background **5**
- 2.1 Discrete-Event Systems . . . 5
- 2.2 Supervisory Control Theory . . . 8
- 2.3 Industrial Automation and PLCs . . . 11

3 Applied SCT and Related Approaches **21**
- 3.1 Extensions of SCT . . . 21
- 3.2 Applicatons – Tools and Case Studies . . . 27

4 Controller Synthesis with SCT **33**
- 4.1 Synthesis – Definition and Classification . . . 33
- 4.2 Continuous Control, Discrete Control and SCT . . . 34
- 4.3 Requirements and Specifications . . . 34
- 4.4 The Role of Specifications in the SCT . . . 39
- 4.5 Operational Specifications for Synthesis Techniques . . . 43
- 4.6 Conclusion . . . 45

5 A Concept for Runtime Supervision of PLC Programs Using DES **49**
- 5.1 Introduction . . . 50
- 5.2 Related Approaches . . . 56
- 5.3 Modeling Concept . . . 57
- 5.4 Basic Operations . . . 62
- 5.5 Preemption . . . 64
- 5.6 Cyclic Events . . . 74
- 5.7 Conditional Transitions and Prohibitions . . . 76
- 5.8 Formal Model . . . 82

6 SynTACS **119**
- 6.1 Working with SynTACS . . . 119
- 6.2 Software Architecture . . . 126
- 6.3 SynTACS Runtime Framework . . . 127
- 6.4 Limitations . . . 133

6.5 Remote Supervisor 135

7 Evaluation **139**
7.1 User Study 139
7.2 Case Study: M3P.AC 144
7.3 Further Case Studies 147
7.4 Long-Term Usability 149
7.5 Benchmark of Incremental Synthesis 149

8 Conclusion **153**

List of Figures

2.1 Example automaton 7
2.2 The supervisory control closed loop 9

3.1 Timed DES to capture timing dependencies 26

4.1 Illustration of productivity requirements 36
4.2 Categories of requirements and specifications in discrete control 37
4.3 Specification *Empty-full-empty-cycle* 42
4.4 Operative controller implementation for dosing tank 43

5.1 Comparison of control loop paradigms 52
5.2 Execution cycle 53
5.3 Exemplary specification 58
5.4 Simple Plant 59
5.5 SynTACS event scheme 61
5.6 Composition of a simple specification and plant model 63
5.7 Preemption – motivating example 65
5.8 Example with preemption 67
5.9 Preemption of further events 71
5.10 Cascade of enforced events 72
5.11 Intended cascade of enforced events 73
5.12 Uncontrollable cascade, interrupted by controllable cyclic event 75
5.13 Example of a heating tank. All alphabets are disjoint. 76
5.14 Additional plant model to define when gelatinization is possible 78
5.15 Condition refinement in plant composition 79
5.16 Specification-plant composition with conditions 81
5.17 Specification and regular plant approximation 90
5.18 Example for incremental synthesis 105
5.19 Preemptively but not conventionally realizable synthesis automaton 112
5.20 Auxiliary plant G_{aux} 112

6.1 Screenshot of SynTACS 120
6.2 Example with preemption and timers 123
6.3 Template example 125
6.4 Screenshot: Instantiation of template 126
6.5 Architecture of SynTACS 128

6.6 SynTACS event scheme 129
6.7 SynTACS Runtime Framework 132
6.8 Problematic enforcement-by-Design 134
6.9 SyRF for remote supervisor 136

7.1 User Study: Foreknowledge 141
7.2 Usability rating for SynTACS 143
7.3 Degree of difficulty in modeling DES 143
7.4 M3P.AC 144
7.5 Evaluation: Example of specification and plant 146
7.6 fischertechnik PPC 147
7.7 Transfer line 150
7.8 Several connected TLs with multiple machines each 150

Acronyms and Initialisms

ALARP	as low as reasonably practical
BDD	binary decision diagram
BSCP	basic supervisory control problem
BSCP-NB	basic supervisory control problem with nonblocking-ness
CFC	Continuous Function Chart
CS	computer science
CTL	computational tree logic
DCS	distributed control system
DEDS	discrete-event dynamic system
DES	discrete-event system
DFA	deterministic finite automaton
FBD	Function Block Diagram
GEF	the Graphical Editing Framework
GUI	graphical user interface
IDE	integrated development environment
IEC	International Electrotechnical Commission
IL	Instruction List
IPCCSL	infimal prefix-closed controllable superlanguage
LD	Ladder Diagram
PCS	process control system
PFC	Procedural Function Chart
PII	Process Image of Inputs
PIO	Process Image of Outputs
PLC	programmable logic controller
POU	Program Organization Unit
PPC	pneumatic processing center

PROFIBUS	process field bus
PROFIBUS DP	process field bus (for) decentralized perihperals
RCP	rich client platform
SAT	satisfiability
SCL	Structured Control Language
SCSL	supremal controllable sublanguage
SCT	supervisory control theory
SFC	Sequential Function Chart
SIL	safety-integrity level
SMT	satisfiability modulo theories
ST	Structured Text
STCT	SmartTCT
STS	state tree structure
SynTACS	Synthesis Tool for Automation Controller Supervision
SyRF	SynTACS Runtime Framework
TCT	Toy Control Theory
TL	transfer line
TON	on-delay timer
UML	Unified Modeling Language

Chapter 1

Introduction

Software is one of the most inherent elements of today's electronic and mechatronic systems. What originally began with the automated processing of business transactions in mainframe systems has long since found its way into all areas of daily life. Nowadays, there is hardly any technical device that does not involve a microprocessor and software to be executed on it. The opportunities are obvious: Software allows using general-purpose devices, which can be produced in large quantities, for very specific tasks. It can easily be copied, transferred, modified and maintained compared to purely hardware-based solutions.

Besides the functional and architectural aspects of a system, nonfunctional requirements have always played a central role in software development. For the controls of machines and industrial systems, safety is probably the most prominent example, which is in many cases even regulated by law. Accordingly, safety must often be enforced by concepts and systems which are independent from functional aspects of the system.

As the complexity of the tasks solved by software increased over time, it became inevitable that the actual implementation is preceded by a specification and design phase. Usually, different kinds of models are invoked to that end, such as the 13 parts of the Unified Modeling Language (UML) but, particularly in the embedded domain, also block diagrams, state machines or MATLAB/Simulink.

In that context, there have always been attempts to automate as much of the process of creating software as possible. That has several reasons. On the one hand, the automatic derivation of software code from models can save time and cost as the developers and engineers can concentrate on the modeling and need not bother with implementation details. Besides, such mechanisms can improve reusability as the same model may serve as a basis to generate code for multiple different hardware platforms. On the other hand, a correct method is guaranteed to yield sound results. Apparently, that only holds if the models which serve as input are correct too. Thus, when using generative methods, the developer has to entirely rely on the preciseness of these models as the chances to find errors in generated artifacts are usually small.

Two different kinds of software derivation need be distinguished: The derivation of code from a model *of the solution* and the creation of a solution from a model *of the problem*. Throughout this dissertation, the first will be referred to as *generation* whereas the second is called *synthesis*.

The supervisory control theory (SCT) introduced and formally defined by Ramadge and Wonham [87] is a calculus for the synthesis of *supervisors* from discrete-event systems. It originates from the field of discrete control engineering and hence aims to provide a method to control a system with a discrete nature. The core idea is that two discrete, state- or language-based models are provided by the user to describe the control problem: A model of the uncontrolled plant that represents all possible behaviors, and a specification of the legal behavior. The main task of the supervisor is first to ensure compliance with the specification and second to prevent the system from blocking. Some of the system's dynamics are assumed to be uncontrollable. Thus, synthesis has to find a solution that does achieve these two goals in a technically realizable way.

1.1 Motivation

Synthesizing controllers with the Ramadge-Wonham framework has been the subject of theoretically oriented research for a notable time. Nonetheless, these methods still struggle with real-world applications for several reasons. One amongst these is the computational complexity that comes with handling regular state spaces due to the exponentially growing number of possible state combinations over several components. The majority of contributions in the area face this issue by introducing abstractions, symbolic methods or heuristics. What has rarely been addressed though is an analysis on which aspects of a control scenario can actually benefit from SCT and in which way. Numerous case studies exist in which a controller for a simple system has successfully been synthesized using that framework. In that context, particularly process and production plants received frequent attention. However, it appears unlikely that, e.g., an entire production facility will be controlled by SCT-synthesized supervisors in foreseeable time.

One of the main objectives of this work, hence, was to identify system aspects and parts that allow for SCT-synthesized solutions in a way that is computationally tractable, well-applicable and, first of all, reasonable from the application's point of view. It turned out that one class of requirements suits the notion of SCT very well. They will be addressed as *side conditions* and basically represent a generalization of safety constraints. A side condition is a requirement that, independently from the functional goals of the system, needs to be respected during the entire operation of the facility. Side conditions are not necessarily invariants but may impose restrictions depending on the current state of the system. More details will be given in Chapter 4.

1.2 Contribution and Outline

This dissertation comprises mainly two contributions. The first is a critical discussion on applying syntheses in general and the SCT in particular for the purpose of controller synthesis. Although several case studies have shown the overall applicability of the method, there are conceptual limitations one should be aware of. The second and main contribution is an approach to model side conditions and synthesize a software framework from these.

It monitors and supervises an existing controller implementation with the objective of guaranteeing compliance with the requirements as the controller is running. The approach is based on an altered version of the SCT, which turned very suitable for this kind of runtime supervision. Based on that concept, a tool implementation has been developed to evaluate its suitability and limitations. The tool is called SynTACS and offers end-to-end support from the initial modeling up to the generation of executable PLC code. Strong emphasis has been put on usability and convenient applicability in order to estimate the appropriateness of SCT to serve that purpose in practice. This particularly includes an intuitive modeling concept which avoids the necessity of redundant manual tasks.

During the development, the formal framework and the approach were adapted and modified towards the addressed problems and not the other way around, i.e., the focus was not to find an example which works well for the SCT but rather to justify and extend SCT to make it suitable for many practically relevant scenarios. Therefore, concepts as explicit prohibitions, alphabetless specifications, different trigger and action classes or preemption are utilized. Furthermore, an incremental method is presented. Since it is tailored to the needs of the considered use case, it is able to apply narrower criteria for the necessity of composing modular automata than other approaches, making it more efficient.

Since the introduced modeling concept introduces extensive modifications to the original framework by Ramadge and Wonham, a formalization is given and the soundness of the implemented methods is proven on that basis. Moreover, the practicality and usability of SynTACS has been evaluated.

Chapter 2 gives an introduction to the technical background of this thesis, i.e., to discrete-event systems, the SCT and, very briefly, to industrial manufacturing and process engineering. It is followed by Chapter 3 which roughly summarizes various existing contributions to the SCT which were the results of several different incentives. It further gives an overview of related work, alternative modeling concepts, tools and case studies. Chapter 4 is concerned with the general discussion about syntheses and SCT mentioned above. Throughout Chapter 5, the concept for the runtime supervision approach is presented including the modifications and additions to SCT and a self-contained formalization of the approach. Based on that concept, the tool SynTACS is introduced in Chapter 6. An evaluation of both the tool and the formal approach itself is given in Chapter 7. Chapter 8 finally draws a conclusion.

Chapter 2

Background

This chapter is meant to give a brief overview about discrete-event systems (DES) and the classic supervisory control theory (SCT). The first two sections, 2.1 and 2.2 are mainly based on the textbook by Cassandras and Lafortune [23], which is widely accepted as a standard reference in the community.

This dissertation primarily addresses the application domain of industrial automation. For that reason, an overview of the control methods and components commonly used in this field is given in Section 2.3.

2.1 Discrete-Event Systems

In the area of systems theory, usually two different types of causal systems are distinguished: static systems for which the output exclusively depends on the current input, and dynamic systems, which possess an internal state such that not only current but also past values influence the system's current output. Dynamic systems are further partitioned into four different classes regarding their time and value domains as both can be either continuous or discrete [23].

DES, sometimes referred to as discrete event systems or – more precisely – discrete-event dynamic systems (DEDS) are characterized by a discrete state space. Their active state can change at arbitrary points in time. Thus, DES are counted to the continuous-time systems although timing is not explicitly considered by most approaches. State changes are called *transitions* and are always triggered by *events*, where an event is usually the logic representation of a certain physical happening. Both events and transitions have no duration. Further, two events are assumed to never occur at the exact same time.

Example 2.1. Consider a bucket that can hold 12 identical marbles. Marbles can be put into the bucket or removed from it one by one in arbitrary order. This bucket can be represented by a DES with two events (*put*, *remove*) and 13 distinguishable states (since the bucket can be empty too). △

Languages The dynamics of continuous systems can usually be described with differential equations, which offer an elegant way to capture the entire possible behavior in a compact representation. Discrete events are not differentiable though. Besides, a functional

description of a discrete state space would be very cumbersome to work with. Instead, a standard DES is characterized by the set of its possible event sequences, called the *language* of the DES. A formal language is a set of finite strings over an alphabet of events Σ. The Kleene closure of Σ, denoted by Σ^* contains all possible finite strings over Σ. Thus, each language L over Σ is a subset of Σ^*.

In general, both the language but also the DES itself can be infinite. However, the vast majority of contributions, such as [28, 30, 75, 87, 88, 119], concentrate on DES with finite state spaces instead. Note that these have regular languages, which can also be infinite in general.

Since a DES is assumed to be causal, its language must be prefix-closed[1]. That is, for every event sequence from the language, all prefixes of this sequence (substrings that start at the beginning but may be shorter) must be contained in the language as well. From the perspective of the DES this makes sense as it can only produce events but never take them back once they are emitted. The prefix closure, i.e., the set of all prefixes, of a language L is usually denoted by $\overline{L}$. Hence, L is prefix-closed if and only if $L = \overline{L}$.

Example 2.2. Consider the prefix-closed language L. $\texttt{abc}, \texttt{def} \in L \implies \texttt{a}, \texttt{ab}, \texttt{d}, \texttt{de} \in L$. $\triangle$

More details and examples on prefix closure can be found in [23].

2.1.1 Automata

Infinite languages are difficult to handle in their natural shape of sets of strings. Particularly, the definition of properties and algorithms, such as analyses and syntheses, can be problematic using languages only. Hence, a more compact, yet sufficiently expressive, representation of DES is required. For DES with regular languages, the most popular concepts are deterministic finite automata (DFAs) and Petri nets [48]. While the former offer a rather generic view on the state space, the latter suit the needs of scenarios where discrete, countable items are moving through a fixed, predefined topologic structure. Both have equivalent expressiveness if the Petri nets are restricted to finite markings.

Throughout this dissertation, automata are used extensively to define DES and the operations on them. Petri nets still represent a niche in the DES community and will not be considered here. The formalisms sketched in this chapter shall give an impression about the common modeling paradigm and further provide a basis to understand the techniques presented in the following sections and in Chapter 3. A detailed and self-contained model for the approach presented in Chapter 5 will be given in Section 5.8.

In their textbook [23] Cassandras and Lafortune define a deterministic finite automaton (DFA) as a sextuple $G = (Q, \Sigma, f, \Gamma, q_0, Q_m)$, consisting of a finite set of states Q, an alphabet Σ, a partial transition function $f : Q \times \Sigma \rightharpoonup Q$, an active-event map $\Gamma : Q \rightharpoonup 2^\Sigma$, an initial state q_0, and a set of marker states Q_m[2]. Σ contains all events that the automaton is concerned with, i.e., the events that play a role for the represented DES. Γ assigns to

[1] Simply called *closed* in Wonham's publications

[2] In the book, the set of states is denoted X, as common for continuous systems, and the alphabet is called E. Q and Σ are widely used in computer science as well as in more recent DES publications. In order to avoid confusion, a consistent notation is used throughout this dissertation.

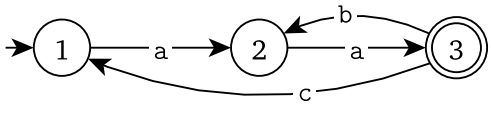


Figure 2.1: Example automaton A

each state $q \in Q$ the set of all events e such that $f(q, e)$ is defined. Last, marker states define those circumstances in which the DES is considered stable or safe. By ensuring the reachability of at least one such state the system is guaranteed not to block, a property which is often addressed as *nonblockingness*.

The majority of contributions inside the DES community have either adopted or slightly altered this definition to their needs. Note that it differs from the common definition of theoretical computer science (CS), where the transition function of a DFA, usually $\delta : Q \times \Sigma \to Q$ (over a state space Q and alphabet Σ) has to be total. This also affects the definition of languages. Since the language $\mathfrak{L}$ of a DES shall reflect all possible behaviors of the system, it is defined as the set of strings over Σ, where f is transitively defined, i.e., which the corresponding automaton G can execute. Hence, it is often called the *language generated* by the automaton, $\mathfrak{L}(G)$. For nonblockingness, additionally the *language marked* by G, $\mathfrak{L}_m(G)$, is considered. It contains all event strings that, executed on G, end in a marker state $q \in Q_m$. Obviously, $\mathfrak{L}_m(G) \subseteq \mathfrak{L}(G)$.

In CS, the marked language is usually considered exclusively, as for a total transition function δ of an automaton A, $\mathfrak{L}(A)$ would always return the alphabet's Kleene closure Σ^*, i.e., the set of all possible strings over Σ. Both concepts can be used to express the exact same problems and properties, e.g. by introducing sink states to the CS automata. Still, this is a vivid example of how differently both communities have developed during the past decades. Unfortunately, in many cases it is not trivial to translate knowledge which has been gained and formally proven in one community to the other, cf. [31].

Example 2.3. Figure 2.1 shows a sample automaton A of three states $Q = \{1, 2, 3\}, Q_m = \{3\}$. For $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, its language is given by the prefix closure of the language L, $\mathfrak{L}(A) = \overline{L}$, where L comprises all words defined by the regular expression $(\mathtt{aa}(\mathtt{ba})^*\mathtt{c})^*$. $\mathfrak{L}_m(A)$ is given by $L\mathtt{aa}$. Thus, for instance, $\mathtt{a}, \mathtt{aab}, \mathtt{aabaca}, \mathtt{aacaa} \in \mathfrak{L}(A)$ and $\mathtt{aacaa} \in \mathfrak{L}_m(A)$ but $\mathtt{aab} \notin \mathfrak{L}_m(A)$. △

2.1.2 Operations on Automata

Cassandras and Lafortune introduce several unary and binary operators on automata that are relevant for this dissertation. The unary operations are accessible, co-accessible, trim and complement. The first reduces an automaton to the part reachable from the initial state while the second rules out the states from which no marked state can be reached. Trim combines both commutative functions. The binary operators are the parallel composition, sometimes called *shuffle product* [88], and the automaton product.

When two automata G_1, G_2 are composed in parallel, denoted by $G_{\text{comp}} = G_1 \parallel G_2$, their alphabets Σ_1, Σ_2 are united to $\Sigma = \Sigma_1 \cup \Sigma_2$. The transition function f of G_{comp} then reflects exactly that behavior which is considered possible by both automata. Note that each of them only reasons about the events contained in its respective alphabet.

The automaton product $G_{\text{prod}} = G_1 \times G_2$ works similar except it intersects the alphabets: $\Sigma = \Sigma_1 \cap \Sigma_2$. Its language $\mathfrak{L}(G_{\text{prod}})$ is also the intersection of the original automata's languages, i.e., only the strings that were generated by both automata reside. The formalization of the parallel composition's language $\mathfrak{L}(G_{\text{comp}})$ is slightly more complex as it requires the notion of natural projections. A formal description is omitted here. It can be found in Section 5.8.1.5 (Lemma 5) and in [23]. Both compositions are commutative and associative, i.e., $G_1 \parallel (G_2 \parallel G_3) = (G_1 \parallel G_2) \parallel G_3 = (G_2 \parallel G_1) \parallel G_3$, analogously for $\times$. The automaton product is only mentioned for the sake of completeness but does not play a role in this dissertation.

2.2 Supervisory Control Theory

The SCT is a framework that intends to adapt the classic concept of a closed control loop containing the uncontrolled plant on the one hand and the controller on the other, as widely applied in continuous control, to discrete-event systems. It has been developed by P. J. Ramadge and W. M. Wonham in the 1980's. Although there have been closely related papers by these authors before, [87] is widely regarded as the first peer-reviewed publication[3] of the coherent framework, although the term supervisory control theory is not explicitly mentioned yet. Earlier contributions, e.g., [118], concentrate on language-related considerations and introduce the concept of the *supremal controllable sublanguage* but do not yet apply them onto a closed-loop setup. In addition, several illustrative examples are given. The invited paper [88] summarizes the earlier papers and examples, including more advanced scenarios, such as modular SCT or *coordinators* (see Section 3.1.2).

2.2.1 Setup

The control loop of the basic framework consists of two components: A *generator* and a *supervisor*.

Generator The generator is a DES which spontaneously creates events. The set of all event sequences that a given generator can produce from its initialization to any arbitrary moment thereafter forms a prefix-closed language. Since the generator is commonly modeled as a DFA or Petri net, that language is usually regular. The active-event set Γ of the automaton determines, depending on the current state, which events can theoretically be emitted next. In the classic setup, all events are assumed to originate from the generator. It depends on the application which physical entities the generator represents in the respective case, cf. Section 5.1.3.

[3] previously published as a technical report [89]

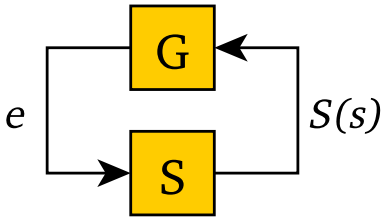


Figure 2.2: The supervisory control closed loop

Supervisor The role of the supervisor is to restrict the behavior of the generator to a certain subset of its language by *disabling* single events. To that end, it receives from the generator all events that have been produced in the order of their emission. In return, the set of *enabled* events, i.e., those that the supervisor allows to occur next, are transferred back to the generator. The resulting control loop is depicted in Figure 2.2.

Since the supervisor is a dynamic system, it can theoretically consider the entire past event string $s \in \Sigma^*$ from the very beginning of the execution up to the last generated event for that decision. Formally, a supervisor is a function $S : \mathfrak{L}(G) \to 2^{\Sigma}$. However, since $\mathfrak{L}(G)$ is typically infinite, a technical realization of S would require infinite memory too. Besides, examples that require a non-regular supervisor for a regular generator seem rather artificial. For these reasons, usually a regular *supervisor realization* in terms of another DFA is used instead. The closed-loop behavior of the supervised generator is in that case given by $\mathfrak{L}(S \parallel G)$.

Controllability The alphabet Σ is partitioned into the disjoint subsets of controllable and uncontrollable events, denoted by Σ_c and Σ_{uc}, such that $\Sigma = \Sigma_c \uplus \Sigma_{uc}$. A supervisor is called *admissible* if it never disables a pending uncontrollable event, i.e., S admissible iff for all $s \in \mathfrak{L}(G)$ holds

$$\Sigma_{uc} \cap \Gamma(f(q_0, s)) \subseteq S(s). \tag{2.1}$$

The claim for admissibility is sometimes referred to as the *control law* of SCT [29].

Specification In addition to the generator, which represents the possible, uncontrolled behavior of the DES, a (safety) specification can be provided to define its *legal* behavior. It regularly has the shape of a DFA as well. Thus, for a specification H, the aggregation of all legal system behaviors is given by the language of that DFA, $\mathfrak{L}(H)$. Note that in case of H having an alphabet $\Sigma_H \neq \Sigma$, only the events inside Σ_H are affected by this particular specification. That means, if H does not provide an outbound transition for a certain $e \in \Sigma \setminus \Sigma_H$ leaving the currently active state, this event may still occur legally. In case of $e \in \Sigma_H$, it has to be prevented though. The entire possible and legal behavior of the DES can be obtained from the parallel composition of H and G and is given by $\mathfrak{L}(H \parallel G)$.

Blocking The Ramadge and Wonham framework provides a basic concept for dead- and livelock avoidance. This is achieved using the marker states Q_m defined by G and H. Additionally to its compliance with the specification, a supervisor must ensure that at least one marker state is reachable at any time, regardless of which events the generator produces. Following that definition, a marker state can be understood as a "stable" or "idle" state to which the system must always be able to return. The marked language consists of all strings that end in such a marker state. In order to assure that, at any time, at least one of these can be reached, the supervised system must be restricted to those strings that can be completed to an element of the marked language, i.e., its prefix closure.

2.2.2 Supervisor Synthesis

In contrast to a supervisor, which has to be admissible, a specification is allowed to exclude uncontrollable events, even if they are possible according to G in that state. In this case, obviously no admissible supervisor S exists such that $\mathfrak{L}(S \parallel G) = \mathfrak{L}(H \parallel G)$ because S would need to disable the uncontrollable undesired event at a certain point. H is then called *uncontrollable* on G. Note that this conflict does not arise if all prohibited uncontrollable events are unreachable in G. Thus, admissibility does not depend on S exclusively but also on Γ as reflected by (2.1).

The main goal of supervisory control theory (SCT) is to synthesize an admissible supervisor S that guarantees

1. Compliance with H, i.e., $\mathfrak{L}(S \parallel G) \subseteq \mathfrak{L}(H \parallel G)$
2. Nonblockingness, i.e., $\mathfrak{L}(S \parallel G) \subseteq \overline{\mathfrak{L}_m(G)}$, where $\overline{L}$ denotes the prefix closure of L.

When achieving these goals, two issues can still arise: First, the specification can clash with the required admissibility of the supervisor if it removes states that the generator can enter on uncontrollable events. Second, there can be situations in which uncontrollable events lead to blocking. Remember that these can occur spontaneously in any state q where they are active, i.e., contained in $\Gamma(q)$.

Controllability Theorem A language is called *controllable* if an admissible supervisor exists for that language. That is the case if no uncontrollable event can occur which would "leave" the language. This is formalized in the *controllability theorem* [118, 87]:

> For languages K and $M = \overline{M}$ over an alphabet $\Sigma = \Sigma_c \uplus \Sigma_{uc}$, K is called *controllable* with respect to M and Σ_{uc} if
>
> $$\overline{K}\Sigma_{uc} \cap M \subseteq \overline{K}. \tag{2.2}$$

In words, (2.2) claims that any prefix of K (the legal language), complemented by any uncontrollable event that can occur with respect to M (the possible language), must still be legal. Consequently, we use $M = \mathfrak{L}(G)$ to express the controllability of a language K on the generator G.

Controllability with respect to $\mathfrak{L}(G)$ is neither automatically given for $\mathfrak{L}(H \parallel G)$ nor for $\overline{\mathfrak{L}_m(G)}$ nor the combination of both, $\overline{\mathfrak{L}_m(H \parallel G)}$, for the reasons described above. Hence, in the context of SCT, there exist two primary approaches to establish controllability for such: The supremal controllable sublanguage (SCSL) and the infimal prefix-closed controllable superlanguage (IPCCSL). The first technique corresponds to: Restricting the generator's behavior further until controllability is achieved. The second addresses relaxing the specification until it can be guaranteed controllably. In the context of this dissertation, relaxations will not be considered. Instead, the specification is assumed to be final and does not allow any tolerance.

BSCP and Maximal Permissiveness The underlying control problem is called *basic supervisory control problem* (BSCP) or *basic supervisory control problem with nonblockingness* (BSCP-NB) if nonblockingness is considered. Note that the empty language $K = \emptyset$[4] does always fulfill (2.2). Thus, there is always a trivial solution to BSCP. This solution would basically disallow the system to do anything and is thus reasonable neither for practical nor theoretical considerations. Nonetheless, it raises the question of how BSCP/BSCP-NB should be solved optimally. The answer is usually maximally permissive or, equivalently, minimally restrictive.

Summarized – The result of BSCP/BSCP-NB is the *maximally permissive supervisor* which restricts the generator as little as possible in order to ensure goals 1. or 1. and 2. from above. The language of this supervisor S^{mp} applied to the generator is the SCSL of $K = \overline{\mathfrak{L}_m(H \parallel G)}$. It is denoted by $K^{\uparrow C}$. Thus, $K^{\uparrow C} = \mathfrak{L}(S^{\text{mp}} || G)$.

Algorithm The standard algorithm to compute S^{mp} for BSCP-NB can be found in [23]. An equivalent algorithm albeit based on the underlying formalism of the Synthesis Tool for Automation Controller Supervision (SynTACS) is listed and explained in Section 5.8.2.

2.3 Industrial Automation and PLCs

Originally, SCT was meant to provide a versatile theoretical framework for manifold discrete systems across all branches and domains. Among these, applications from or related to industrial automation have been of special interest inside the SCT community pretty early. The SCT-based automation of a "thermal multiprocessor" by S. Balemi et al. [10] was one of the pioneering contributions in this area. It will be revisited in Section 3.2.2.

Since the approach of this thesis also targets industrial production, even though from a different perspective, a brief overview of the field, along with the applied measures of control, is given in the following.

[4] note that $\emptyset \neq \{\varepsilon\}$

2.3.1 Discrete Manufacturing and Process Engineering

In factory automation, usually two different branches are distinguished: Discrete manufacturing systems and process systems [70]. Although the technical requirements to the utilized hardware platforms are not that different, two very distinct control cultures have developed rather independently.

Discrete Manufacturing A manufacturing plant is in general a production facility where products are thought of and handled as discrete workpieces. A production line usually consists of several workstations or machines, often connected by a logistic system, e.g., conveyor belts. The workstations perform several, precisely defined tasks on the workpieces as they move through the plant. These processing steps must be triggered, controlled and coordinated among each other as well as with the transport system. The probably most advanced examples for manufacturing lines can be found in the automotive industry where not only the internal procedures must be controlled but also the peripheral logistic is involved to realize just-in-time supply and delivery of all components.

But also a single multi-purpose machine, carrying out several actions on a workpiece, each triggered by a human operator, can be regarded as a manufacturing plant.

Both extremes have in common that the plant is equipped with a number of sensors and actuators which form the interface between the controls and the physics. Real-time requirements often play an essential role in manufacturing and the time constants are comparatively short. Disturbances or temporary failure of a component, however, are often not as costly and severe as in chemical processes as the reactivation times of the facilities are typically shorter.

Process Engineering The term process engineering subsumes several industrial branches that deal with liquid or bulk goods. According to [74], five different sectors are subject of process automation:

- Process industries in the narrower sense (chemistry, pharmaceutics, foods and beverages)
- Primary commodities (coal, oil, etc.)
- Base materials (paper, glass, metals, etc.)
- Power plants
- Environmental facilities (fresh and waste water treatment, garbage incineration, air purification, etc.)

Usually, two types of processes are distinguished: *conti(nuous)* and *batch processes* [70]. In a conti process, the product is transported through the system and processed continuously. The desired treatments are performed by the different components along the plant as the liquid/the good flows by. The main advantage is that these processes often scale very well, which allows high throughputs at reasonable cost. Disadvantages are low flexibility and

often extremely high cost in case of a failure. In batch processes, the product is handled in defined volumes, called batches. The different stages of treatments are applied in terms of subsequent steps. Usually, one batch of the product or of multiple reagents are poured into a reactor in which the actual processing takes place. When the reaction has finished, the product is transported to the next station where it receives the subsequent treatment, according to the recipe. After the first reactor is evacuated, it can be filled with the next batch, i.e., the processing steps can usually be carried out in an interleaved manner.

The two types of processes require different kinds of controls. In conti processes, the methods of continuous control apply widely. The subsequent stages of the recipe are manifested statically in the structure of the plant. Thus, the controls are mainly responsible to *keep the process stable* [70]. This task addresses the fundamental objective of continuous control engineering: keeping a reference signal as close as possible to the desired set point. Depending on the process, this can be very challenging indeed. Nonetheless, the methods of discrete control are hardly applicable on conti processes by nature.

Due to their discretely separable steps, batch processes have much more similarities to discrete manufacturing than conti processes do. This allows the process to be expressed in terms of hybrid or even fully discrete models, such as (hybrid) automata or Petri nets.

The presented approach was evaluated on a batch process plant model simulating a wastewater treatment plant. It could be shown that the presented method is well applicable for this kind of processes.

2.3.2 PLCs and PCSs

The more different components are involved in a plant the more vital is the need not only for an automation concept in general but for a unified controller platform. It shall allow for an easy integration in the plant by the means of the electrical wiring. Additionally, the system must provide a certain degree of robustness for being suitable to operate on the shop floor. One of the most important requirements is that the controller logic must be realizable in a way that enables plant engineers and technical staff to design and implement the controls with their previous knowledge. At least, no advanced programming or electronic skills should be necessary.

PLCs Programmable logic controllers (PLCs) are a class of control systems for the automation of industrial production. In contrast to earlier solutions that often involved customized hardware, PLCs consist of standard components and can be configured and programmed to the needs of the respective application by software. Today, they define the de-facto standard for industrial automation.

Internally, a PLC is a computer system, usually equipped with a medium-sized microcontroller, memory and communication interfaces. Nonetheless, a PLC is characterized by two important concepts which separate them from other embedded systems: First, they consist of robust, reliable hardware in a solid housing which is modularly extendable by further components, such as modules for digital or analog I/O. Sensors and actuators can easily be connected to these I/O terminals without soldering. In particular, layout and production of

customized circuits, as often necessary for other control systems, such as microcontrollers or FPGAs, is not required. Second, a PLC comes with a full software bundle including a uniform execution paradigm, five standardized special programming languages and an integrated development environment (IDE), which, besides the programming, allows to configure (*"to project"*) the hardware graphically. Altogether, this makes PLCs not only extremely versatile in comparison with the earlier hardwired solutions but also easy to install, configure and program, all reasons that led to the commercial success of the concept. Up to the 90's there has been a great variety of PLC vendors following different approaches and concepts.

In order to improve compatibility, the first international standard for PLCs has been developed in 1993 and released by the International Electrotechnical Commission (IEC)[54, 55]. It is not mandatory to be followed but means to provide guidelines towards more consistency. Shortly after that, *PLCopen* has been founded, a non-commercial organization that further promotes and encourages unification and exchangeability in the field. Besides, they released a collection of semi-formal specifications for standard safety function blocks.

PCSs In the field of process automation, hardwired process control systems (PCSs) were gradually replaced by programmable systems too. Since the facilities can have huge dimensions, even for only a single product, centralized controller solutions were predominantly used in the earlier days. However, as the cost for installing and maintaining such systems can be substantial, distributed control systems (DCSs) became prevalent. Another reason is that since the introduction of programmable PCSs, the lifetime that a generation of control systems is used has decreased steadily and is nowadays significantly shorter than the one of the controlled facility itself [74]. DCSs can be renewed significantly quicker than centralized systems as they require far less wiring due to their close location to sensors and actuators. Their substitution thus causes shorter idle periods for the facilities and hence lower cost.

Today, programmable PCSs are often based on PLCs that are significantly cheaper in most cases and can be installed in the field. In order to fully qualify them as PCSs, vendors offer special versions of their PLCs which meet the requirements and high quality standards. Besides, IDEs, programming languages and other software [32], tailored to the needs for process engineering, are often provided. In parts, these vary significantly from the toolchains as used for classic PLC programming. The approach presented in Chapter 5 has been developed for and evaluated on PLCs. For that reason, the following paragraphs concentrate on the characteristics and specialties of the latter. It should be mentioned that, thanks to the harmonization of PLC and PCS, it was possible to evaluate the presented concept on a process plant using a Siemens S7-400 PLC.

2.3.3 PLC-based Controllers

When programming PLCs or PCSs some peculiarities need to be considered that distinguish them from many other, e.g., directly microcontroller-based systems. The most prominent

are the programming languages that, in parts, differ significantly from those usually applied for embedded or other software systems.

2.3.3.1 Cyclic Execution

Classically, PLCs programs are ran in the *cyclic execution/cyclic operating mode*. In that mode, the PLC cyclically follows a fixed execution routine consisting of mainly four phases [12]:

1. Internal Checks – In the first phase, the PLC performs internal hardware checks that are predefined by the vendor and which the controls developer need not to care about.

2. Inputs – The PLC samples the signals at its hardware inputs (analog and digital). The results are stored in a special memory section called *Process Image of Inputs* (PII).

3. Program execution – In this phase, the user-defined controller program is executed once. The program has read access to the PII to get the most recently sampled input values. Instead of writing output values directly to the hardware signals, these must be stored in the *Process Image of Outputs* (PIO).

4. Outputs – The PLC applies the output values stored in the PIO onto the hardware outputs. Analog values are converted to output voltages according to the projected configuration.

After the fourth phase, the first one is immediately executed again; the cycle repeats. This execution paradigm has several consequences that must be respected during program development. After being captured, the input values are “frozen” in the PII. Although the terms PII and PIO originally refer to Siemens systems, they will be used for all considered PLCs architectures throughout this thesis.

For a realistic controller program it can be assumed that the execution of phase 3 takes significantly longer than aggregated durations of 1, 2 and 4. As a consequence, in the worst case it takes approximately two full cycles until the PLC’s outputs mirror the reaction on a change of input. This would be the case when the inputs change right after phase 2. However, the principle of PII and PIO guarantees consistent input- and output values throughout the entire program execution. This can be a significant benefit in border cases that can be challenging to detect and, in case of a faulty implementation, might lead to unexpected or undesired behavior.

The systems controlled by PLCs often involve real-time requirements. Since the worst-case reaction time of these depends on the cycle length, an upper bound for the duration of one cycle, i.e., for one program execution, should be defined. The developer must ensure that this bound is not exceeded by the program. For that reason, typical PLC code rarely contains loops. In particular, `while` loops are usually avoided as their runtime is hard to estimate. The upper bound of the execution can be monitored by a watchdog that automatically enters an error state when the limit is exceeded, e.g., to trigger an alarm or bring the plant to a safe state.

2.3.3.2 Program Organization Units

According to the standard, three different categories of software elements, called *Program Organization Units* (POUs), exist. They are distinguished by their internal state and their capabilities to access the peripherals [55, 60, 67].

Function Blocks (FB) represent a certain sub-functionality of the control system, e.g., a PID controller or an RS bistable. A function block can contain arbitrarily many variables, inputs and outputs. Instead of being called or executed directly, function blocks must be instantiated. Each instance possesses its own internal state which persists after its execution and throughout the entire cycle. Technically, that means that the declared variables are stored for each instance separately and kept between two FB calls. In object-oriented programming, a function block would correspond to a class containing exactly one method.

Functions (FUN) are used to encapsulate side-effect free computations. They neither have an internal state nor are they instantiated but called directly instead. Like FBs, functions can declare arbitrary internal variables, inputs and outputs. Additionally, a function has exactly one return value. This value is supposed to be consistent on the inputs, i.e., on the same input values, it is supposed to always produce the same return value. To guarantee the absence of side-effects, functions are not allowed to call function blocks but only other functions. After termination, the internal variables of a function are discarded.

Programs (PROG) form the basis of PLC software. They behave similar like FBs. Additionally, the program can access the PII and PIO of the PLC and declare global variables. Simple controllers consist of one PROG definition, which is cyclically executed, i.e., after termination, the PROG is executed again in terms of the four phases described above.

2.3.4 Programming Languages

The IEC standard 61131-3 [55] defines five different programming languages to implement the body of a POU. For the head, which contains variables and defines the interface of the POU, it specifies a uniform notation. Every PROG, FB or FUN can be provided in one of these languages independently.

The five standardized programming languages are called *Instruction List*, *Structured Text*, *Function Block Diagram*, *Ladder Diagram* and *Sequential Function Chart*. The first two of these are textual languages while the remaining three are graphical ones. The standard and, according to that, several IDEs allow to mix the languages to a certain degree, e.g., a transition condition inside an SFC function block may be provided as LD.

Structured Text is the only of the five IEC languages which has the shape of a structured programming language. It contains elements as, e.g., `IF` or `FOR` to indicate branches and *for* loops. In the industrial context it is still not very widespread. Nonetheless, since the

language allows fast and highly flexible implementations of POUs, there has been a shift towards ST from other languages in the past years, especially from the rather inconvenient LD and IL. Vendors as Siemens additionally try to promote *Structured Control Language* – the Siemens dialect of ST – by emphasizing its advantages over the other languages.

Due to its expressiveness, clear readability and purely textual shape, which allows for a straightforward code generation, Structured Text and its dialects have been used exclusively to implement the SynTACS Runtime Framework presented in Section 6.3. Extensive information about the all five languages can be found, e.g., in [60] or [67].

Languages for PCS Traditional PCS are very complex and include multiple control layers that can be realized very differently. For recipes, for example, there exist plenty of representations. It depends on the PCS which one is used to configure or program the recipe controller. An important one is the *Procedural Function Chart* [34, 58], which has some similarities to SFC. For PLC-based PCSs, there also exist further languages, tailored to the needs of process engineering. Although not explicitly standardized, vendors share at least some concepts regarding these languages. A prominent one is *Continuous Function Chart*, which models signal flows is very similar to FBD. Besides, SFC plays an important role in controlling batch processes.

2.3.5 Safety and Reliability

Many production or process systems impose strict requirements on safety. The reasons are manifold: Delicate materials that are toxic, hot, very reactive, caustic or radioactive must be handled with special care when involved in processes. In manufacturing facilities on the other hand any deformation or damage of expensive components, for instance due to inaccurately placed or stuck workpieces, must be avoided. In all cases, the risk of humans, employees but also local residents, being harmed or killed by malfunctioning production facilities, must be reduced to an *acceptable measure* [102].

In general, *risk* is a measure that involves likelihood and impact of damage. Leveson defines risk as the combination of *hazard level*, *likelihood* and *exposure/duration*[66]. According to Montenegro, it is the product of probability and impact [80] of a damage. Birolini additionally emphasizes the importance of weighting [15].

Usually, risks shall be kept *as low as reasonably practical* (ALARP)[66]. However, it can be very controversial what *reasonably practical* means, especially due to the fact that safety often increases cost. The persons who are in lieu of cost (e.g. product managers) are often not the same as who are affected by the risk (workers, residents, the environment in general, etc.) [66]. To that end, most countries claim automated systems to comply with certain standards by law. The international standard IEC 61508 [56] defines the requirements to functional safety in electric, electronic and programmable electronic systems. By that, it also covers production plants. Which requirements have to be respected in a particular case depends on the *safety-integrity level* (SIL) that applies to the scenario, see [56, 102]. The standard IEC 61511 [57] additionally defines specialized requirements for the process industry, where the highest layers, SIL 3 and SIL 4 apply in most cases.

2.3.5.1 Redundancy

One central concept to achieve safety is redundancy. In the following, two types of redundancies are distinguished as they address different kinds of hazards, caused by different reasons.

Homogeneous Redundancy Redundant hardware can be used to face failures of controllers and sensors during runtime. In that case, two or more controllers, potentially equipped with their own sensors and communication systems, can perform the same calculations by running the same software. If one device fails, the other can fully replace it. This kind of redundancy is called *homogeneous* [80] or *design redundancy* [66].

Diverse Redundancy Homogeneous redundancy is only able to cope with random runtime faults caused by hardware failure. Systematic or design errors would still affect all redundant components though. In these cases, *diverse redundancy*, sometimes called *design diversity*, can be applied. Instead of multiple hardware components executing the same software code, different implementations from distinct developer teams are used.

Goals of Redundancy Redundant hard- or software obviously produce redundant data. Depending on the kind of redundancy and the quality and expressiveness of that data, it can be used to improve the system in one of the following ways:

- As soon as the observations or calculated results deviate (more than tolerated), a fault is *detected*. The plant can then be driven to a safe state and/or alarm can be triggered. In that case, the system is called *fail safe* [80]. Safety is improved but reliability is not.
- Based on additional measurements and models of the system the failure can not only be detected but corrected, i.e., the redundancy is able to reconstruct all necessary data and stay functional. These systems are called *fault tolerant*. Additionally to their safety, the redundancy also enhances their reliability[5].

2.3.5.2 Formal Methods

On SIL 3 and SIL 4, formal methods are recommended or even required. A formal method is a paradigm for the development or analysis of software which is based on a mathematical calculus. A calculus is called *sound* if every result that it yields on a given input is correct. In turn, it is called *complete* if it yields a result for all inputs.

The correctness of the results thus follows directly from the soundness of the method, which can be proven. Additionally to the soundness, some formal methods are also complete, e.g. model-checking [27], a formal verification technique, while others, like static interval analysis, are not.

[5] Indeed, improved reliability can even decrease safety [66], e.g., because the system is not shut down but run on a less confident basis, i.e., without or with less redundancy.

Formal analysis methods have successfully been applied to PLC programs in the past. The tool Arcade.PLC [14, 13] is able to verify PLC programs with respect to an efficiently computable subclass of computational tree logic (CTL). Additionally, it offers several static analyses, such as interval and value set analysis, reachable and dead code analysis and many more [108]. However, formal verification is computationally expensive and often intractable due to huge state spaces.

An alternative to the analysis of existing code is to synthesize the controller or parts of the software from a formal specification straight-ahead. SCT has often be used to serve as a calculus for controller synthesis. Although formally sound, several, rarely discussed conceptual issues exist with this approach. These will be the main subject of Chapter 4.

A different technique is runtime verification. As for analysis, a formal calculus is used to guarantee compliance with formal specifications. However, that is achieved during runtime. The main advantage of runtime verification is that it avoids an exhaustive analysis of the entire achievable state space ex-ante but concentrates on the current and imminent states during runtime, yet still on a formally sound foundation. The price to pay are the additionally required memory and CPU capacities. The supervision of controllers using formally synthesized code as proposed by this dissertation can be counted to the runtime verification methods.

Chapter 3

Applied SCT and Related Approaches

This chapter gives an overview about the most important extensions and improvements of the classical SCT. Thereafter, several SCT tools will be introduced and discussed, followed by some case studies that are relevant for the succeeding chapters.

3.1 Extensions of SCT

Since Ramadge and Wonham presented their approach in the 80's, many interpretations, modifications and extensions on and around SCT, as introduced in Chapter 2, have been conceived. One of the most important among these is certainly the utilization of modularity. In this section, the core ideas of some existing approaches are briefly sketched. The concepts of *in/out automata* and *condition/event systems* are omitted here for the sake of brevity but will be discussed roughly in the sections 5.5 and 5.7.1.

3.1.1 Modular DES

A DES is called *modular* if the specification, the supervisor, the generator or several of these are not given as one but in terms of multiple models which usually represent different parts or aspects of the whole system. The interoperation of these modular parts then yields the semantics that would apply for the entire system.

In most cases, a modular DES provides a set of automata which synchronize via their events. The behavior of the composite DES can be obtained by the parallel composition of all automata. There are exceptions though, e.g., the state tree structures approach by Ma and Wonham [73].

Modular Specification Wonham and Ramadge examined how multiple supervisors can be combined in order to controllably satisfy several specifications at the same time without blocking [120]. For that, the languages of the supervised closed-loop system need to be nonconflicting, which means that for every shared prefix they share at least one word containing that prefix: $\overline{\mathfrak{L}(S_1) \cap \mathfrak{L}(S_2)} = \overline{\mathfrak{L}(S_1)} \cap \overline{\mathfrak{L}(S_2)}$. Wonham and Ramadge showed that this is always the case for so-called *nested* supervisors [120]. An alternative is calculating the *minimally restrictive non-innerblocking solution* as proposed by Chen and Lafortune [24].

The presented algorithm subsequently prunes the language by removing shared prefixes of uncommon words, until the solution is nonconflicting. For regular languages, this solution can always be calculated [24]. Prefix-closed languages are trivially always nonconflicting.

In [19, 18], a slightly different approach is followed: The authors subsequently add (local) sub-specifications to the system in a counter-example guided manner until a global specification can be guaranteed.

Modular Plant While the approaches sketched above consider modularity in terms of multiple specifications and supervisors on one monolithic plant model, De Queiroz and Cury analyzed modular plants. They showed that, for two independently modeled plant parts with disjoint alphabets, synthesis and composition are commutable, i.e., the SCSL for the plants' composition is the same as the composition of their respective SCSLs [28].

Handling distinct parts of the plant independently is highly desirable as the state space of the full composition of all sub-plants, which would be necessary for monolithic supervisor synthesis using the Ramadge and Wonham's algorithm, can become enormously large for realistic systems. In the worst case, the growth is exponentially over the number of automata [51]. This is often referred to as *state explosion* [115, 48, 113, 30] or *exponential state space blow-up* [3, 101].

This method has been refined by Åkesson et al. [3]. They propose a procedure where several supervisors are computed subsequently in order to establish controllability when composed with each other. For each not yet controllable supervisor, the automata of all plant parts are gathered which share an uncontrollable event with that supervisor. Based on their composition, the algorithm synthesizes an additional supervisor, which guarantees controllability and maximal permissiveness when composed with the original supervisors. The method presented in Section 5.8.3 of this dissertation operates in a similar way but uses an even narrower criterion for including a plant into consideration that still guarantees maximal permissiveness when applied incrementally.

Hill and Tilbury [49] lifted the modular plant approach to nonblocking supervisors, which naturally requires larger state spaces and hence causes higher computational complexity.

3.1.2 Nonblockingness in Modular DES

A modular plant model consists of multiple automata, which usually define the behavior of local components. Likewise, it is often the case that specifications also deal with local requirements, such as a safety rule for a certain component. The approach by Åkesson et al. [3] as well as the incremental method in Section 5.8.3 exploit this fact by only considering the relevant plant parts, which keeps the models relatively small. Theoretically, one single supervisor can be built from every (sub)specification and operate concurrently to the others[1]. This works perfectly fine to guarantee compliance with all specifications.

[1] There is no need to compose the supervisors on the automata level. However, an event must be enabled or ignored by **all** supervisors to be admissible; otherwise it is disabled.

Though, to guarantee nonblockingness it is necessary to preclude that the supervisors' languages are nonconflicting, which in principle can involve the entire system. The reason is that nonblockingness is in general a global system property.

Several approaches exist to avoid exhaustive composition by still achieving maximal permissiveness provably. Some of these will be sketched in the following.

Compositional Synthesis The compositional synthesis, presented by Flordal et al. [35], is an elegant incremental method to solve nonblockingness problems using abstraction and masking.

In the first step, the specification is transformed to a nonblockingness problem. This is done by converting it into an additional plant model (generator) G_R by introducing a blocking state $q_\perp$. For every event $e \in \Sigma_R$ and every state q which does not allow e, i.e., $e \notin \Gamma_R(q)$, a transition to $q_\perp$ is introduced: $f(q, e) := q_\perp$. All actions which were forbidden by R would thus block on G_R and hence also on the composition with the plant model. This obviously works for multiple specifications and a modular plant model too. The problem is this way reduced to establishing nonblockingness on $G_{R_1}, \ldots, G_{R_n}, G_1, \ldots, G_m$.

Instead of computing $G = G_{R_1} \parallel \ldots \parallel G_{R_n} \parallel G_1 \parallel \ldots \parallel G_m$ directly, a procedure of interleaved composition steps, masking steps and *partial synthesis* steps is applied. The basic idea is that for each intermediate result the local events, i.e., those that are not shared with any further plant, are anonymized. In general, this can result in nondeterministic automata. If such an event leads to blocking, it can be disabled in the controllable case or isolated in the uncontrollable case, which shrinks the reachable state space. Further, as other automata will neither synchronize on these local nor on the anonymized events, the corresponding transitions can be hidden. The origin and target states are joined, which further decreases the size of the automaton. Each joined state is annotated with the name of the original states it represents. That allows to reconstruct which events were allowed and which were not. After that, the following iteration adds the next plant component and so forth.

Using this method, the size of intermediate automata can be reduced significantly. However, the ordering of the plants is crucial for the algorithm's performance. Its worst-case complexity is the same as for the monolithic approach.

State Tree Structures In their work [73, 72], Ma and Wonham present a framework to synthesize nonblocking supervisors from state tree structures (STS), allowing for very large state spaces in the magnitude of 10^{24} [73]. Instead of considering a set of automata, an STS provides the behavior of the entire system in one hierarchical representation. The STS consists of hierarchically organized states, which are the nodes of a tree. A state can either be a composite AND state, a composite OR state, or a simple (atomic) state. Composite states embody the inner nodes of the tree whereas the atomic states form its leaves.

An OR state has the following semantics: When it is active, exactly one of its child states, w.r.t. the tree structure, must be active too. Each OR state is provided with a *holon*, a structure similar to a classic automaton. Like the latter, it owns a function which defines the transitions between its child states.

For an AND state, all children, typically OR states, are active at the same time. They synchronize via their events as ordinary modular automata. There can also be transitions leaving the entire AND state for another AND, OR or atomic state. These transitions are defined in the holon of the parent OR state. The root of an STS is a composite state which trivially is always active and subsumes the entire system as its children.

Due to their structure, STS can be abstracted very well, allowing for efficient symbolic algorithms. However, the theory behind STS and these methods is rather complex and would exceed the scope of this overview by far.

Hierarchical Approaches with Coordinators The coordinator approach by Su et al. [109] refines the notion of coordinators by Ramadge and Wonham [88]. It exploits the typical locality of specifications that has been mentioned above. The approach is based on the assumption that for each plant G_i, the majority of requirements can be handled locally. Thus, every plant is provided with its own specification H_i where H_i's alphabet is a subset of G_i's.

First, for each plant-specification pair G_i, H_i, a local supervisor S_i is computed using the standard synthesis algorithm. According to [120], these S_i can have conflicting languages and hence lead to blocking. This issue can be tackled using a coordinator.

A coordinator is basically a supervisor for supervised systems, i.e., instead of enforcing $H_1, \ldots, H_n$ on $G_1, \ldots, G_n$, it enforces nonblockingness for $(S_1 \parallel G_1), \ldots, (S_n \parallel G_n)$. The problem is that this generally still requires an analysis of the global system behavior. However, instead of composing all supervised subsystems $(S_1 \parallel G_1) \parallel \ldots \parallel (S_n \parallel G_n)$, which would again end in an exponential blow-up of the statespace, the subsystems are abstracted first. Similar as for the compositional approach described above, some local events are masked out while the states only divided by those events are fused. Finally, the coordinator can be synthesized based on the composition of the abstracted subsystems, which is typically much smaller.

Since the local systems are ensured to be nonblocking due to their local supervision, blocking is only possible as a result of their interaction. This suggests the assumption that it could be suitable to consider shared events only, i.e, those which are contained in the alphabets of at least two subsystems, and mask out all purely local events. Unfortunately, although this would indeed be sufficient to guarantee nonblockingness, it is not maximally permissive in general. The reason is that disabling a shared event in many cases imposes a stronger restriction on the system than actually required. This is particularly the case when one or several disabled local events would be able to reliably prevent blocking "at a later stage" than the shared event would. Finding the optimal *projection alphabet* for the abstraction is essential for the coordinative approach to succeed. If it is too large, the state space grows unnecessarily large and with it the time and memory consumption. If it is too small, the result is not the supremal controllable sublanguage. Unfortunately, the problem of finding that alphabet is NP hard over the number of events.

Closely related is the *aggregative synthesis* method [110]. Conceptually it is a mixture of the compositional and the coordinator approach by introducing coordination on an incrementally growing model.

Further, it should be mentioned that there is another hierarchical approach, developed by Schmidt et al. [94, 95]. It realizes a similar idea as the coordinator method but operates on languages rather than on automata.

3.1.3 Timed DES

In many applications, timing plays an essential role. This is not surprising since DES are dynamic systems. While timing is crucial for continuous systems, DES can in many cases circumvent the need for an explicit time model. This comes from the fact that for discrete problems, the *order* of event occurrences is often of much higher relevance than the actual moments of their happening. Nonetheless, formalizing time can bring some advantages, such as:

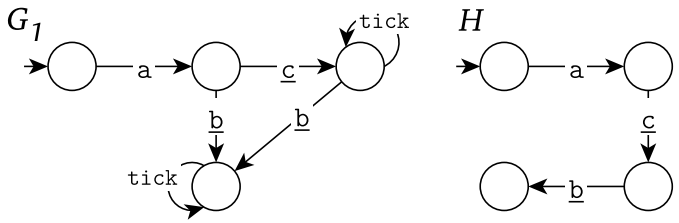
- Provide a foundation for *preemption*
- Extending controllability to otherwise uncontrollable problems
- Increase permissiveness by considering time dependencies

To that end, Brandin and Wonham introduced timed DES [20]. Following their approach, events can be equipped with time constraints over a clock. Each event has a lower bound and an upper bound for the timespan that has to pass before its (next) occurrence. The alphabet is partitioned into the classes of *prospective events* and *remote events*, where the first are assumed to happen before a finite deadline while the second can occur within finite time but need not to.

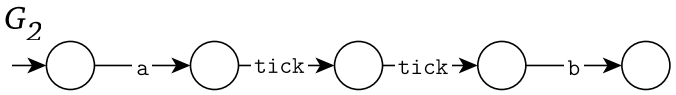
In contrast to timed automata [5], the authors pragmatically discretized time. That allows to treat the timed DES as a finite-state system, provided that the underlying untimed model is also finite. The result is an ordinary DFA, where a special *tick* event is used to indicate that one discrete time unit has passed. The transitions for the remaining events reflect the structure of the untimed model. Further, they respect the events' timing constraints in accordance with the number of tick transitions which have been taken since their last occurrence.

Since timed DES take into account the temporal dependencies of events, they allow for an increased permissiveness in many situations or find controllable supervisors for problems which were uncontrollable using untimed DES.

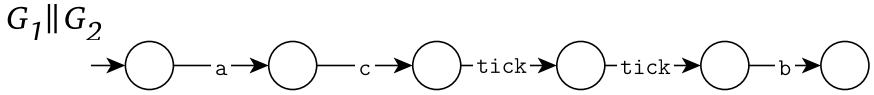
Example 3.1. Consider the plant G_1 and specification H in Figure 3.1a, where $\mathtt{a} \in \Sigma_c$, $\underline{\mathtt{b}}, \underline{\mathtt{c}} \in \Sigma_{uc}$. Ignore the `tick` transitions for the moment. Blocking is not considered. The specification prohibits $\underline{\mathtt{b}}$ to occur before $\underline{\mathtt{c}}$. According to G_1, this order would be possible though. Hence, a supervisor would need to disable a in the initial state. Now assume that it is known that $\underline{\mathtt{b}}$ will not happen before two time units of well-defined length have passed. Also, $\underline{\mathtt{c}}$ is guaranteed to happen right after a. Thus, the tick transitions are added to G_1 indicating that no time can pass between a and $\underline{\mathtt{c}}$. Moreover, G_2 as shown in Fig. 3.1b is introduced. The composition of G_1 and G_2 reveals that, thanks to the time dependencies of $\mathtt{a}, \underline{\mathtt{b}}$ and $\underline{\mathtt{c}}$, H is immediately fulfilled. Thus, a can stay enabled – the permissiveness has increased. △



(a) Plant G_1 and specification H



(b) Plant G_2 captures the timing of a and b



(c) Composition $G_1 \parallel G_2$ shows that H is controllable

Figure 3.1: Timed DES to capture timing dependencies

In this small example, the timing constraints are rather trivial so that the logical ordering of b and c is obvious and could have been incorporated in an untimed plant model as well. This is usually not the case for more complex systems though when time dependencies between multiple events need to be considered which exclude each other only under certain circumstances.

Enriching DES with timing also allows to indirectly enforce controllable events in certain settings. The detailed description of this technique and how it can be used to realize preemption of undesired uncontrollable events is postponed to Section 5.5, where timed DES will be revisited.

3.1.4 Reactive Synthesis

The *theory of infinite games* and, closely related to it, the field of *reactive synthesis* are research areas in theoretical computer science. They are based on classic automata theory, augmented with concepts as *Büchi* or *Rabin* automata, and have their origins in Church's Problem [25, 112] of synthesizing a finite automaton (called "*circuit*") for specifications over infinite languages. Büchi and Landweber [21] solved that problem and, by that, formed the foundation of the field of *reactive synthesis*. Ever since, a variety of amendments, adaptations and interpretations have been contributed to that field. The framework of Ramadge and Wonham is one of them. Nonetheless, due to the control engineering background of the latter, the focuses of both communities drifted apart as well as the employed formalisms. Although there have been attempts to bridge that gap [31], the communities mainly still

follow different objectives and the results are often hardly transferable. Hence, concerns and approaches of reactive synthesis will not be addressed throughout the rest of this dissertation. Instead, the presented methods are discussed on the background of DES and SCT.

3.1.4.1 Symbolic Methods

Already since the early 90's, symbolic methods have been used to encode the transitional structure of automata. The majority of attempts is based on binary decision diagrams (BDDs), e.g., [50, 113, 73]. More recent approaches use *satisfiability* or *satisfiability modulo theories* solvers [116] or the model checker IC3 [101, 103].

Using symbolic methods, i.e., expressing automata by logic formulae, has three major advantages. First, the representation is often significantly smaller in terms of memory consumption than an object-oriented or array-based format, which makes it more suitable for large-scale intermediate or final results. Second, the community developing and improving the methods on SAT/SMT/BDDs is very vivid and has gained impressively performant solvers during the last decade allowing for much larger state spaces than conventional implementations. The third advantage is the flexibility of symbolic methods. Nearly every DES-related problem can be translated to one or even all of the mentioned formalisms, which allows to use the efficient heuristics "on top" of other DES-based techniques.

3.2 Applicatons – Tools and Case Studies

This section gives an overview about existing approaches and successful applications of the SCT.

3.2.1 Tools

First, an excerpt of the existing tool landscape is given. It is in large parts based on the results of [Ney, 2014].

TCT Toy Control Theory (TCT) was the very first SCT tool implementation. It has been developed by Wonham's research group at the University of Toronto. The main purpose was to show the theory's feasibility. TCT is entirely command-line based. Sequences of actions must be defined in terms of scripts. Since its first version, TCT underwent multiple improvements and has been ported onto several operating systems under different names. The probably most important enhancement is the spin-off SmartTCT (STCT) that introduced a new algorithm which improved the tool's resource consumption for complex systems significantly [123]. The formalism of state tree structures discussed above has also been integrated to TCT. Still, the tool addresses theoretical purposes rather than practical applications. Code generation is not provided. It should be mentioned though that TCT has successfully been used in such scenarios, e.g. [65]. Wonham's monograph [117] gives a

comprehensive introduction not only to the original SCT but also to TCT and is still updated regularly.

UMDES and DESUMA The tool UMDES is a C library of algorithms for DES developed at and named after the *University of Michigan Discrete Event Systems Group*. Automata can be provided and stored as UMDES files. The library provides the general operations, such as composition, trim, SCSL, and so forth.

DESUMA [91], developed at *Mount Allison University*, is a graphical editor for UMDES that improves the manual definition of automata significantly. As for TCT, code generation is not supported natively.

Nadzoru Nadzoru is an open-source SCT tool which has been presented rather recently [84]. It includes a graphical automaton editor. The user is supposed to automate the workflow by scripts, but which can be written inside the tool GUI. Scripting is supported by a list of available functions which can be inserted to the script using the mouse. Besides several analyses on the DES models, Nadzoru includes a general-purpose code generation of IEC compliant Structured Text and ANSI-C.

Supremica Supremica is a DES analysis and synthesis tool developed at Chalmers University [2]. It has served as prototyping platform for plenty of the highly sophisticated methods developed by Fabian, Åkesson, et al. (cf. Section 3.1.2). Further, its main functions such as modeling and syntheses are fully accessible via a conveniently usable GUI. General-purpose code generation was present in former versions of Supremica but seems discontinued.

In [Ney, 2014], the suitability of Supremica's code generation for SCT-synthesized controls for two model plants[2] has been examined. Unfortunately, notable additional effort in terms of transformations and manual programming was necessary until the code was actually executable and worked in the desired way. However, it should be emphasized that the experiment succeeded and the controls did indeed work finally.

In comparison to other tools, the variety of advanced available synthesis methods [35, 113, 3, 1, 75, 78], especially to efficiently achieve non-blocking, combined with a good usability, lifts Supremica on an outstanding position.

libFAUDES and DESTool The project of libFAUDES was started by Bernd Opitz and is a C++ library for discrete-event system [81, 83]. Although conceptually similar to UMDES, a much stronger emphasis was put on modularity and extendability. Since then, the library was further developed, maintained and extended by Thomas Moor and his group.

Like DESUMA for UMDES, DESTool represents a graphical user interface to conveniently work and interact with the functionalities provided by libFAUDES. Further powerful extensions of the original library are: *Coordination Control, (hierarchical) I/O systems, fault-tolerant control*, and more. Another extension worth mentioning is *Pushdown*. It lifts the

[2] fischertechnik® pneumatic processing center, see Section 7.3, and *3D robot*

expressiveness of DES from regular to context-free languages using pushdown automata instead of ordinary DFAs based on the approach of Schmuck et al. [96].

Altogether, libFAUDES and DESTool provide a powerful toolbox. They address both theoretical and applicational aspects of DES, including advanced methods of SCT.

Further tools Apparently, the above list is far from being complete. Several further tools and approaches exist. Some of these are Grail [90], Valid and Ver, to name a few.

3.2.2 Case Studies

The tools listed above have been evaluated in several contexts. Additionally, a number of case studies exist which involved a straightforward implementation of the formalisms instead of utilizing one of these tools. In the considered context, those case studies which dealt with the control of physical systems are of particular interest.

Supervisory Control of a Rapid Thermal Multiprocessor This case study by Balemi et al. [10, 9] was one of the most remarkable in the field of applied SCT. It is very relevant for the approaches presented in this dissertation for mainly two reasons. It does not only include a successfully operating end-to-end approach from theoretical foundations to a soft- and hardware realization, but also is its architecture similar to the one of presented here. Its goal was the complete automation of a *rapid thermal multiprocessor* using the SCT. The presented approach is characterized by three major aspects:

1. Distinction between *implicit specifications* and *explicit specifications*, also called *recipes*. These different types of specifications are handled and synthesized independently.

2. From the explicit specifications, a controller is derived instead of a supervisor. In contrast to a standard supervisor, as described by Ramadge and Wonham (cf. Section 2.2), it does not disable a subset of controllable events but produces those. In return, the plant only emits uncontrollable events.

3. Instead of automata, languages are represented by BDDs in the implementation. The presented formalism concentrates on the languages themselves though.

According to the first two aspects, the control loop of Balemi's approach does not contain two logical participants but three. The plant, the supervisors synthesized from implicit specifications, and the controllers derived from the explicit ones. Although motivated differently (cf. Chapter 4), this is very similar to the three-participants concept introduced in Section 5.1.3.

The implicit specifications serve to define constraints and side conditions for the plant. This includes what Balemi calls *fundamental liveness*, the claim that all plant components can always return to their initial state. Indeed, this addresses the standard nonblockingness property of DES rather than real liveness. A recipe is basically an automaton, typically of linear or circular shape, which defines a sequence of events that is necessary to achieve the actual production goal.

In the final setup, the controller communicates with the plant via the supervisor. Additionally to the automated controls, a user can interact with the system through a graphical control interface at the computer. The supervisor checks the operations and enters an error state if the implicit constraints are violated [10].

Although the case study was regarded a success and therefore has often been cited, it seems that the core idea of separating explicit and implicit requirements fell into oblivion and had little impact on the later work inside the community. Instead, the majority of upcoming approaches tried to address both kinds within one supervisor, leading to the *choice* problem, see Section 4.4.1.

Untimed Operating Procedures in Batch Chemical Processes In their paper [121] Yeh and Chang present the synthesis of controls for chemical batch processes, including three examples. In comparison to Balemi, they apply the SCT rather straightforwardly, which gives a good impression about the results that the Ramadge-Wonham framework produces. The examples seem to not deal with physical components. At least, the paper does not mention that the supervisors have actually been tested on real hardware.

In a second paper [122], the authors introduce a method for "*emergency response procedures*", also based on the SCT, which addresses aspects of fault-tolerant control. This way, it has some similarities to the approach by Schuh and Lunze, which is briefly introduced in Section 5.2

Theme Park Vehicles Another case study was subject of Stefan Forschelen's Master's Thesis and the follow-up paper [36, 37]. Its goal was to realize the controller of a theme park vehicle, called *Multimover*, that allows one or more passengers to ride on it through a theme park attraction. An operator can control the movement by three buttons, a *reset* button and one respective button to trigger the vehicle driving *forwards* and *backwards*. When driving, the Multimover is supposed to follow a wire in the ground using a built-in steering motor. Besides, a number of additional control tasks, such as emergency stops and operating LED lights, has to be achieved.

While plant models are defined in terms of modular automata as usual, for specifications the formula-based description by Markovski [77] is used. This allows particularly to formulate state-to-state requirements, i.e., which states may/must/must not be active at the same time. In contrast to that, traditional specification automata allow for state-to-event requirements only, which can be a significant disadvantage (cf. Section 5.7).

Supervisor synthesis has been performed using BDDs as well as the coordinated and the aggregative approaches [109, 110] described in Section 3.1.2.

All control tasks have been specified in terms of DES. The resulting supervisors were embedded into an execution framework implemented in Python. This framework includes certain interpretations of several aspects of the SCT which influence the semantic notion of both the modeling and the supervisors: Like in Balemi's approach, a controller is realized which *executes* controllable events, while uncontrollable events are considered to originate from the plant only. However, in this case study all requirements have to be realized through the supervisor. Whenever the plant emits an uncontrollable event, the supervisor

implementation tracks it on its state space by executing the according transitions. In the next step, a so-called *control decision maker* picks one of the admissible controllable events leaving the new state and executes it. If there are more than one of such, it is assumed that the requirements, which the supervisor has been derived from, allow for any of the associated actions to be eligible. In that case, one of them is picked arbitrarily: "[...] the first controllable event that is allowed [...]", [37]. Finally, the chosen event is executed on the supervisor and the associated control action is transmitted to the plant.

This interpretation of the SCT leads to a reactive system that replies with a control action on every uncontrollable event, if any is feasible. Naturally, this hardly influences how operational requirements – in the wording of Balemi: explicit specifications – are handled. If a specific action is desired deterministically in response to a certain occurrence on the plant, the modeler has to exclude that further controllable events will be available in the succeeding state. This, however, sometimes requires detailed knowledge about other components of the plant, which need not necessarily be directly related to the considered one.

Note further that the scenario targeted by the study does not involve real explicit specifications like a recipe. Instead, the entire behavior of the Multimover is determined by the buttons used by the operator, and the wired route. The controller that results from the cooperation of the synthesized supervisor and the control decision maker only has to react on these inputs as specified. Complex automation tasks, however, are not required. The authors claim that maintenance has been improved as applying changes to the automata, followed by synthesis, took less time than introducing the same change to a manually implemented controller. However, it is questionable whether this applies for arbitrary changes of the requirements or only for specific ones. Further, the effort of initially establishing and debugging the required automata, compared to a conventional implementation, is not mentioned. Also, a comparison between the presented reactive framework and reactive languages such as Céu [6], which are mainly designed to solve tasks as the one from this case study, would be interesting. Apparently, this would clearly have exceeded the scope of a single Master's thesis though.

In the context of another scenario [Ney, 2014], the providing of automata to synthesize a controller through the SCT consumed twice the amount of development time and was significantly more error-prone than a manual implementation. A more detailled discussion on the advantages and disadvantages of formal synthesis methods to derive controllers will be postponed to the succeeding chapter.

Flexible Manufacturing System Another interesting case study is the one presented by Schmidt, Moor and Perk [82]. It is based on libFaudes [81] and the hierarchical synthesis algorithm [94, 95], mentioned in Section 3.1.2. The authors synthesized the controller for a fischertechnik® model plant, similar to the one described in Section 7.3. In order to capture simple timing conditions, Brandin and Wonham's time model [20] was used, i.e., a special controllable event is introduced to embody the passage of time. Recall that disabling this event claims the controller to execute another event immediately. The problem of CHOICE

is addressed using event priorities: When more than one controllable event is admissible by the supervisor, the one having the highest priority is picked.

3.2.3 Industrial Application

Despite the age and maturity of SCT within the scientific community, industrial applications are still pending. There have indeed been several contributions that analyzed the industrial applicability of SCT or SCT-based methods, some even on real components, e.g., [111]. To the best of the authors knowledge, however, there is no actual application in the area of process or manufacturing systems yet in the sense that a plant is productively operative using an SCT-synthesized supervisor or controller.

This stands in harsh contrast to other formal methods, such as formal verification and there is a simple reason for that: Verification can be performed on top of an established existing development process. Although it obviously causes additional effort, and hence cost, it cannot affect the result, i.e., the controller in a negative sense.

Synthesized controllers, however, would replace certain steps in the established development processes. Even if the chances for provenly error-free code increased[3] on the long term, it would still mean a considerable risk in the first place.

[3] Errors can still be introduced during the specification.

Chapter 4

Controller Synthesis with SCT

The automated synthesis of solutions is one of the oldest objectives in classical engineering as well as in software engineering. This section discusses the different meanings of the term *synthesis* in several areas. This ambiguity can lead to misconceptions. After that, the requirements for a successful application of synthesis are analyzed, first in general, then transferred to the setting of DES and SCT. Based on that, a critical debate is added on why applying the SCT for controller synthesis goals in the shape of how it is usually done might be an undesirable path and why a partial synthesis, which concentrates on certain aspects, can be more reasonable in practice. That motivates the main contribution of this dissertation, an approach for synthesizing safety measures, presented in Chapter 5.

4.1 Synthesis – Definition and Classification

The term *synthesis* in general stands for *"the combination of components or elements to form a connected whole"*[1]. Originally, the word was used to describe the mathematical method of deriving a geometrical construction or, later, a theorem from already proven ones. Philosophy transferred the term to a more generalized level: For Aristotle, a *synthetic method* is a general method for realizing an objective from existing ones (in analogy to Plato's *hypothetic method*) [79]. Ever since, synthesis has been closely related to its opposite, the process of analysis, which tries to extract information from a given situation, also by utilizing formal rules. Today, the term synthesis is used in a large variety of scientific and technical areas, amongst which chemistry is probably the most prominent. Nearly all of these agree on synthesis being a process that respects rules which are, depending on the context, formally defined or apply by nature. Nonetheless, in applications which are located on the edge between several areas of research, confusion is not unlikely as different objectives might be addressed by the term.

Supervisory control theory is a branch of control engineering. Hence, for continuous systems, it is not surprising that each system is described by a mathematical model first. In a second step, a suitable controller can be designed and developed based on that model [71]. In discrete control, including the fields of DES and SCT, the same paradigm is applied in general [70].

[1] Source: Oxford English Dictionary

In controls, the term synthesis became established to subsume methods for the (semi-) automated derivation of controllers. In the continuous world, this often refers to finding the right parameters, e.g., for a PID controller, whereas in DES the supervisor itself or its language are *synthesized*.

4.2 Continuous Control, Discrete Control and SCT

Ramadge and Wonham built their theory on the view that the admissible behavior of a system is always a subset of its physical capabilities. In analogy to continuous closed-loop controllers, formal descriptions of the possible and the legal behaviors, both given as DES, serve as input. The output is a supervisor. On the foundation of both the models and the theory itself being correct, it guarantees that the legal behavior will not be exceeded.

The main objective of a continuous closed-loop controller, for instance, for a conti process, cf. Section 2.3.1, is to keep the system's trajectory inside a tolerated deviation from a reference value [39]. In other words, the task of the controller is to guide the system inside an admissible part of the continuous state space and counteract every attempt of the system to eventually escape this state space before it is too late. The SCT aims to realize the exact same goal for discrete systems. In fact, the sentence above is still accurate if the term "continuous" is replaced by "discrete" and "controller" by "supervisor".

In practice, the main tasks of discrete control differ from those of continuous control [70]. The latter deals with stabilizing highly dynamic processes whereas the former typically has the task of coordinating sequences of several steps in a production chain or a recipe. These sequences are often realized on a higher layer of abstraction while continuous control directly deals with the (usually continuous) physics of the system. Hybrid systems aim to merge both aspects into one model.

To summarize, the SCT adopts the notion and objective of continuous control for discrete systems. Both restrict the system behavior to the admissible part of the state space.

4.3 Requirements and Specifications

Software engineering distinguishes between two types of requirements, which are called *functional* and *nonfunctional* [104]. The former define the central functionalities of the product while the latter, also called *software qualities*, define how good and under which conditions these functionalities must be realized. Nonfunctional requirements cover way more aspects than the actual behavior of a system, such as time-to-market, reliability or cost, and hence go far beyond the scope of controller synthesis.

However, in several experiments [Ney, 2014], it turned out that some requirements could conveniently be modeled, while others were difficult and seemed less suitable for SCT-based synthesis. It appears appropriate to analyze the nature of these types of requirements and identify a suitable distinction for them.

4.3.1 Goals of Discrete Control

Following the definition by Lunze [70], the main objective in controlling discrete systems is to find a controller that chooses inputs for the system depending on its outputs in a way that the control goal is achieved. He further distinguishes four categories of objectives: *Reaching a predefined state, realizing a predefined sequence of steps, avoiding prohibited states* and *avoiding prohibited transitions*. The formalization of these goals is called a *specification*. In the remainder of this chapter, Lunze's notion of control goals will serve as a foundation for the further discussion.

The former two of them, *reaching a certain state* or *realizing a given sequence* will be addressed as *productivity requirements* in the following as they define what is necessary for the facility to be productive at all. The latter two types reflect aspects which have to be respected while the productivity requirements are achieved. They will be called *side conditions* from here on. Note that the name shall not suggest that these conditions were less important than the productivity requirements, as actually the opposite is often the case.

Productivity requirements and side conditions seem to be closely related to functional and nonfunctional requirements from software engineering and could be considered as special cases of those. Yet, the latter terms will be avoided in the discussion since they are too coarse and ambiguous.

On a more abstract level, productivity requirements can be perceived to define the *minimally* necessary behavior of the system, i.e., they specify what has to be done *at least*. Side conditions in return delimit the *supremal* legal behavior, i.e., they introduce the boundaries of what a system is allowed to do. Only if the latter shape a superset of the former, the system is actually implementable. In that case, every implementation which meets at least the productivity requirements but does not exceed the requirements of the side conditions is admissible. Figure 4.1 illustrates that. The productivity requirements need all × markings to be covered as these represent the required system behaviors. Side conditions abstractly disqualify entire spaces embracing those behaviors which would violate them.

The following example intends to further improve the intuition for these two categories:

Example 4.1. Consider a tank which is equipped with a stirrer and a heating unit. The productivity requirements claim that a liquid shall be poured into the tank, then heated up to 70° C. Finally the tank shall be drained. The side conditions claims that the heating must never be operated without the stirrer being active to avoid local overheating, which could permanently damage the plant. The productivity requirement needs the specified steps to be followed by the controller. However, it does not explicitly preclude additional actions before, after or between those steps, as long as these to not interfere with the actual goal. This leads to an infinite set of admissible control sequences. From this set, the side condition removes all those behaviors which at any time involve the heating being active while the stirrer is off. Note that in this example still an infinite number of admissible control sequences remains, which is usually the case. If a stirrer was not installed in the tank at all, the cost and power consumption would probably decrease without affecting the productivity requirement. The side condition does not require a stirrer to be present or run as long as the heater is off. Nevertheless, it restricts the solution space to those solutions

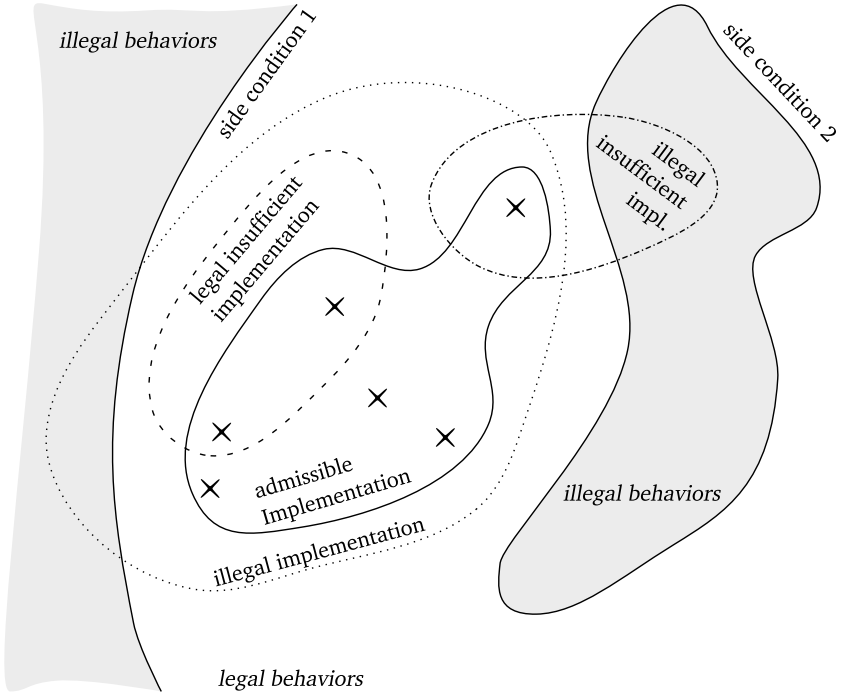


Figure 4.1: Illustration of productivity requirements, represented by ×, and side conditions.

where a stirrer is used while the liquid is heated. Both requirements are necessary to realize proper controls. Due to the potential damage when neglecting the side condition, it can even be accredited a greater importance than the productivity requirement. This, of course, always depends on the particular scenario. △

Multiple productivity requirements are logically joined by considering their *least upper bound*, which specifies what at least has to be performed in order to fulfill all of them. Accordingly, side conditions must be composed by finding a common lower bound of legal behaviors. In Figure 4.1 these bounds correspond to the union of all × and the intersection of all safe (white) areas.

4.3.2 Operational and Declarative Specifications

Besides the two requirement types, there exist also two different kinds of specifications, *declarative* and *operational* ones [59]. An operational, or *model-based*, specification provides an abstracted version of a solution. Hence it may also be considered as an *abstracted implementation*. A declarative, also called *denotational*, specification instead makes statements about properties of the solution and of the problem itself using some logic, e.g., propositional or linear temporal logic. Usually, it encompasses a pre- and a postcondition with the semantics that whenever the precondition holds before the execution of the specified system or operation then the postcondition must hold thereafter. If this implication is true for every possible execution, the specification is fulfilled. In other words, operational speci-

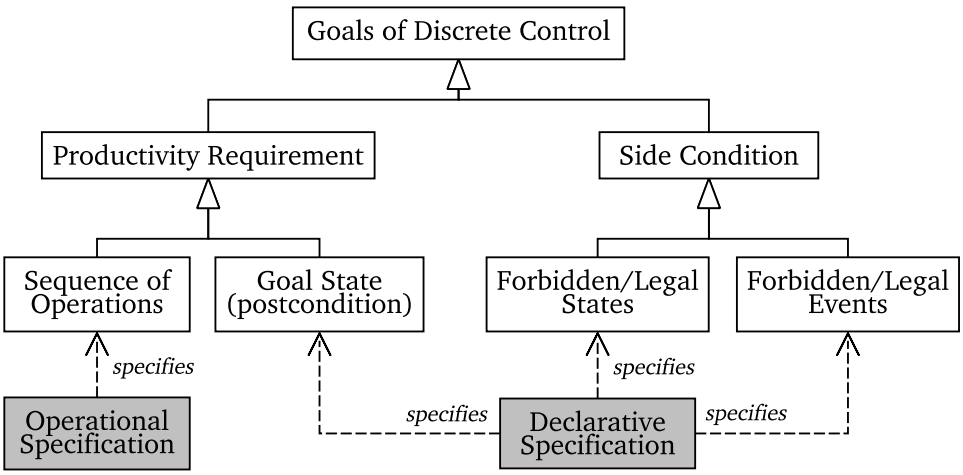


Figure 4.2: Categories of requirements and specifications in discrete control

fications abstractly define *how* the implementation is supposed to work whereas declarative specifications only state *what* it may, must or must not do.

Having this in mind, the four categories of goals in discrete control each fit one of these two specification types. Side conditions can usually best be defined using declarative specifications. Logics allow to declare certain classes of behaviors either undesired or legal using their attributes. In the field of temporal logics and distributed computing, this is called a *safety property* [7], an assumption that can be disproven by a finite counterexample. In contrast, an operational definition of forbidden states or events is hardly possible as that would require either operational instructions for every particular situation which can lead to a violation of the requirement for every single possible implementation, or an exhaustive negative list of all critical implementations, which would usually be infinite.

The two subtypes of productivity requirements must be considered independently. The first one, which defines an initial and a *target state*, precisely meets the notion of a declarative specification. If time or the number of operations until that target is reached are not of particular interest, this corresponds to a *liveness property*, as it can only be disproven by an infinite counterexample [7]. If time, cost or the number of operations required to reach the target state do matter, there are two options: Either the according measure is assigned a hard limit which must not be exceeded, or the value shall be minimized. The first case must be formulated as a safety property again, yet on a domain that supports counting or measuring time. Real-time critical control tasks are a good example. Each of their deadlines can be violated by a finite counterexample, so they can be specified using safety properties exclusively.

In the second case, where no explicit limitations exist, an optimization problem needs to be defined over the solution space. The latter is declaratively specified by the productivity requirement's liveness properties on the one hand and the side conditions' safety properties on the other hand. Nonblockingness, as achieved by traditional SCT, is a declaratively

definable requirement too. Although not precisely a liveness property[2], it can also be used as a basis for that optimization procedure, since it guarantees the goal states being reachable. Assuming that not reaching that goal is the worst possible outcome through the eyes of optimization, the liveness property becomes dispensable in this case.

The second category of productivity requirements, *realizing a predefined sequence of steps*, requires an operational specification, i.e., a model of the desired procedure. The final implementation is supposed to match that model in terms of the latter being a valid abstraction of the former. Figure 4.2 illustrates the relationships of the different requirement and specification types as discussed above.

Usually, a combination of operational and declarative specifications is used to describe a given control scenario in practice.

4.3.3 Specifications through Automata

DES and automata embody a mathematically formalized and graphically displayable view onto a regular language or a system with regular behavior [51, 23]. Automata are also widely used in control engineering and embedded systems to operatively specify the behavior of a controller or a part of it. The organization PLCopen, for instance, provides a reference automaton, along with a textual declarative specification, for every function block in their safety library [85].

Theoretical computer science uses automata as a concept to formally *describe* the behavior of a system or a regular relaxation of it, e.g., for the sake of formal verification. Engineering, including software engineering, instead uses them rather to *prescribe* the desired behavior before it has actually been implemented. In this case, the semantics are often less strict and allow for elements that are not regular by nature, such as arithmetics. In these cases they are often referred to as *state machines* or *state charts* to emphasize their figurative character. In both cases, descriptive or prescriptive, the automata have operational semantics.

The case of automata being used as prescriptive, operational models for code generation, is particularly interesting from the viewpoint of this discussion. States, transitions and transition guards, which are often part of prescriptive state charts, then possess strict semantics and are accurately translated to the according code snippets. The PLC language Sequential Function Chart (cf. Section 2.3.4) is another example for a prescriptive state machine, even though it allows multiple active steps at once and hence slightly differs from ordinary state machines.

Automata can also be used to represent declarative specifications. Thus, every safety property can be translated to a DFA without loss of expressiveness. This is possible because, first, an automaton allows multiple paths even when these are initiated by different controllable operations – this leaves room for multiple acceptable solutions. Second, an automaton in the narrower sense only makes assumptions or imposes restrictions about the symbols (events) of its own alphabet. This way, automata are well suited to formulate requirements which are only based on a small portion of aspects of the entire system, in

[2] Nonblockingness claims that a marker state stays *reachable* whereas liveness demands that it is eventually *reached*.

analogy to temporal logic formulae. It should be mentioned that liveness properties require more powerful structures to be expressed, such as Büchi or Rabin automata [7].

4.4 The Role of Specifications in the SCT

SCT establishes controllability on a given regular specification language (or automaton) on given system models. This is done by imposing a controllable sub-behavior (sublanguage) of the specification, i.e., by further narrowing its restrictions. It has been discussed above that reducing a specified behavior is only admissible for side conditions, as they define the supremal legal behavior. Cutting down on productivity requirements would instead result in an insufficient realization as that would mean to drop certain production goals. Indeed, SCT specifications typically have a declarative character. Apparently, it would also be possible to feed the algorithm with a full, linear sequence of operations, i.e., an operational specification of productivity requirements. Section 4.5 discusses why this is not desirable.

There are many problems for which providing a declarative specification is much easier than finding an implementation. A simple example is the square root function: The specification is as compact as: *The square of the result always equals the input value.* The research area of *program synthesis* deals with such problems by consulting heuristics for automated proof tree deduction. However, that is far beyond the capabilities of SCT.

It seems that SCT is rather problematic when it comes to the realization of productivity in general. The following sections discuss that in detail and analyze the formerly introduced case studies regarding this concern.

But first, based on the above observations, the main thesis of this Chapter can be anticipated:

1. Supervisory control theory is a well-suitable framework to enforce compliance with side conditions which are not controllable by nature, i.e., which involve undesired yet uncontrollable events.

2. The derivation of a full controller is rather impractical as the SCT lacks appropriate support for productivity requirements.

The validation of the first point, is the central objective of the tool implementation presented in Chapter 5. The second aspect has been one result of the case studies by [Ney, 2014].

4.4.1 Using Nonblockingness to Achieve Productivity

Often, marker states and nonblockingness are employed to achieve productivity. Intuitively that makes sense, since a plant which does not operate appears like it would block. Claiming that a certain state – representing the production goal – is reached is, in accordance with Lunze's goals of discrete control, an appropriate way to declaratively specify what the controller is supposed to do. Nevertheless, in the context of SCT this is a fallacy, as the goal of synthesis is still maximal permissiveness. As long as the generator, i.e., the plant, does

neither (uncontrollably) head towards a state from which that goal would be unreachable nor to a state forbidden by the specification, the supervisor would not intervene.

The CHOICE Problem Several approaches have been presented of how to tackle this issue and enforce productivity on the plant. There seems to be a broad consensus on that controllable events are rather suitable to express operations of actuators, whereas uncontrollable events represent the dynamic responses of the plant to these actions, which are measured and captured by sensors [9, 10, 29, 37, 50, 82].

Based on this perception, Dietrich et al. distinguish between a supervisor and its *implementation* [30], an automaton that allows at most one controllable event per state. This event then corresponds to the control action which is to be executed next. Whether it is always triggered immediately, e.g. [37, 82] or, in case of timed DES, only when an action is due [20, 117], depends on the notion of the execution framework. The problem of selecting one amongst multiple admissible candidates has been formulated by Fabian and Hellgren [33] and is called the CHOICE problem.

In a number of case studies, it rather seemed to be treated like a peripheral matter, solved, e.g., by fixed priorities [82] or by random choice [37].

However, when automatic controller synthesis is the goal, the CHOICE problem should be considered as one of the most crucial and central aspects of the entire procedure. Unfortunately, even the identification of what characterizes a good solution to it is highly nontrivial [22].

When utilizing simple criteria for CHOICE, the specification needs to be designed tight enough to lead to a usable implementation afterwards. For the presented case studies, this worked well indeed. Nonetheless, during the specification phase it is not an easy task to determine when the specification is unambiguous enough to lead to the expected controller finally. At least, this is not possible if the specification designer is not aware of the details of the final result in advance. Thus, providing a sufficient set of specifications for that is a manual task and can be challenging. Besides, Schmidt et al. pointed out that "[...] profound knowledge of DES theory and the use of a suitable software tool are essential [...]" [82].

For a more reasonable solution to CHOICE, it is necessary to provide a metric, such as cost, to rate the available options. The supervisory control problem then automatically becomes or at least involves an optimization problem. The *optimal control theory* approach by Sengupta and Lafortune [100] as well as the *optimal directed control framework* by Kumar et al. [52, 53, 64] address this optimization aspect. Unfortunately, they only work for monolithic models.

4.4.2 Requirements and Specifications in Existing Case Studies

Considering the aspects discussed above in the context of the case studies described in Section 3.2.2, one similarity stands out: Explicit specifications of productivity requirements are avoided. Although the introduction of the studies usually involves a textual description of either what the system shall do (e.g., Theme Park Vehicles [37] and Flexible Manufacturing

System [82]) or even about the order of necessary actions (Untimed Operating Procedures in Batch Chemical Processes [121]), these requirements are almost never formalized explicitly. Indeed, marker states are used to identify goals to a certain degree. But still, a proper CHOICE mechanism is required to realize productivity through nonblockingness.

The study *Theme Park Vehicles* leaves most control decisions to the wire that is installed in the ground and to be followed. In this case, the productivity requirements are literally hardwired into the system. The remaining instructions are given by the user via the buttons, obviating the need for further productivity requirements. The remaining control tasks are all purely reactive, which the SCT is not precisely designed for [22]. Since the implemented control decision maker is based on randomization, additional specification automata are required to restrict the solution space to the actually desired result. Note that side conditions were successfully enforced using the SCT in this example. For plant automation in general, however, it cannot be assumed that all choices regarding productivity would either be prescribed by the hardware itself or made by a human operator. Instead, it is one of the main objectives of factory automation to allow more flexible hardware and leave control decisions to programmable, software-based controllers without the need of human interaction.

In case of the *Batch Chemical Processes* study [121], an implicitly given productivity specification is missing. There is no "wire" to follow. Hence, even larger specification automata are required that are not directly related to an actual side condition. Whenever there is a fork of two or more possible control actions, one specification implicitly forbids all but one alternative, such that finally exactly one control strategy remains. Some specifications, called "auxiliary automata", even have rather operational character, meaning they predetermine many decisions in advance. The resulting supervisors are so linearly shaped that, in consideration of how SCT synthesis works, it does not appear very convincing that they are the result of a specification representing arbitrary factory requirements. Instead, they seem to precisely circumscribe a certain, desired solution, which also could have been implemented right away, once it is known.

Another, though entirely conceptual, example can be found in [30]. It is based on [47] and synthesizes the controls for a dosing tank, equipped with an inlet and a drainage valve, two level sensors, one at the top and one at the bottom of the tank, and a stirrer. The productivity requirement is textually declared as "supply a defined amount of liquid material to subsequent process units" [30]. Additionally, the side condition "The fluid must be stirred since it will gelatize [sic.] [...] if the substance is not in motion." is given.

From the productivity requirement, multiple automata are derived. One of them is depicted in Figure 4.3. It restricts the behavior in the sense that, it disallows to open the drainage during the procedure until the tank is full. The event *openA* opens the inlet valve, *openB* the drainage valve. *L1on, L2on, L1off* and *L2off* are uncontrollable events which represent the two level sensors of the tank. *L2on* indicates that the upper level sensor, corresponding to the volume of one dose, has been reached whereas *L2off* occurs when the level decreases below that sensor. *L1on/off* work accordingly for the second sensor indicating that the tank contains a notable amount of liquid at all, i.e., the tank is considered empty after *L1off* has occurred. Note that the automaton only has the semantics that it prohibits certain events state-wise, e.g., *L1off* in the bottom-right state, which would correspond to emptying the tank before the dose is reached. The remaining automata

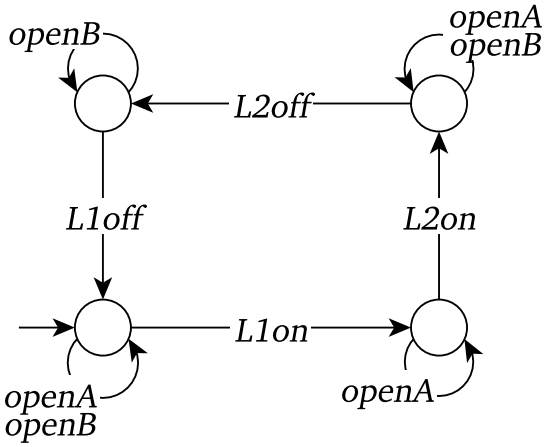


Figure 4.3: Specification *Empty-full-empty-cycle*, redrawn after [30]

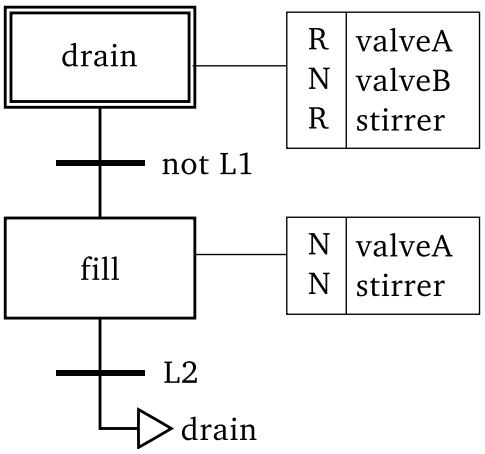
further restrict the system, e.g., by forbidding gelatinization, direct pass through or closing valves before a change of level has been detected.

By these automata, the state space is again declaratively reduced by ruling out those instances that do not meet the productivity requirement. Of course, eliminating potential solutions that contradict the desired goals is reasonable. Problematic is that the requirement must be manually broken down into a set of safety properties until the desired solution is found – a criterion which also must be checked manually as there is no formalization of the productivity requirement. There is even one automaton of 6 states and 20 transitions that only servers the purpose to avoid the stirrer being switched on and off when that shall not or even must not happen. The former refers to the productivity – a controller that turns on and off the stirrer several times before the tank is filled for the first time would be legal but not optimal – while the latter is a side condition due to the risk of gelatinization.

In case of this example, the productivity requirement can be described by one sentence. An operational procedure which realizes that requirement by only triggering controllable events and branching on uncontrollable ones comprises eleven lines of pseudo-code or, alternatively, a 2-step SFC. Both are listed in Figure 4.4 (“R” resets a variable to *false*. “N” sets a variable to *true* as long as the step is active and resets it to *false* thereafter.). The SCT realization instead requires 10 automata with a total of 33 states and 56 transitions to be manually defined before synthesis is invoked. These automata have to be defined closely “around” the desired solution in order to actually yield it. For more complex scenarios with much more sophisticated recipes, it appears unlikely that all necessary automata will be provided correctly in the first place in a way that a solution is obtained which meets the recipe. Even more unlikely is a reduction of effort, especially due to the necessary manual double-checking of the result. Since synthesis is correct by construction, verification against the original models is pointless. Instead, a separation of productivity requirements and side conditions would help to guarantee the latter to be satisfied. The automata of both could be used as input for synthesis at the same time. Nonetheless, this would not reduce

```
1  while(true)
2      closeB
3      openA
4      stirrerOn
5      while not L2On
6          wait
7      closeA
8      openB
9      stirrerOff
10     while not L1Off
11         wait
```

(a) Pseudo code



(b) SFC implementation

Figure 4.4: Operative controller implementation for dosing tank

the overhead and error-proneness of the transformation from productivity requirements to safety properties.

In the study Flexible Manufacturing System [82], timing constraints are invoked to achieve progress. These allow the productivity requirements to be expressed as a safety property straightforwardly. The idea is simple (cf. Section 3.1.3): A deadline is introduced for every action that the controller has to trigger eventually for the sake of productivity. Since time is represented by controllable *tick events* [20, 117] a reaction must be enforced before time is allowed to continue to elapse. When there is only one controllable event leaving the current state, the corresponding operation is guaranteed to be executed immediately. It is up to the framework implementation to realize these model semantics. Although using the notion of timing to guarantee productivity has a certain elegance, two problems remain. First, it still requires a manual transformation from the textually given sequential requirements to declaratively specified side conditions. Second, following this paradigm, control actions might be unnecessarily delayed by the supervisor as maximal permissiveness is a fundamental concept of SCT. In this case the result would, instead of being optimal, rather be the worst, still legal solution.

4.5 Operational Specifications for Synthesis Techniques

Balemi [9, 10] used operational specifications (“explicit liveness specifications”) to formalize productivity requirements and declarative (”implicit”) specifications for side conditions. Both are given in terms of automata, however, designed towards divergent paradigms and hence differently shaped. Automata defining operational specifications usually consist of long sequences of succeeding states and transitions with few branches. Forks labeled by controllable events are avoided as these correspond to situations where the expected behavior is not clearly defined. In contrast, declarative specifications leave the greatest

possible degree of freedom to the system by prohibiting only those transitions which contradict the represented requirement.

Operational specifications have one inherent drawback, which becomes crucial when used for any kind of automated syntheses. First, it must be noticed that a concrete controller or, generally speaking, software implementation can be referred as an operational specification too, even though one of very fine granularity. It precisely instructs the hardware about every single action, depending on the current and past inputs. Operational specifications as used in classical (software) engineering are usually incomplete and hence must be accompanied by declarative ones, i.e., postconditions on the result. Compared to the final implementation, the elicited operational specifications are usually formulated more coarsely, either verbally or through well-defined abstractions. They rather represent a procedural pattern than full information about what shall be achieved. Filling the gaps inside the operational specification towards the full implementation, according to the declarative specifications, is a manageable task for a human programmer. His or her experience, together with creativity and intelligence help interpreting and thus *interpolating* the operational aspects correctly, i.e., in the sense that the declaratively given postcondition is guaranteed. Still, misconceptions are possible and an apparent source of software errors.

The problem is that high-level operational specifications need abstractions. By definition, a proper abstraction cannot provide the same amount of information as the concrete solution. In return, to be complete with respect to the desired goals, an operational specification must contain at least the amount of information as the solution does. Otherwise, the system is *underspecified* and hence, from a formal point of view, underdefined.

Of course, it is possible to unambiguously describe a certain procedure using high-level operations. Basically, that is the core idea of structured programming languages, such as C, where subroutines are encapsulated in high-level functions. This, however, is a matter of software architecture and code reusability but not of synthesis, as all low-level functionality needs to be explicitly defined at any time. Thus, for purely operational specifications, there are only two possible scenarios: Either the system is underspecified and a human needs to *reasonably* fill the gaps, or the system is fully defined and synthesis is not necessary. Since supervisory control is not able to derive solutions from declaratively specified productivity requirements, all aspects which are missing within the operational specification need to be covered by side conditions. For that reason, the size of the models, in terms of their informative content, is likely to exceed the size of the desired synthesis result. Intuitively, that makes sense as no synthesis technique has the ability to divine unstated requirements.

In Balemi's *explicit liveness specifications*, several events (operations) are abstracted by the one that is considered most important. Declarative specifications and plant automata define the circumstances which make these events possible. This way, the outcome of synthesis is a refinement of the abstract operational sequence which is guaranteed to respect the side conditions. This worked well for Balemi's sample recipe. But simply abstracting a procedure by one of its events can be problematic if the final result is to be synthesized:

- Whenever the side conditions or plant models do not restrict the occurrences of that event to precisely those scenarios the designer of the operational automation had in mind, synthesis is likely to produce unexpected results.

- Optimality (cost, time, etc.) is still not achieved or even approximated by any means.
- Full specification is still necessary.

Indeed, the presented approach avoids exhaustive declarative side-condition specifications "around the solution" and instead uses an operational "recipe automaton" to reduce the solution space towards the desired controller. Technically, the applied method stays the same, except that the input specifications are stricter in the operational case. The most compact way to operationally specify the system would again be a linear automaton only containing high-level events which represent, or encapsulate, well-defined sub routines, i.e., a concrete implementation.

4.6 Conclusion

A novel method can be reasonable if it has the potential to either improve the quality of the results or reduce the effort, and thus cost, in producing them. A discrete control task is characterized by two kinds of goals: productivity requirements and side conditions. An operational specification for the former kind needs to be either complete or accompanied by sufficiently restrictive declarative side conditions and information about the plant's capabilities (e.g. in the shape of plant automata). In both cases, at least the same amount of information is necessary as for the solution itself as, otherwise, the system would be underdefined. The same holds if all productivity requirements are circumscribed by "artificial" auxiliary side conditions until the solution space is reduced to reasonable ones. In both cases, the effort of providing the necessary modeling is expected to be higher than for a particular implementation which respects all real side conditions.

Supervisory control theory The SCT is based on regular languages and automata. Its specifications are only able to restrict a given behavior, which rather suits the needs of continuous than discrete controllers. There is no way to provide information about *what the system is supposed to do* as input information to the SCT but only what it may or must not do. Accordingly, in most case studies, the behavior was enriched by randomized control decision makers and restricted by declaratively specified auxiliary side conditions until the productivity requirements were met. Besides, the fundamental objective, maximal permissiveness, conceptually clashes with the idea of a straight ahead, deterministic and predictable final controller. Due to that, even the simplest aspects regarding optimality, which a human programmer would naturally follow, are not considered in SCT-synthesized solutions. The case study developed in the context of [Ney, 2014] substantiates these points. Providing the models for SCT synthesis required approximately twice the amount of time compared to a conventional controller implementation. It turned out more error-prone and the result was less adapt- and extendable.

> "Eine geradlinige Entwicklung ist ohne tiefgehende Planung und ein ausgeprägtes Bewusstsein für die Arbeitsweise mit ereignisdiskreten Systemen sehr

> erschwert. Bei instinktiver und modularer Modellierung ergeben sich oft nicht-steuerbare Systeme oder solche mit unbewusst falschem Verhalten durch kleine Fehler in der Modellierung. Eine manuelle Analyse des Ergebnisses ist aufgrund der Größe des resultierenden Zustandsraumes ausgeschlossen. [...] Insgesamt zeigte sich, dass die Entwicklung unter Nutzung der Synthese deutlich umfangreicher und aufwändiger als eine direkte manuelle Implementierung einer Steuerung ist" [Ney, 2014, p. 60][3]

The main focus of all discussed case studies was showing the feasibility of SCT for realizing hardware controllers but not to critically evaluate its suitability. The presented examples in all studies would have been easy to implement by hand compared to the number and complexity of the required automata.

The SCT itself is a formal framework which intends to formally reason about controllable and uncontrollable behaviors of discrete-event systems, including the notion of the supremal controllable sublanguage. However, it seems hardly suitable for the synthesis of runtime controllers for technical systems. The author believes that the root of this misconception could be the ambiguous uses of the terms *controller* and *synthesis* in different areas.

Loss of Redundancy Finally, one general point remains which has not been considered yet. Syntheses are meant to provide solutions that are correct-by-construction. This brings up the problem that these can only be correct with respect to the originally provided specifications.

In formal verification, the specifications do not *influence* the result. Instead, the latter is checked against the former. This can be seen as a form of *design diversity* (cf. Section 2.3.5), at least for the software implementation, as the specification and the implementation are manual tasks, usually performed by different persons. In contrast to that, all syntheses have in common that this redundancy is lost [17]. Verifying a synthesized software against its original specifications is obviously pointless. That means, one must fully rely on the correctness of the specifications, which, at least in case of the SCT can be much more sophisticated to design than an equivalent (operational) implementation.

Program Syntheses from Declarative Specifications Synthesis techniques which are based on declarative productivity requirements, i.e., that derive a procedure based on pre- and post-conditions can indeed have the potential to actually decrease the manual modeling effort. One example, the square root function, was already given above. These methods are summarized under the term *program syntheses* and apply (semi-)automated verification techniques backwards. Most of them invoke heuristics and symbolic methods to deduct typical operations for certain criteria. In the past years, *deductive*, *counter-example guided inductive* (CEGIS) and *syntax-guided* syntheses [4, 11, 45, 46, 76, 106, 107] have gained

[3] Translation by the author: Without extensive planning and deep knowledge about discrete-event systems, a straight development is very difficult. Intuitive and modular modeling often yields systems that are uncontrollable or show wrong behavior due to small mistakes in the models. Manual analysis of the result is impossible for the size of its state space. [...] Altogether, development turned out more extensive and required significantly higher effort compared to manual implementation of a controller.

impressive results. Unfortunately, the methods are still struggling with large problem and solution spaces and hence far from being applied in practical or industrial scenarios.

Essence The supervisory control theory by Ramadge and Wonham is a theoretical framework that transfers the notion of systems and control onto discrete-event system borrowing the formalisms and algorithms of automata theory. It lacks in realizing productivity requirements and hence misses one of the two main aspects of discrete control. Although several case studies show that it is indeed possible to synthesize proper controllers using the SCT, it is hardly practical and seems unlikely to reduce the effort or enhance the quality of the results. In the current stage of synthesis methods and particularly in the field of the SCT, the author proposes to stick to conventional implementations by-hand in order to realize productivity requirements. The PLC programming languages specified by the IEC [55] include several high-level paradigms to allow for a convenient and problem-oriented implementation, such as the sequence-of-steps perspective realized by SFC or the signal-flow approach of FBD. For mainly reactive systems, alternative implementation concepts, such as Céu, could be utilized in the future [93].

Nevertheless, the SCT is an elegant way to establish controllability for potentially uncontrollable side conditions. Since it constantly follows the paradigm of maximal permissiveness, it is perfectly suitable for a minimally invasive *controller supervision*. Particularly in situations in which the controller tends to be changed frequently or compliance with all side conditions is difficult to ensure, runtime supervision can be a solution to guarantee safety. The succeeding Chapter 5 presents an approach which addresses precisely this kind of application of SCT.

Chapter 5

A Concept for Runtime Supervision of PLC Programs Using DES

The supervisory control theory by Ramadge and Wonham aims to enforce a given declarative specification on a discrete-event system by limiting its uncontrolled behavior. The framework follows the philosophy of maximal permissiveness, which means that the system is restricted as little as possible. The previous chapter pointed out why this concept is rather unsuitable to handle those requirements which prescribe what the system under control is supposed to do and which steps it shall follow to that end. Nevertheless, besides these requirements on productivity, there usually exist numerous side conditions which must be respected. Demands on functional safety are probably the most prominent examples.

The origins of the requirements and the aspects they are concerned with can be manifold. Some, for example, arise from the characteristics of delicate goods, some originate from the physical capabilities of the plant while others formalize the demands of legal safety regulations. What they all have in common, though, is the fact that they reduce the space of permissible behaviors of the system under control. It is the task of the controller to enforce compliance with these restrictions while accomplishing productivity. However, correctness cannot always be guaranteed. Particularly in case of complex, manually implemented controllers, critical border cases can be missed quickly. This chapter introduces a new concept for DES-based supervision of PLC programs with the goal of enforcing these side conditions during runtime. Its core is a framework which monitors the plant along with its controller and validates the controller's decisions. In order to deal with uncontrollability, the specifications that reflect the side conditions are not used for supervision directly. Instead, a supervisor is synthesized using methods that are inspired by and closely related to the original SCT.

This chapter consists of the following contents. First, an introduction to the addressed problem are given including a description of the primary use case. Then, in Chapter 5.2 related approaches are discussed that, in contrast to the ones presented in Section 3, are not based on SCT but address a similar use case in practice. Section 5.3 gives an overview on how specifications and the system are to be modelled. It is followed by Section 5.4, which introduces basic functionalities such as composition and synthesis. On that basis, Section 5.5 discusses some conceptual limitations of SCT when dealing with unstable states and presents a possible solution to these, called *preemption*. An alternative to that is discussed

in the follow-up Section 5.6. In Section 5.7, conditional transitions are introduced as a way towards more intuitive and efficient modeling by relating transitions to existing states. Finally, Section 5.8 puts the presented concepts and algorithms on a formal basis and proofs the soundness of the approach.

5.1 Introduction

The presented concept aims to examine the practicability of supervisor synthesis in the context of production and process control. Compared to those approaches and tools that primarily address algorithmic and complexity aspects of SCT, a stronger emphasis is put on a convenient and intuitive way of modeling and the integrability of the solutions to soft- and hardware platforms as used in industrial automation.

5.1.1 Motivation

Side conditions in the sense of Chapter 4 play an essential role in industrial processes. Particularly for the sake of safety, it is not unusual that additional measures, realized in both hard- and software, are added to a control system to preclude violations of the given requirements. These measures can approach different scenarios. Some aim to reduce risks caused by human interaction while others primarily address hardware issues, e.g., due to mechanical blocking or damaged components. Another source of malfunction can be an inaccurate controller implementation itself. Software tests are a good way to find bugs in an implementation. Nonetheless, exhaustive, systematic black- and white-box testing is still rarely applied in factory automation. Besides, several possible ways exist how safety critical misbehaviors can come to execution on real hardware:

- A failed test indicates that the software contains errors whereas the opposite is not necessarily true
- The safety critical behavior arises only under very special circumstances which have not been considered while testing although they are possible
- The controller is still under development and tested or parameterized on the hardware without executing an entire test suite after every change
- The controller, its parameters or implementation undergo frequent changes

Formal verification techniques, such as model checking, theoretically can be invoked to ensure compliance with all side conditions on every possible execution. Unfortunately, this is not always feasible due to the significant amounts of required computing power. The presented work uses a different approach. Instead of statically analyzing[1] the controller in advance, its actions are monitored during runtime, with respect to the imposed side

[1] Some sources count model checking to the dynamic analyses instead. Anyway, the behavior of the controller implementation is analyzed pre-runtime.

conditions, and intervenes if necessary. In the context of future Industry 4.0 applications, especially the last of the above bullets will become vital. When methods, algorithms and controllers are dynamically distributed, shared or borrowed via the Internet [16, 114], mechanisms that guarantee safety for humans and the equipment itself will be inevitable.

5.1.2 Setting

The addressed base scenario consists of a hardware plant whose sensors and actuators are connected to the I/O ports of a PLC. The latter hosts a controller program which implements the productivity requirements of a process or recipe that shall be realized on the plant. Although the controller is supposed to respect all specified side conditions, it is assumed that this has not been formally verified and hence cannot be guaranteed with certainty.

Many side conditions can be broken down to regular properties, which hence can be formalizd declaratively in terms of discrete states and events, i.e, as finite automaton (cf. Chapter 4). The developer of the controls is not necessarily the same person as the one who provides these models. In case of foreign control routines, as loaded from a future Industry 4.0 cloud service, an employee of the plant operator could create the models to make sure that the controller does not damage the plant and respects all relevant restrictions.

Additionally, information about the plant's physical operating capabilities should also be provided in order to establish controllability and increase permissiveness. The user provides *plant model* automata to that end. These only need to contain events which are actually related to or relevant for the specifications. An exhaustive behavioral model, as it would theoretically be required to synthesize a *controller* from any kind of productivity requirements, is usually not necessary. A single supervisor is synthesized from each specification. Finally, all supervisors are subsumed in an executable framework which is generated from the automaton representations and can be integrated to the original PLC project. The framework also monitors the relevant signals of the controller and the plant during runtime. Operations that have been approved by the supervisor, are forwarded to the plant while rejected ones are blocked.

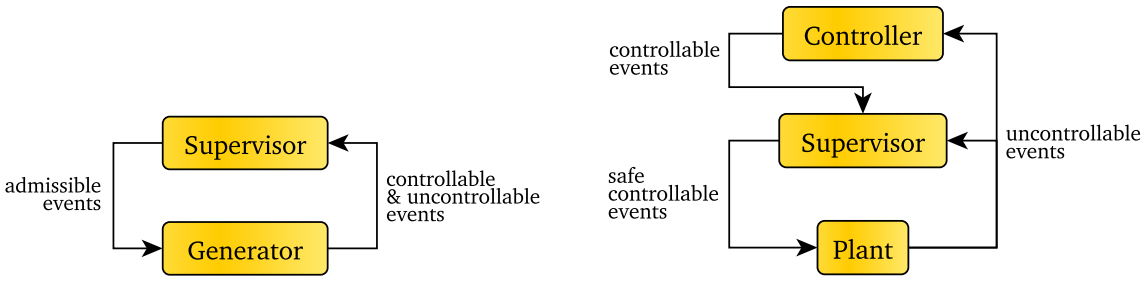
5.1.3 Framework

Supervision should operate in a minimally invasive way in order to minimize interference with a correctly working controller. That includes the following goals.

- Allow to implement the controller without knowledge of the supervision.
- The controller's actions and output signals should be distorted as little as possible.
- The supervision itself shall be maximally permissive.

Nonetheless, a violation of all specified side conditions shall be excluded by any means.

The supervisors themselves are synthesized using the concepts and algorithms of DES and SCT. However, the closed-loop structure differs from the one defined by Ramadge and Wonham and as presented in Section 2.1 in the sense that it is adapted to three participants:



(a) Ramadge and Wonham's control loop (b) Control Loop of presented approach

Figure 5.1: Comparison of control loop paradigms

the plant, the controller and the supervisor(s). Figure 5.1 shows the two control loops schematically.

Similar to many case studies and applications, uncontrollable events are used to represent things that originate from the plant and therefore cannot simply be disabled. Actions that are triggered by the controller are modeled as controllable events as the supervisor can inhibit them.

5.1.3.1 Connection to SCT

In traditional SCT, the supervisor can disable controllable events and by that declare them as inadmissible to occur next. If it is maximally permissive, it disables exactly those events which would leave the SCSL of the closed-loop system. Transferred to the setting of the presented approach, such an event corresponds to a controller action that leads to a state from which undesired behavior cannot be prevented anymore by disabling events. This state does not necessarily need to be specified illegal itself. To be outside the SCSL it suffices that a transition or a cascade of several transitions exists that leads from that state to a specification violation and is uncontrollable. The strength of SCT in this context is its ability to estimate the potential consequences of an action with respect to what is uncontrollable and hence cannot be averted, but also to what is possible to happen in which situation. Nonetheless, some major changes have been introduced to the formal framework, mainly to account for the circumstance that an existing controller shall be supervised instead of being replaced by the supervisor. Instead of transforming the modeling concepts onto existing formalisms, a complete and self-contained formalization is given and used to prove the algorithms' correctness.

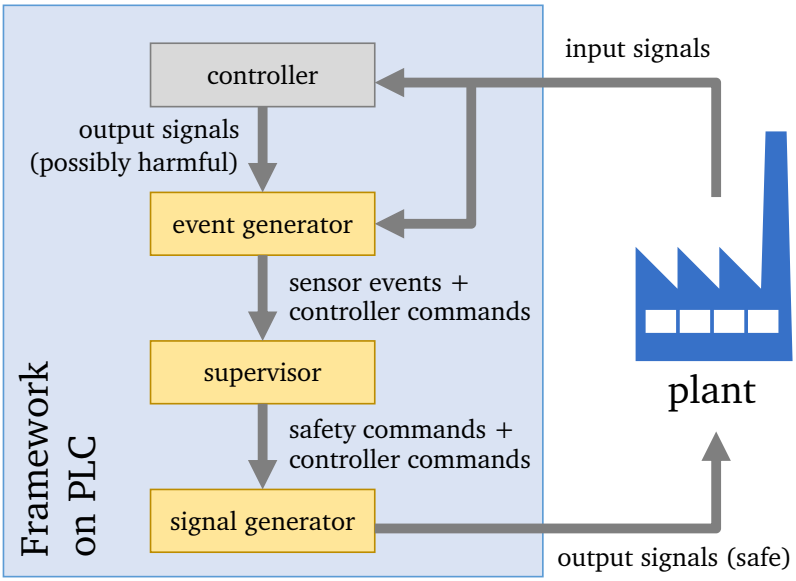


Figure 5.2: Execution cycle (Figure taken from [43], slightly adapted)

5.1.3.2 Execution Cycle

In order to realize supervised controls on the target PLC, a runtime framework is required to coordinate the control flow and provide the necessary data to supervision. In the presented concept, the controller is executed in advance of the supervisors. This stands in contrast to the majority of other approaches and case studies where the supervisor offers the controller/control decision maker a variety of admissible events to choose from. The reason is that the framework is supposed to supervise arbitrary controllers. Thus, it cannot be assumed that the controller is able to interpret a list of admissible events properly or that its logic operates in terms of discrete events at all. Instead, the controller is treated as a black-box which the entire event system is built around.

The supervision framework exploits the cyclic execution model of PLCs (cf. Section 2.3.3.1) as sketched in Figure 5.2. It is dynamically generated from the synthesized supervisors and the event definitions provided by the user.

First, the controller is executed. Remember that it only operates on the PII and PIO caches but cannot access the hardware in- and outputs directly. An *event generator* analyzes both input and output signals on these caches to detect occurrences of the defined controllable and uncontrollable events. Next, the supervisors are consulted. They execute uncontrollable events first since these had physically occurred before the controller was invoked. After that, the admissibility of the controllable events caused by the controller is checked. If one of them is forbidden/disabled by any supervisor, it is marked as *rejected* and the current state of all supervisors stays unaffected by that event. Only if all supervisors agree that an event was triggered legitimately, they execute their according transitions, if applicable. A final step is responsible for all approved controllable events being realized appropriately, while rejected events must not take effect. The different methods for that are subsumed as

signal generator in the scheme. Apparently, the most intuitive and easiest solution to this is resetting all events which correspond to blocked events to the values they had before the cycle while leaving the remainder of PIO as is. However, some assumptions must be made for this approach to be applicable, which imposes several restrictions on the kinds of requirements and scenarios which can be addressed this way [Timmermanns, 2015]. More details on the realization of blocked and approved events will be given in Section 6.3.

Anyhow, it is not possible for the controller to bypass the supervisor as all produced output values are cached in PIO first. Since the output phase concludes the PLC cycle as a whole, the supervisors are always invoked before any operations are propagated to the hardware.

5.1.4 Nonblockingness

One of the most critical issues of SCT is the exponential state space blow-up when all components of a modular system are composed, cf. Section 3.1.1. However, dealing with the behavior of the entire system is not always necessary. The Ramadge-Wonham framework originally defines two main objectives for supervisor synthesis, first to establish controllability for a given specification and second to guarantee nonblockingness.

However, enforcing nonblockingness on a fixed controller implementation is either not possible or at least impracticable for several reasons beyond the computational complexity. First, it must be assumed that none of the controller's actions is expendable, i.e., disabling one of them would probably jeopardize the result. Remember that productivity requirements delimit the minimally necessary behavior of a production goal. Second, since the controller is assumed to be developed without awareness of the supervision, it is probably not able to recognize that one of its actions has been disabled. In return, on the one hand, a controller which reckons being blocked at any time must be implemented very conservatively – not precisely the use case addressed above. On the other hand, it would need to provide an alternative recipe for that case to still guarantee productivity and nonblockingness.

A controller that holds a set of spare actions for every thinkable combination of blocked operations is obviously not possible. Hence, a controller would need to recompute its procedure dynamically in order to work around the disabled operation towards its control goal. Due to the large variety of combinations of disabled events, this corresponds to an on-the-fly controller synthesis, which would be even more challenging than controller synthesis ex-ante. It is questionable whether supervision would still be required if that problem was solved satisfactorily.

Assume that a controller exists which provides a manageable number of spare recipes such that, in case one of them is disabled, another could be chosen. Then synthesis would need to estimate the impact of blocking a certain action to still guarantee nonblockingness. In other words, a model of the controller in the shape of a DES would be necessary, a task which is highly non-trivial to achieve automatically. Still, the designer of the controller would need to purposely implement those spare methods for the sake of providing alternatives for the supervisor.

This approach concentrates on enforcing the specifications of top-priority side-conditions like safety. If one of these is imperiled to be violated, e.g., by a cascade of uncontrollable

events, supervision intervenes. In that situation, blocking is tolerated. Note that this only affects controllers which are erroneous in the sense that they disrespect such a requirement. Proper controls, however, would not block.

From an applicational perspective, disregarding nonblockingness avoids computational problems such as the state space blow-up, which is not sufficiently solved yet. Besides, it allows for more efficient synthesis methods which focus on the avoidance of critical states using narrower criteria than it would be necessary for nonblocking.

5.1.5 Classification of the Approach

Consider a set of critical safety requirements that has been modeled successfully and synthesized into a supervisor. When applied onto the controller as shown in Figure 5.2, it ensures that the latter will not violate these requirements as long as the framework runs. In the light of safety and reliability as shortly introduced in Section 2.3.5, this principally makes the control system *fail safe* regarding implementation errors in the controller.

If the safety requirements have also been taken into account during controller development, they are realized *redundantly* in the sense that two different representations of the requirement have come to implementation – the controller and the supervisor. Only if both are erroneous, the system would fail against these requirements. Therefore, the approach can be classified as a measure of *diverse redundancy*. Note that, as long as the framework is run on the same device as the controller, hardware failure is not covered. This could be changed in future versions using independent external devices for supervision.

While safety is improved, reliability stays unaltered. Apparently, supervision cannot replace the controller as it does not realize productivity requirements.

Amongst the formal methods, the presented one is situated between *synthesis* and *runtime verification*. On the one hand, supervisors are synthesized from specifications and plant automata in order to achieve controllability in a maximally permissive way. On the other hand is the result, which is correct by construction, used to guarantee compliance with that specification during runtime.

5.1.6 Further Techniques

In the following, some additional techniques are sketched briefly. A detailed motivation and description of these can be found in the succeeding sections. Control scenarios frequently involve real-time requirements. Safety specifications in particular often include hard deadlines, e.g., for a certain reaction. Unfortunately, SCT is not capable of dealing with timing constraints at all. The timed-DES formalism by Brandin and Wonham presented in Section 3.1.3 tackles this issue. However, it usually comes with a significant growth of the state space and often even requires more complicated models from the user since all events need to be related to tick events.

In this work, enforceable events are utilized directly. Conceptually, they roughly work as follows: Whenever a supervisor enters a state which has an outbound enforced transition, this transition has to be taken immediately. This stands in contrast to the behavior of a

supervisor derived from a timed DES, which would actively trigger a transition no earlier than ultimately necessary. The reason is that synthesis yields a maximally permissive supervisor and therefore prohibits the *tick* event, which represents the passage of time, only in states where waiting any longer would be illegal, cf. Section 3.1.3.

In the presented approach, an action is only enforced by synthesis if it preempts an imminent, undesired and uncontrollable incident. Since enforceable events are not reduced onto the two classical event classes, an explicit time model is not required.

An alternative method, based on a simple cyclic re-evaluation of events, also allows the supervisor to react on critical states by prohibiting the controller to "continue its current operation". In order to avoid problematic safety reactions being triggered too early, it is necessary to measure the time that has passed since the occurrence of a certain event and thus delay an action or a prohibition.

In order to enhance the process and decrease the effort of modeling, conditional transitions are introduced. These allow to relate transitions and prohibitions to the currently active state of other components. By that, the need to model paths redundantly, particularly in specifications and plant models, is mostly eliminated.

5.2 Related Approaches

Besides the academic tools that exist in the SCT area (cf. Section 3.2.1), there are approaches from other domains which address a similar use case but apply different methods and formalisms.

Software-based safety measures are not a new concept but have already existed for several decades. However, from simple, low-level mechanisms as interlocks, they have evolved towards advanced, model-based safety systems. By adding the SCT-based framework on top of safety models, this approach goes one step further and, by that, lifts interlocks towards specifications which are uncontrollable in the first place using their SCSL.

Balemi's work (see Section 3.2.2) already included a basic version of a supervision framework similar to the one presented in this chapter. Although it was tailored to the presented case study, it already offered a certain degree of flexibility and even a basic graphical user interface. However, only the supervision of manual operation was supported. Monitoring manually implemented controllers was not part of his research nor was the integration to the cyclic setting of PLCs.

One recent contribution which should be mentioned in this context has been presented by Riera et al. [92]. Similarly as described above, a framework is installed on a PLC in order to monitor a given controller and guarantee safety during runtime. The specifications are provided in terms of plain logic formulae over the signals. It seems that a model-based forecast of unavoidable violations as performed by the SCT, is not supported by this approach.

Another distantly related work has been presented by Prati et al. [86]. It takes specifications in the shape of Petri nets and transforms them into systematic test cases reflecting the requirements defined by the cause-and-effect matrix of the process.

Schuh and Lunze use *deterministic I/O automata* to realize fault-tolerant controllers [97, 98, 99]. These automata can be understood as a hybrid of DES and reactive controller models. On the one hand they represent the physical capabilities of the plant using states and transitions. On the other hand does every transition of the controller assign an output action to every transition, i.e., every input is reactively associated with an output, depending on the current state. Instead of deriving an entire controller from the models, the focus of this work is set to adapting an existing one in case of a runtime fault. When a fault is detected, a model of the faulty plant is computed from the diagnostic result of the detector. Based on the existing controller and that model, an alternative controller is derived.

5.3 Modeling Concept

The strength of applying formal methods is that they produce results that are correct by construction. Therefore, it is essential that also the models which serve as input for the algorithms have clear and unambiguous semantics. Nonetheless should the manual effort in providing these be as little as possible when the methods come to application.

5.3.1 Automata

The original Ramadge and Wonham framework already uses automata to describe the behavior of generator and specification [87]. Finite-memory supervisors are often represented by automata too [23].

Although these automata have very different semantics and serve different purposes, they are usually defined the same way. The meaning of a transition, or its absence, thus always depends on the current role of the automaton but also on the context and the progress of synthesis. In the following, four different types of automata are introduced to express different kinds of logic statements. These are *specifications, supervisors, plant models* and *synthesis* automata.

Specifications Through the eyes of a user, specifications are the most important type of automaton as they serve to express the actual requirements which shall be enforced. Side conditions are often formulated in a prohibitive way. In consequence, all actions and events which do not contradict the requirement are implicitly allowed. Specification automata work analogously [43]. Instead of defining the entire legal behavior over a given alphabet, as common in SCT, they impose explicit event *prohibitions*. This is similar to the notion of *bad states* as used in some other approaches, except the bad states themselves are not part of the model. Instead, the events which would lead to these states are forbidden straightforwardly. Every event which is not prohibited in a state is principally allowed to occur.

Specifications can consist of multiple states. When a transition for the event e exists from one state q to another one q', the current state will be changed from q to q', once e occurs. When there is no such transition, the current state is kept, i.e., the specification ignores the event.

(a) Specification automaton (b) Equivalent SCT specification, $\Sigma = \{\mathtt{a},\mathtt{b},\mathtt{c}\}$

Figure 5.3: Exemplary specification

Example 5.1. Figure 5.3a shows an exemplary specification. The initial state 1, for instance, has an outbound a-transition leading to state 2. a and c are permitted in all states, although they do not always cause a state change. The only prohibition is the forbidden b-event in the state 4, denoted by a rectangular tag next to the state. 5.3b shows the equivalent specification for the Ramadge-Wonham framework. △

Obviously, both specification types offer the same expressiveness and can easily be transformed into each other. Nevertheless, the prohibitive shape in Figure 5.3a has some advantages in the application. First, an explicitly defined alphabet is not required for this type of automaton. Instead, it depends on the particular states which events are relevant in a certain situation. For an event which does neither define an outgoing transition nor is prohibited in the current state, it is regardless whether it is considered somewhere else in the automaton, and hence part of the alphabet, or not. Second, prohibitions are more apparent as they can clearly be seen and easily be added to arbitrary states. The third beneficial aspect regards prohibitions in synthesis automata and thus is discussed below.

Supervisors Supervisors are very closely related to specifications. They have the same semantics and structure, except that they only prohibit controllable events. Hence, every supervisor can be considered as a specification whereas the opposite is not true in general. Thus, the automaton depicted in Figure 5.3a would also be a valid supervisor if b is controllable. Supervisors can be synthesized from specifications using the algorithms of SCT. Thanks to their controllability in the sense of [87, 23], they do not only define *what is legal and what is not* but include *how that can be achieved*.

Plant Models Plant models define the physical limitations of certain parts of the plant. In contrast to specifications and supervisors, they have an explicit alphabet containing all events of which the model makes statements about.

Plant models take the place of the generator in traditional SCT. However, they have a slightly different connotation. A generator is intended to represent the entire behavior of the uncontrolled system. Correspondingly, the SCSL of the closed loop of generator and supervisor covers all eligible behaviors. In the presented apporach, plant models instead only serve the purpose of increasing the permissiveness of synthesized supervisors.

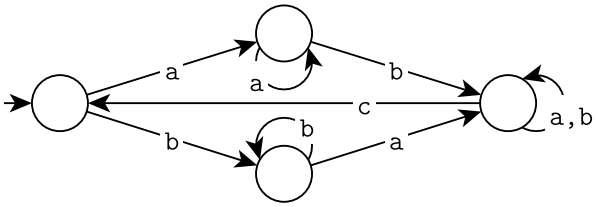


Figure 5.4: Simple Plant, $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$

There are indeed cases in which a supervisor cannot even be derived without the necessary knowledge about the plant, meaning that safe operation of the system would be considered impossible. However, the bare existence of a supervisor does not give any guarantee for the productivity requirements being achievable. It is possible that the SCSL is nonempty but supervision is still too restrictive for productivity without further plant models. Hence, the step from an empty supervisor to a nonempty one is considered equivalently to an increase of permissiveness.

Example 5.2. Figure 5.4 shows an example of a simple plant model. It expresses the fact that the event `c` can only occur if both `a` and `b` happened before since `c`'s last occurrence as well as initially. It does not capture any limitations of `a` and `b` since every state has outbound transitions on these two events. The same holds for all remaining events e that are not in the alphabet, $e \notin \Sigma$, and which the plant model thus does not make any claims about. △

Plant Contract It is very important that a plant model is *complete* with respect to its alphabet. Therefore, the following contract forms the basis for the correctness of synthesized supervisors.

> *In a plant model, for every controllable and uncontrollable event in the alphabet, a transition must be provided in every state where it is possible to occur.*

This usually involves self-loops, i.e., transitions from a state to itself. When a given state has no outbound transition for a certain event in the alphabet, it is considered to be physically impossible to occur in this situation. A plant model may, however, over-approximate when events can arise, i.e., it may define a transition on $e \in \Sigma$ even in states where it is or might physically not possible. This must be regarded particularly if it depends on further components whether or not an event e can take place. If the granularity of information that one plant automaton has is insufficient to definitely *exclude* the occurrence of e in a state q, an e-transition must be established at q. Adding too many transitions to a plant can result in worse permissiveness or even the nonexistence of a supervisor, whereas too less of them can cause an incorrect and thus unsafe one.

Synthesis Automaton The supervisor synthesis algorithm which is utilized by Ramadge and Wonham in their original publication [87], and can be found in [23], operates on one

specification and one generator at the same time. It further requires a mapping between the states of these automata to determine which ones are controllably safe and which are not.

In modular approaches, this association is not implicitly given anymore as specifications and plant automata may have extremely different structures. Several solutions exist for that issue. The *compositional synthesis* approach, which is implemented in Supremica [35] for instance, introduces an artificial blocking state, i.e., a sink. The algorithm inserts a transition to that sink for every illegal event. This way, the problem of safety is transformed into a nonblockingness problem on plant automata. A state mapping is no longer required and the automata can simply be composed with each other.

Since nonblockingness is disregarded, this technique would not work. However, thanks to the explicitly modeled prohibitions, it is still possible to join specifications and plants into one compact representation. In case of the monolithic method, this composition is done before invoking the actual synthesis, which then is carried out on the composite and operates purely destructive[2]. Thus, an automaton type for the intermediate result is required. It needs to cover the aspects of specifications and plant automata, i.e., which events are (im)possible and which are (il)legal. As it represents an intermediate artifact for synthesis, it is called *synthesis automaton*. It has transitions, prohibitions and a *plant alphabet*, containing all events of which the automaton has "plant knowledge", i.e., for which the plant contract holds. Note that the automaton may have transitions and prohibitions of events which are not contained in that alphabet. These have the same semantics as in specifications, i.e., a transition changes the state but it is not known whether an event is physically possible or not. The formal definition and semantics of synthesis automata are given in Section 5.8.1.

A synthesis automaton which is free of uncontrollable prohibitions can be converted to a supervisor by simply discarding its alphabet.

5.3.2 Events

Events represent everything which can in principle change the physical state of the plant or the logic state of a requirement. This includes actions from the PLC as well as events on the plant which are measured through sensors. The presented event scheme of the tool has been developed in the context of the thesis project [Gatto, 2016]. Parts of it have already been published in [41]. It involves a significant variety of different kinds of events, which are distinguished by means of the following criteria: The *event type*, the *trigger class* and the *action class*. A graphical overview about the event classification scheme can be found in Figure 5.5.

Event Types The key distinction of events is by their type. The classical partitioning into controllable and uncontrollable events is extended by a third type, the enforceable events. Controllable events originate from the controller and must be approved by all supervisors whereas uncontrollable events represent things that happen on the plant. Enforceable events

[2] If preemption is disabled, synthesis does only delete but never insert or change states and transitions.

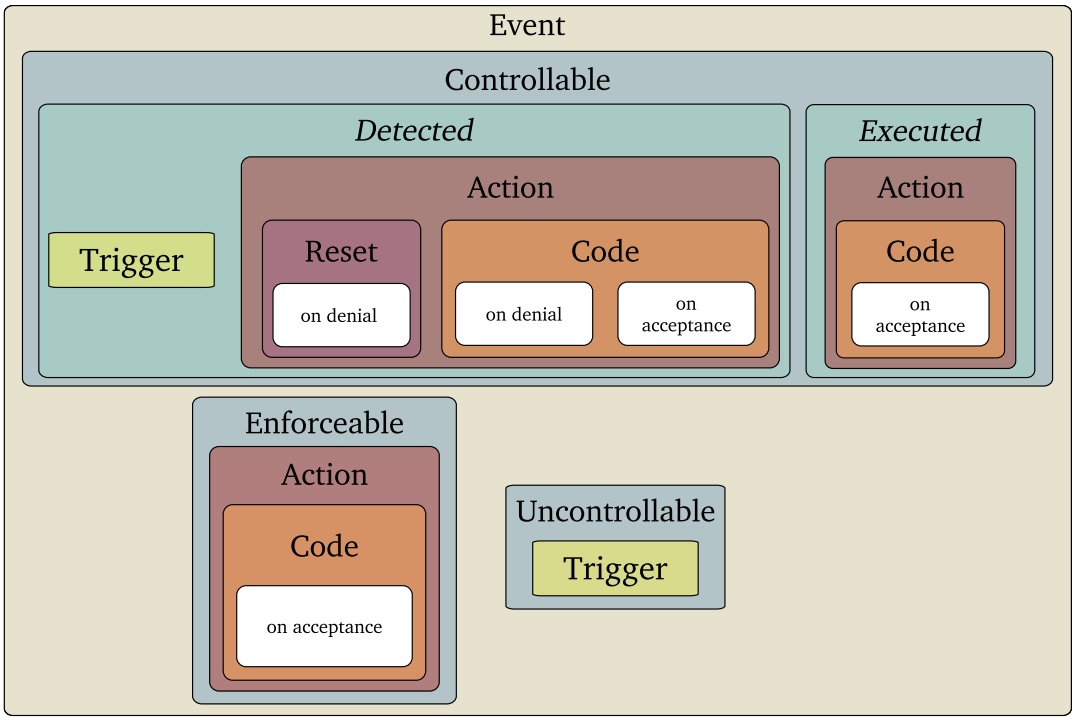


Figure 5.5: Types of events, figure based on [Gatto, 2016]

can be triggered by a supervisor independently from the controller and other supervisors to prevent uncontrollable events. This is explained in Section 5.5 in detail.

Trigger Classes In the standard case, controllable events are detected after the controller has finished its cycle by analyzing its output. When the latter changes in a specific way, an event has occurred. Alternatively, the controller can be allowed to execute events explicitly. These two modes of operation will in the following be referred to as *event detection* and *event execution*.

In contrast to the controllable ones, uncontrollable events always need to be detected as it is not possible for the plant to communicate with the supervisor directly in the considered setting. Enforceable events in return are always triggered by the supervisor and thus can be counted to the executed events as well.

Action Classes Controllable and enforceable events need to be assigned a certain kind of action which is either carried out when they are approved or when they are blocked. Since the ways of how events influence a system can be manifold, it depends on the situation and the single case which kind of action suits best. There are three action classes called *reset on denial, code on acceptance* and *code on denial* available for detected controllable events.

The first of these appears most intuitive. The relevant output values are captured before the controller execution. If an event is disapproved by a supervisor, the corresponding parts of PIO can be reset to their original values before being applied to the hardware. Controllers frequently work on numerical values and often involve logics which go beyond

the scope of the provided regular DES. Also, there can be multiple events manipulating the same outputs. In both cases, it would be too limiting to only allow to reset *all* outputs that an event potentially writes to [Timmermanns, 2015].

The action class *code on denial* allows the user to specify one or several instructions which are executed when the event is blocked. This way, she can manually specify how the actions which correspond to that particular event can best be reversed. *Code on acceptance* works similarly except the code is executed when the event was approved. For executable controllable events and enforceable events, *code on acceptance* is mandatory as they are always explicitly invoked, in the former case by the controller, in the latter by the supervisor itself. It is allowed to use this action class for detected controllable events too, e.g., if the controller writes a value to a cache outside PIO, which shall be copied to the latter if the event is authorized.

Section 6.3.1 illuminates how the different trigger and action classes are technically realized.

5.4 Basic Operations

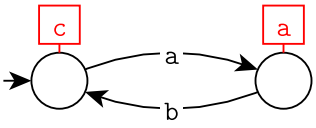
After the user has provided all relevant specifications, plant models and events, the supervisors can be synthesized. Within the scope of this thesis, two different algorithms are utilized and discussed to that end, the classical monolithic one and a new incremental method. For the monolithic method, the user first has to compose each specification with all relevant plant models, while the incremental algorithm adds available plant information automatically.

An exhaustive and formally founded description of the algorithms is postponed to Section 5.8.

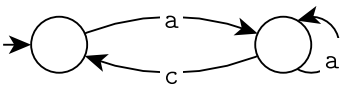
5.4.1 Composition

The rules of composing two automata depends on their respective types and so does the outcome. The composition of two specifications again yields a specification. Analogously can two plants be composed to another plant. In both cases, the order does not matter as the operations are commutative. Note that it rarely makes sense to compose two specifications as the resulting supervisor is very likely to be larger than two separately synthesized supervisors. An exception is the use of enforceable events which are potentially prohibited by another specification. To detect such conflicts, the composition of both is necessary.

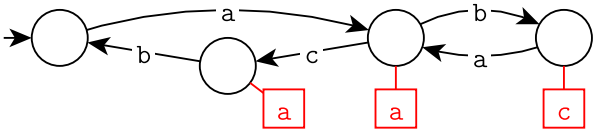
When a specification is composed with a plant model, the result is a synthesis automaton. It reflects all prohibitions of the specification as well as the limitations of the plant. Conceptually, the specification is refined on the information given by the plant model in the sense that the resulting automaton is able to distinguish between prohibitions of events which are possible and those which are impossible or can occur only under certain circumstances. Hence, this process is also referred to as the plant model being *applied onto* the specification. After that, the resulting synthesis automaton can be composed with more plant models to further refine its permissiveness. However, it is illegal to compose a synthesis automaton



(a) Specification



(b) Plant, $\Sigma = \{a, c\}$



(c) Composition of both automata, $\Sigma^{\Pi} = \{a, c\}$

Figure 5.6: Composition of a simple specification and plant model

with another specification as this can cause inconsistent models and hence lead to incorrect results.

Example 5.3. Figure 5.6 shows a simple specification (alphabetless), a plant model featuring the two events a and c, and their composition. The permissiveness alphabet Σ^{Π} of the latter also involves only a and c as it is not known yet when b is possible. △

5.4.2 Monolithic Synthesis

Monolithic Synthesis is applicable onto specifications and synthesis automata. It establishes controllability given on the information contained in the automaton, i.e., the more plant models are included, the more permissive will the supervisor be. It is up to the user which ones she wants to add to the composition. Adding all available plants always guarantees the *maximally achievable permissiveness* although the same can often be accomplished more efficiently by using only a subset of these. Note that in the literature, *monolithic synthesis* usually addresses the former case where the entire knowledge about the model is taken into account. In the context of this work, it instead means that a supervisor is derived from an already computed composition using all contained but no further information.

The non-preemptive variant of this synthesis works analogously to the original SCSL algorithm given in [23] except it operates on a single synthesis automaton instead of separate specification and plant automata.

5.4.3 Incremental Synthesis

This technique interleaves synthesis with the required compositions. It is invoked on either a single or a composite specification and lazily adds the necessary plant models from the project on demand. Using this technique, a user can obtain a supervisor from a specification without bothering about which plant models are relevant. Thanks to that it is very easy to use. Nonetheless, does it guarantee a maximally permissive result and automatically

neglects irrelevant plants parts. Especially larger projects with several specifications for different, loosely connected components can significantly benefit from that when supervisors are incrementally synthesized from the specifications independently as the algorithm automatically picks the relevant models.

The key idea of incremental synthesis is to compute the composition of specifications and plants only in situations where supervision can potentially benefit from the information of the plant. In contrast to the modular synthesis algorithm by Åkesson et al. [3] where the applied necessary criterion for that is the closure of event sharing[3], incremental synthesis only adds a plant model if a state would need to be isolated by synthesis in the very next step. The results are still maximally permissive, however the models can be significantly smaller. A detailed explanation of the criterion and a formal definition of the algorithm, including a comparison to Åkesson's method, are postponed to Section 5.8.3.

5.4.4 Minimization

Intermediate results can be trimmed by removing unreachable states. This corresponds to the *accessible* function as defined and used in the literature, e.g. [23].

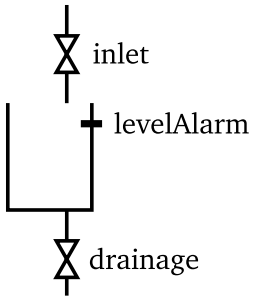
Further minimization is possible by adapting the common method [51, 23] to the automaton types described above. The result is a worklist algorithm that tries to distinguish states by their prohibitions and their adjacent successor states. First, all states are assumed to be equivalent and added to the worklist. During the procedure, states are compared pair-wise. Whenever a difference is found, the pair is marked as *distinguishable* and the predecessors of the respective states are added to the worklist to be checked again. Once the worklist is empty, the algorithm terminates. All non-distinguishable state pairs are fused.

5.5 Preemption

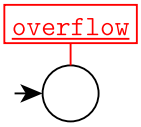
Real plants frequently involve *unstable states*. Here, "unstable" addresses a situation in which the system may not reside arbitrarily long but which instead must be left within a certain amount of time. Often, that is not only a matter of proper productivity but also of safety.

In practice, real-time hardware and software allow to safely and legally operate systems which involve unstable states. Note that this is not an issue of speed or computational performance. Real-time requirements sometimes allow reaction times in the magnitude of several seconds. It must be guaranteed though that the reaction can by no means be delayed any longer, regardless of the circumstances. Controllers for hard real-time requirements are widely accepted in practice and legally acknowledged. Neglecting this kind of reactive control paradigm would result in unnecessarily narrow restrictions to the system. In the following, the term *preemption* addresses the anticipation of a, typically undesired, incident by a timely reaction.

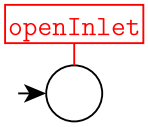
[3] The smallest set of automata where each of them shares an event with the specification or with another automaton in that set.



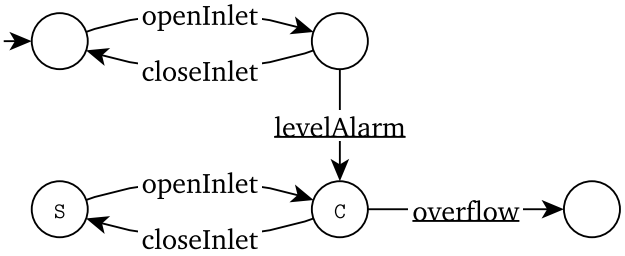
(a) P&I diagram (figure taken from [41])



(b) Specification



(c) Conventionally synthesized supervisor



(d) Plant model, $\Sigma = \{\texttt{openInlet}, \texttt{closeInlet}, \underline{\texttt{levelAlarm}}, \underline{\texttt{overflow}}\}$

Figure 5.7: Motivating Example. Uncontrollable events are underlined.

5.5.1 Limitations of Classic SCT

The supervisory control theory conservatively avoids unstable states that can lead to a violation of the specified requirements. In particular, this rules out any situations which are unstable with respect to uncontrollable events. When the SCT is used for the supervision of practical applications, this can be a significant limitation. This shall be illustrated by an introductory example.

Example 5.4. Consider the physical plant sketched in Figure 5.7a. It shows a tank, equipped with a level sensor, one inlet and one drainage valve. The inlet can be opened and closed by the controller, represented by the controllable events `openInlet` and `closeInlet`. As soon as the liquid inside the tank reaches the level sensor, the uncontrollable event `levelAlarm` is detected. The level sensor has the purpose to indicate when the inlet must be closed to prevent the tank from overflowing. Thus, a manually implemented safety system would enforce closing the corresponding valve as soon as `levelAlarm` arises.

Consider the specification in Figure 5.7b. It globally prohibits an `overflow`. Since that is uncontrollable, synthesis requires an additional plant model to identify the situations in which `overflow` can actually occur. It is shown in Figure 5.7d. Based on these automata, conventional SCT would yield a supervisor which disables `openInlet` in the initial state. The reason is that it represents the last controllable event before the undesired `overflow` is imminent. This supervisor is obviously able to guarantee safety. Nonetheless, it is far more restrictive than the manually implemented safety system mentioned above. It even

makes the entire plant unusable in this case as the tank may never be filled under this kind of supervision. △

The above example illustrates the limitations of classic SCT, which affect its applicability already in this simple scenario. In Section 4.2, it was already mentioned that SCT prunes the reachable state space and its corresponding language down to its controllably safe and, in some cases, nonblocking parts. This seems apparent since obviously neither a supervisor nor any other controller can add behaviors to a system which are not considered possible to occur. In the application, however, the decisions about if and when controllable events are triggered, is entirely made by the controller under supervision. For that reason, the supervisor would not even be able to *constructively* influence the language of the entire closed-loop system *inside* the boundaries about what is possible on the plant. Instead, it can only grant or block the controller's actions. This problem does not arise in those case studies which deal with the CHOICE problem (cf. Section 4.4.1), e.g., by randomly executing one of the admissible events in every controller/supervisor iteration. The core difference to these approaches is that here a control loop of three participants is considered (cf. Figure 5.1). Thus, the controller makes its choices independently from the supervisor and hence cannot be influenced or even restricted down to execute an appropriate action *in a specific moment*.

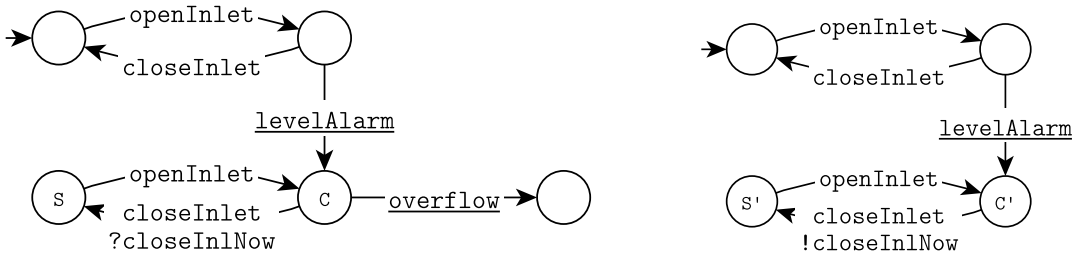
What the methods of classic SCT do allow in this setting is preventing undesired incidents in advance by early restricting the controller in its operation. These kind of safety measures are called *interlocks* as they lock a certain action until it can safely be executed. Indeed, the presented method performs well in synthesizing interlocks using non-preemptive methods, even for complex scenarios involving many different events and states to consider. However, in order to synthesize safety measures that have equivalent permissiveness to the manually implementable one sketched above, either the synthesis method or the notion of events must be altered. The first option, which addresses synthesis of preemptive supervisors, is introduced in the following, while the second will be discussed in Section 5.6.

Note that the plant automaton shown in Figure 5.7d can actually be transferred onto several scenarios, such as a vehicle striving towards an obstacle (replace `openInlet` by `start`, `closeInlet` by `stop`, `levelAlarm` by `obstacleAhead` and `overflow` by `crash`) or the doors of an elevator (replace `openInlet` by `closeDoor`, `closeInlet` by `openDoor`, `levelAlarm` by `lightBarrierInterrupted` and `overflow` by `objectJammed`). In all cases, the same issue arises: The synthesized interlocks disallow any operation of the respective system due to the unstable state.

5.5.2 Enforceable Events

In the past, there have been numerous contributions which deal with augmenting SCT by enforceable[4] events. The earliest among these is the work by Golaszewski and Ramadge [44]. They assume that an enforced event precludes all other events in that state. The conception in this section roughly follows this principle in the sense that enforced events can

[4] Often referred to as *forcible events*



(a) Alternative plant model, $\Sigma = \{$openInlet, closeInlet, levelAlarm, overflow, ?closeInlNow$\}$

(b) Preemptive supervisor.

Figure 5.8: Example with preemption

be used to preempt undesired uncontrollable events in order to obtain a more permissive, yet safe, supervision.

Consider again the manually realized safety system for the tank in Figure 5.7. When safety shall be guaranteed using this mechanism, two constraints need be imposed. First, the level sensor must be installed in a way that, respecting the maximal inflow, the valve can still be closed before the overflow occurs. Second, the safety system must be reliably fast enough to guarantee a sufficiently instant response. Note that the latter aspect is indeed a real-time requirement to the hard- and, if applicable, software of the system. An explicit time model or time measuring capabilities are not required though, as the reaction is to be triggered as soon as possible. Preemption follows the exact same principle.

Example 5.5. Consider the alternative plant model in Figure 5.8 for the scenario from Example 5.4. It has been supplemented by a transition carrying the event `?closeInlNow`. The question mark indicates the enforceability of the event. The synthesis routine detects that the unsafe state `C` can be actively left using this transition and activates/enforces it. The resulting supervisor, shown in Figure 5.8b, now allows the `openInlet` transition and triggers `?closeInlNow` immediately after `levelAlarm` has been detected. The '?' in the label of the enforceable transition has been replaced by a '!' to emphasize the difference between a yet enforceable and an already enforced transition. This supervisor is obviously more permissive than the conservative one in Figure 5.7c and shows indeed the same behavior as the manual safety system. Note that an additional specification which disallows `openInlet` after `?closeInlNow` could serve to synthesize an additional interlock. △

Preemption Contract When preemptive synthesis is supposed to result in a safe system, the same assumptions must hold as for the manually realized system mentioned above. While the second constraint (that the system is sufficiently fast) rather affects the hardware and the real-time properties of the framework implementation, it must still be guaranteed that `levelAlarm` (or, for the analogue examples, `obstacleAhead` and `lightBarrierInterruped`) occurs early enough to successfully intervene. Without a time model, it is not possible to embed these aspects into the automata. Thus, it is up to

the user to decide whether an enforced reaction that is triggered once a state is entered, would be timely or not. The following contract, in the following referred to as *preemption contract*, is imposed between the user and preemptive synthesis:

1. *If an enforceable transition is "offered", synthesis assumes that it can successfully preempt every uncontrollable event leaving the same state.*
2. *An enforceable event which is modeled possible by all plants (or not declared impossible by any plant) in a certain logical state, can indeed be executed on the plant.*

For the sake of stronger permissiveness, preemption is always favored over isolating the state. The latter option is only chosen if there is no enforceable transition or all of them lead to another unsafe state. The second item of the contract becomes vital in the context of plant models being interpreted as an over-approximation of the actually possible behavior, as claimed by the *plant contract* in Section 5.4. For enforceable events the reversed semantics are required, i.e., the enforced events must form an under-approximation of the actually enforceable actions. This is necessary to make sure that every enforced action can indeed be executed when it is expected.

Example 5.6. Consider a model for a chemical plant. Not all physical aspects need to be covered by the plant model (over-approximation). Let e_s be an event representing a sensor reporting a certain threshold. If e_s is part of a plant model's alphabet then there must be an outgoing e_s transition at every state where e_s is physically possible so that synthesis is aware of when this threshold can be reached in principle. Let e_f represent the activation of an emergency cooling mechanism. Then the model must not contain any e_f transitions in states where enabling that mechanism would not be possible (under-approximation). △

When several enforceable transitions exist, one of them is picked randomly. Remember that solving optimization problems is beyond the scope of the presented approach. Hence, it is not able to determine which choice amongst several transitions would be the "best" one. Instead, it is assumed that all enforceable transitions that leave the problematic unstable state equivalently suitable. The significant difference to the approaches which solve CHOICE by randomness is the fact that randomized decisions are applied only for the sake of hazard prevention if more than one enforceable event is safely possible whereas those use them to generate productivity.

5.5.3 Relation to Other Formalisms

Enforcement in Timed DES In his monograph [117], Wonham describes how controllable events can theoretically be enforced on timed DES (see Section 3.1.3). The key is the controllable *tick* event, which can be disabled to enforce an immediate action. This way, the resulting closed-loop language does not tolerate the passage of time until *tick* is admissible again.

To enforce a specific event in a given state, it is necessary that all other events are disabled in that state [44]. Thus, the specification and plant models must set the circumstances

that finally cause the remaining events being disabled. There are methods, e.g., [29], to translate untimed DES with enforced transitions to timed DES that use the Wonham method and involve only the two classical event types.

Although Wonham's method allows an elegant integration of event enforcement into the classic framework of SCT and indeed leads to languages that reflect the desired behavior, it is problematic when supervisors shall be executed during runtime due to several reasons. It is obviously not possible to physically stop time. Hence, it would be necessary that the respective part of the control loop which is responsible for the remaining events interprets the disabled *tick* event properly and instantly executes an event. In the considered application the controller triggers all controllable events (or performs the actions that lead to their detection). However, the reason why supervision is installed on the system is that it must be assumed that the latter is faulty and omits the necessary reactions. As a consequence, if the event is crucial and supervision shall guarantee that it is applied to the plant reliably, it must be the supervisor to execute it. Hence, in this specific scenario it makes sense to stick to three event classes that allow a distinguished handling and formal representation of the events from three different participants.

When controller supervision shall comprehensively realize timing coherences, there is obviously no way around a modeling paradigm that provides full expressiveness on these aspects, such as timed DES. Enforceable transitions instead offer a straightforward method to introduce definite actions which must be executed immediately, even if the controller does not trigger them. Therefore, they combine the advantages of easier modeling, smaller models and a more direct relation to the runtime problem to the price of having no complete time model, which would allow to consider arbitrary dependencies.

I/O Automata This automaton type has been utilized, e.g., by Balemi [9] and by Schuh and Lunze [98]. Regarding the controller response, I/O automata have a reactive design: Instead of passively monitoring the plant and disabling some transitions, the controller enforces one specific action after every sensor event. This direct association between observations and reactions is not provided by Ramadge and Wonham's SCT, neither is the concept followed here. However, since an enforced transition in a supervisor is executed as soon as the corresponding source state is entered, the combination of an uncontrollable transition and a succeeding enforced one can be seen as one logic I/O transition because the supervisor never resides in the intermediate state between both. Still, the goal is not to realize reactive controllers. Enforceable events shall provide a way for the supervisor to actively leave critical states and thus allow the controller to enter them in general. Nonetheless, should these events not be misused to implement productivity requirements or longer sequences as indeed several problems can arise when enforceable events are used too exhaustively. This will be discussed in Section 6.4.2.

5.5.4 Which Events Shall be Preempted?

The basic idea of preemption is that undesired uncontrollable events can be precluded by actions which are enforced timely by the supervisor. This, however, leaves open the

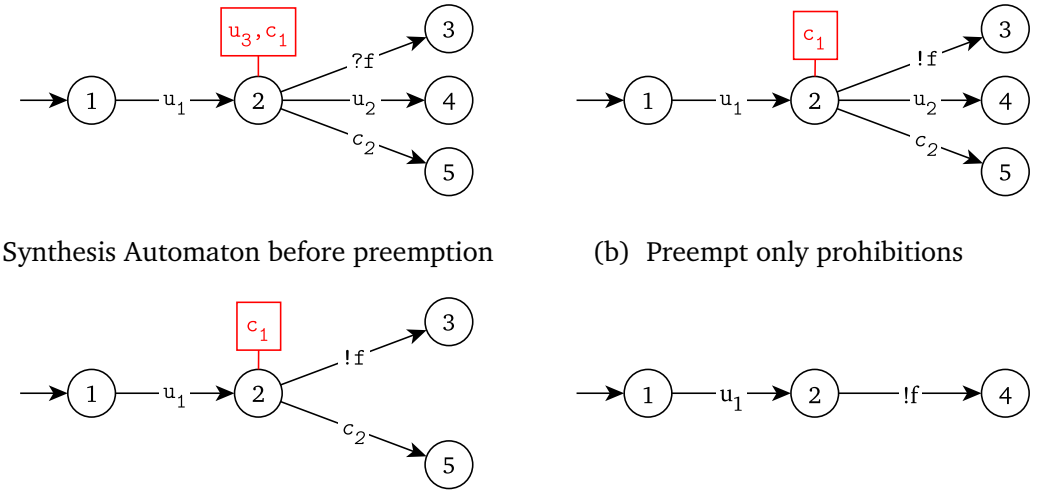
question whether these enforced events preempt the remaining events of the same states as well. Most approaches follow Golaszewski and Ramadge [44] and claim that either one enforced event is enabled or all uncontrollable events. In their work on event enforcement, Diekmann and Weidemann [29] relaxed the control law (Equation 2.1) accordingly.

When an existing controller implementation shall be supervised by the framework, this interpretation of enforcement would collide with the intention of the presented approach. In Section 5.1 it was mentioned that a correctly operating controller should be distorted as little as possible. Hence, the supervisor should only enforce a reaction if the controller misses to do so. Still, it is assumed that a deliberate reaction to a sensor event, triggered by the controller, is likely to be more prudential, differentiated and thus preferable compared to an (emergency) action by the supervisor.

Consider the preemptive supervisor in Figure 5.8b. The enforced event `!closeInlNow` would not only preempt the uncontrollable `overflow` but also the controllable event `closeInlet`. However, `closeInlet` is the proper reaction intended and expected by the control designer. Thus, `!closeInlNow` shall be executed if and only if that action is missing. Adopting Diekmann and Weidemann's control law here would defeat this purpose. Note that the controller is executed before the supervisor but within the same PLC cycle (cf. Figure 5.1b). As a consequence, both receive the information about the tank being full (`levelAlarm`) at the same time. To not interfere with a correct controller implementation it is necessary to *not* preempt controllable events, given that these are not prohibited due to other aspects of the specification. The formalization of preemption, given in Section 5.8.5, follows this principle and indeed yields the very supervisor shown in Figure 5.8b where `closeInlet` is not preempted. That means that `!closeInlNow` is only executed in `C'` during runtime if `closeInlet` has not been detected.

For uncontrollable events, the situation is a bit different. Based on the preemption contract, they are considered as successfully preempted in case a transition is enforced. A permissive method would only discard uncontrollable prohibitions (those which shall be preempted) while leaving uncontrollable transitions untouched. This can be done without risk as it can only enhance the supervisor's preciseness in tracking the plant. Thus, if a sensor event is immediately followed by another uncontrollable event (e.g. a safety reaction implemented in hardware), the supervisor can track both events before a reaction is enforced if that is still necessary at all. Unfortunately, although this notion can be implemented and also formally modeled straightforwardly, it turns out very unhandy when proving the correctness of preemption as it requires to treat events differently that are or will be prohibited at any time during execution. Thus, the definition of the algorithm on the one hand and safety itself on the other hand would depend on each other, making the proof of the algorithm's safety significantly more cumbersome. For these reasons, the formal model and the tool implementation presented in the following chapter diverge in this aspect.

Example 5.7. Figure 5.9 shall illustrate this. $\mathtt{u}_1$ represents a sensor, $\mathtt{u}_3$ an undesired incident and $\mathtt{c}_i$ a controller reaction to $\mathtt{u}_1$ and $\mathtt{u}_2$ a hardware reaction. Part 5.9a shows the composition of a specification prohibiting $\mathtt{u}_3$ and a plant which offers `f` to escape from the only state where $\mathtt{u}_3$ can happen before the latter occurs. Synthesis finds the `f` transition,



(a) Synthesis Automaton before preemption

(b) Preempt only prohibitions

(c) Preempt all uncontrollable events

(d) Preemption as suggested by [44, 29]

Figure 5.9: Preemption of further events. u_i uncontrollable, c controllable, f enforceable

enforces it and removes the prohibition of u_3. The result is shown in Part 5.9b. When the sensor is triggered, u_1 is detected. If the controller reacts properly, c_2 would be recognized too, both would be tracked and the supervisor ends in state 5 at the end of the cycle. `f` would not be executed. In case of an immediate hardware reaction, u_1 and u_2 would be detected and the supervisor would stop in state 4[5]. Thus, `f` is only enforced if neither c_2 nor u_2 occur in the same cycle as u_1. The control law suggested by [44] and formulated by [29] would preempt both events in state 2. Thus, `f` would always be enforced and the supervisor would not be able to track either of them, i.e., it would intervene in a correctly working controller. △

When more than one enforceable event is possible at one state and synthesis enforces one of them, the other one is neither explicitly preempted nor triggered. The reason is that, if there are several supervisors, the other event can still be enforced too. When only one supervisor is used, all enforceable events which are not executed can be taken as preempted as no other participant of the control loop will trigger them. Practically, this is not relevant because synthesis explicitly prohibits events which lead to critical states. If these are not uncontrollable, their preemption is not required since they would not be executed (enforceable events) or granted (controllable events) by the supervisor anyway.

5.5.5 Enforcement Cascades

Sequences of multiple subsequent enforceable transitions should be handled with care. Enforcement is meant to provide a technique for preemption but not to replace operational implementations. When several enforced events are cascaded, they will all be executed by

[5] If events are statically ordered in a reasonable way, see Section 6.4

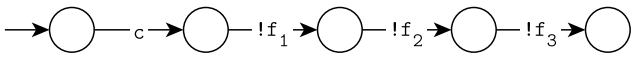


Figure 5.10: Cascade of enforced events

the supervisor within the same PLC cycle. That means that the plant will not recognize the events in their order within the supervisor but simultaneously.

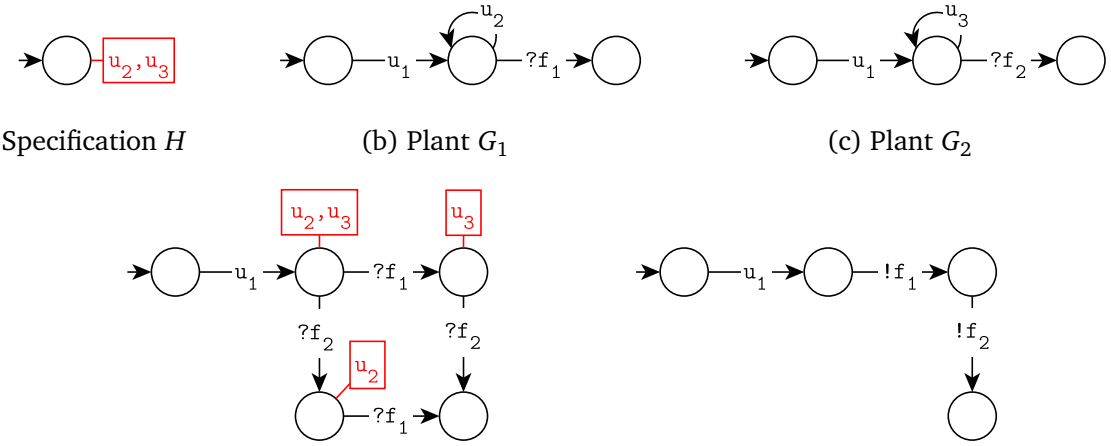
Example 5.8. Given the supervisor shown in Figure 5.10, the framework executes $\mathtt{f}_1, \mathtt{f}_2$ and $\mathtt{f}_3$ in the same cycle. The associated actions are executed after c's action in the given order but still within the same PLC cycle. Hence, the corresponding electric outputs of the PLC are altered at the same time. △

Indeed would it be possible to introduce a special *end-of-cycle* event that could be used to delay the execution of further enforceable events by one cycle each. The cycle time, however, is only assumed to be *sufficiently short* to capture relevant event sequences from the plant but there is no reasonable lower limit to it. As a consequence, even if events were enforced in subsequent cycles, it would not be guaranteed that the plant would be able to properly recognize the events in the correct order and, even more important, physically put them into effect with appropriate timings. The latter is even rather unlikely in practice as actuators are usually significantly slower than the controller operating them. Instead, this matter can be solved with timers, see Section 6.1.3.

If multiple enforced events originate from different requirements, their combination is uncritical and even intended, as shown by the following example.

Example 5.9. Consider the specification and two plant models in Figure 5.11. The two plant models indicate that for each of the prohibited events, $\underline{\mathtt{u}}_2$ and $\underline{\mathtt{u}}_3$, there is one respective enforceable preemption, $?\mathtt{f}_1$ and $?\mathtt{f}_2$. Part 5.11d shows the composition of all three automata. It is clearly visible that both enforceable events need to be executed to effectively preclude both incidents. One of the two possible maximally permissive supervisors is shown in Subfigure 5.11e. The alternative one is analog except it enforces $?\mathtt{f}_2$ first. In either case, both events are enforced within the same cycle. The user could declare priorities for the enforceable events in order to deterministically yield their proper order of execution. This becomes relevant if the events operate on the same variables or memory addresses. However, the user has to make sure that their actions are insensitive with respect to their order, e.g., when writing to the same variables or output addresses. △

More problematic is a cycle of several enforced transitions, which should be avoided by all means. Solving that problem during synthesis would require lookahead mechanisms to find alternative enforceable events. Nonetheless, these cycles are only possible if the user has built them into one model as they cannot be introduced by synthesis or composition. Detecting these cycles can be automated. Resolving them, however, must be a manual task as it affects the semantics of model and specification.



(a) Specification H (b) Plant G_1 (c) Plant G_2

(d) Application of G_1 and G_2 onto H (e) One maximally perm. supervisor

Figure 5.11: Intended cascade of enforced events

5.5.6 General Limitations of Preemption

In the above examples, enforced events immediately cause the unstable, thus hazardous, states to be left. However, assuming that the true hazard has only been averted after a sensor confirms this, for example, that a valve has not only been *instructed* to close, but also the flow has been *measured* to have stopped (uncontrollable sensor event), this is no longer sufficient. In that case, the enforced transition only leads to a state that is still unstable, i.e. contains the forbidden, uncontrollable event, which inevitably leads to the conservative truncation of the state.

Summarized, preemption in untimed DES can only work if the enforceable event itself leads to a safe and stable state. An alternative is to provide and investigate cause-effect models, for instance in the shape of [105, 63, 61, 47], which are able to capture the relation between the enforcement of the actuator on the one hand and the sensor entering the safe state on the other. Unfortunately, even then, success is questionable without a time model, because it needs to be known how long it takes for the measure to take effect and whether this is early enough to reliably prevent undesired incidents. Note that this would involve time constraints on the plant hardware and even the controlled process itself whereas the preemption contract only makes real-time claims to the controller hardware.

5.5.6.1 Fundamental Problem

There is an inherent conflict between the required (deterministic) correctness of the model, which can assert, for example, that a valve is closed after no more than 5 seconds after triggering the actuator, and the fact that a non-controllable event is triggered by a physical sensor. This cannot be solved even with a time model: Either that model is assumed to be correct and the valve always stops the flow before it is too late, or the sensor will wait for feedback for an indefinite period of time and the model is potentially unsafe. In the

first case, the timely effect of the enforced event would be guaranteed and the sensor was dispensable. The enforced event could legitimately lead to the safe state as in the presented examples. In the second case, even a time model would not help.

Requirements such as *"The valve closes regularly within 5 seconds. If this does not happen, an alarm must be triggered."* would be implementable with a time model. However, they can also be accomplished by measuring time outside the model as discussed in Section 6.1.3.

5.6 Cyclic Events

Above it was demonstrated that classic SCT is not able to deal with unstable states without the support of either a time model or enforceable events. An alternative approach is to shift the timing considerations from the model to the execution framework.

The cyclic runtime paradigm of the host PLC causes a natural discretization of time. In every cycle, the controller is able to update its output values based on the information it gets from the hardware inputs. Hence, the resulting reaction time is guaranteed to be below twice the worst-case cycle time.

In contrast to that, controllable events are classically detected upon a rising or falling edge of the specified trigger condition, i.e., when the controller changes its output in a manner which is relevant for the supervised requirement (cf. Section 5.3.2). As a consequence, the supervisor can only authorize these outputs once. If it does so, the values are allowed to stay unaltered for an arbitrarily long time. That means in return, if the values can eventually cause any undesired yet uncontrollable events, the only way how the supervisor can avoid these is to reject the values right in the moment they are applied.

Cyclic events integrate the aspect of a the bounded reaction time of the PLC with discrete-event systems. They have been presented in [41]. A cyclic event is not only detected when the trigger condition changes from false to true (or reversely) but every time that the condition holds after the controller has been executed. This limits the impact of the event to the duration of one cycle and particularly gives the supervisor the opportunity to reconsider its decision if the current state has changed. In contrast to enforceable events, cyclic events do neither represent a new event class nor do they require special treatment by the synthesis algorithms. Instead, they are ordinary controllable events except that they are not triggered only when the controller's output has changed but after every execution of the controller routine, i.e., once a cycle, as long as their trigger is true.

From an SCT perspective, the key to better permissiveness without explicit preemption is to break *uncontrollable cascades*. Typically, these involve one or several sensor events which finally lead to a state with some forbidden uncontrollable event. Cyclically triggered controllable events provide a way to interrupt such cascades.

To make cyclic events work, it is necessary to provide them with *Code on denial* which revokes the output values that triggered the respective event. An alternative was an altered *Reset on denial* action which sets the outputs back to the values they had in the last cycle where the event was not detected.

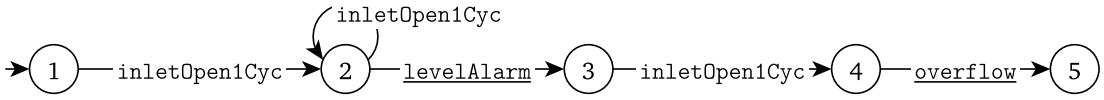


Figure 5.12: Uncontrollable cascade, interrupted by controllable cyclic event

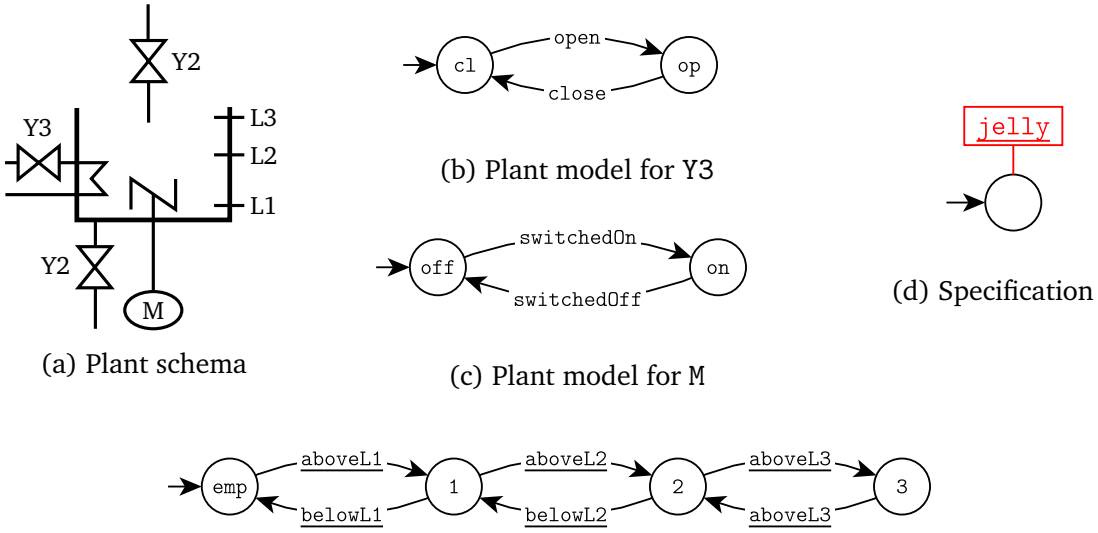
Example 5.10. For this example, the problem of the overflowing tank in Figure 5.7 (pg. 65) is revisited. The cascade of the succeeding uncontrollable events `levelAlarm` and `overflow` needs to be interrupted to increase permissiveness. The event `openInlet` is defined as *detected, controllable* event. It is triggered when the valve is *opened*, i.e., on a (rising) edge of an input. Its action class is *Reset on denial*, i.e., the hardware output which is connected to the valve is set back to the value it had before the controller was invoked if the supervisor rejects it. Remember that synthesis disables `openInlet` in this case.

Figure 5.12 shows an alternative plant automaton where `inletOpen1Cyc` takes the place of `openInlet`. In contrast to the latter, it is not only triggered once the *open* signal is detected but cyclically as long as the valve *is kept open*. Further does it have the action class *code on denial* where the provided code serves to close the valve. Using this plant model, the tank can be filled safely. As soon as the controller attempts to open the valve, the supervisor changes to state 2 and stays there until `levelAlarm` occurs. If then the controller does not close the inlet by setting the output back to value which corresponds to *close*, the supervisor detects `inletOpen1Cyc` once again and rejects it. The specified code then shuts the valve. △

At the first glance, using cyclic events seems counter-intuitive as the controller does not perform an active action which could logically be blocked. On the other hand, the undesired incident (above, `overflow`) does not stay impossible but becomes imminent after some time. Thus, something must happen on the plant that causes a logic change of state.

When comparing the situations (remember the vehicle or the elevator example from Section 5.5.1) which benefit from preemption or cyclic events in terms of increasing permissiveness, one similarity is apparent. They all involve states in which the plants' dynamics evolve on its own after they are entered. In case of Example 5.10, that is the tank which continues being filled without any further action by the controller once the inlet is open. When considering the plant on a more abstract level by hiding the actuators and sensors, the cyclic event makes more sense. It now represents adding a certain volume of liquid to the tank. The precise amount is not known but an upper bound for it thanks to the bounded cycle time. This way, supervision can reasonably determine whether another portion of the liquid can safely be added or not after the level sensor has been reached. The required knowledge about the maximal flow, the worst-case closing time of the valve and the maximum reaction time of the controller are exactly the same as they would be necessary for a conventional safety system.

Example 5.11. Consider a vehicle equipped with a supersonic sensor installed at the front for the sake of obstacle avoidance. Assume that the vehicle has only two possible physical



(a) Plant schema

(b) Plant model for Y3

(c) Plant model for M

(d) Specification

(e) Plant model Lvl representing discretized fill levels

Figure 5.13: Example of a heating tank. All alphabets are disjoint.

states: driving forwards and standing still. Instead of capturing the rising and falling edges of triggers over the speed or the engine's angular momentum for the controllable events, a cyclic event can be defined on the trigger condition speed > 0. Knowing the maximum speed and the cycle time, this event corresponds to *moving the vehicle by a certain known distance*. Again, this event can be granted as long as no obstacle is detected and rejected thereafter. Its *action on denial* then stops the engine and thus prevents a crash. The plant structure is analog to the one in Figure 5.12. △

By breaking uncontrollable cascades already in the automaton, the combination of cyclic events and *action on denial* increases the permissiveness of supervision to a similar extend as preemption[6]. In contrast to the latter it does neither require a third event type nor an adaptation of the synthesis algorithm but can be achieved with conventional SCT.

5.7 Conditional Transitions and Prohibitions

When capturing the coherences and collaboration between several plant parts, states and transitions often need to be provided several times. This is particularly the case when an event needs to be modeled as being possible only when two or more plant components are in a specific state.

Example 5.12. Consider the heating tank example shown in Figure 5.13 which is inspired by the one from [47]. By opening valve Y3, hot liquid streams through a heat exchanger inside the tank. The stirrer can be switched on and off through the motor M. The states of the

[6]The actual preemption is moved out of the scope of DES/SCT into the generated code. The *action on denial* obviously still preempts the incident physically.

plant automaton depicted in Part 5.13e represent four discretized fill levels including empty (`emp`). As in the original example, the liquid must be avoided to gelatinize (uncontrollable `jelly` event). In order to yield a controllable problem, another plant model is required to identify the states in which `jelly` can occur. Assume that it is only possible when the tank contains at least as much product that L1 is reached while the heating is on. Stirring the substance preserves it from gelatinization. The necessary automaton needs to distinguish states of all three existing plant models, i.e., it consists of 16 states, three of which have an outgoing `jelly` transition whereas 13 have none. △

It would in principle be possible to first compute the composition of the existing plant automata, yielding a new plant automaton, and manually add the additional transition afterwards. However, this would not be a well-maintainable and hence not very satisfying solution as it would require to repeat this procedure each time that the original plant automata are changed.

The concepts and results of this section have been published in [42] unless stated differently.

5.7.1 Introduction

Although automata provide a state-based perspective on DES, the semantic notion of these states is not visible outside the automaton they are defined in. Instead, automata are only synchronized on their events, thus on the possible orders of their transitions. From a formal point of view, that makes sense as it allows to intersect their languages, inversely projected onto each other's alphabets.

The human mind, however, rather makes considerations in terms of events (time points) and states (time intervals) and relating both to each other [8]. Hence, for humans modeling automata, i.e., thinking about which event can occur in which state and then causes which transition, is usually easier than providing the language of a plant right away. Unfortunately, this connection of states and transitions is missing amongst several automata. Thus it is not possible to model statements as *"Gelatinization can happen when enough product is in the tank to reach the enabled heating while not being stirred."* without providing an automaton which is able to embody the described situation including all possible paths leading there from the initial state. This issue becomes even more significant when a greater number of plant components are involved and can finally eliminate all benefits of modeling DES modularly.

5.7.1.1 Conditional Transitions and Prohibitions

In order to avoid the necessity of providing all paths to specific, already modeled states redundantly, transitions and prohibitions are equipped with conditions over other automata's states. Depending on the context, these must, may or may not be active for a transition to be takeable and for a prohibition to apply. Practically, such a condition is a propositional logic formula over state identifiers.

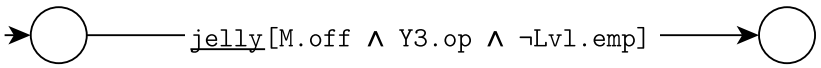


Figure 5.14: Additional plant model to define when gelatinization is possible, $\Sigma = \{\underline{\texttt{jelly}}\}$

If a condition evaluates to false, the corresponding transition/prohibition is ignored as if it did not exist. In the following, *A.q* addresses the state *q* of automaton *A*.

Example 5.13. Figure 5.14 shows the missing plant model for Example 5.12 using a conditional transition. It replaces the 16-states automaton mentioned above, which would be necessary without conditions. △

5.7.1.2 Former Approaches

The idea of augmenting transitions with logic propositions is not new. Lind-Nielsen et al. already applied conditions on active/inactive states to automaton transitions [69], even though in another scientific context. Supremica [1, 2] (cf. Section 3.2.1) follows a slightly different approach and introduces numeric variables to the automata which can be arithmetically manipulated by transition *actions* and checked on transition *guards*. Since these variables are restricted to finite, thus regular, domains [1], Supremica's concept and the one introduced here can theoretically be translated into each other.

Another modeling paradigm is followed by *condition/event systems* [105], which have successfully been used for supervisory control scenarios in the past [61, 63]. The idea of relating logic events with logic states, represented by conditions, forms the conceptual foundation of these systems. One application is the modeling and analysis of cause-effect relationships between several transitions. Condition/event systems are always based on a discrete time model [105].

5.7.2 Resolving Conditions in Composite Automata

Conditional transitions and prohibitions introduce further dependencies between the automata describing a problem. Since the methods of SCT require definite, unconditional transitions, these dependencies need be resolved before synthesis. This is done during automaton composition. Since conditions may only reference existing states of other automata from the scope of the same control problem, the comprehensive composition of all these automata is, by definition, condition-free. This is because in the result of that composition, every condition can finally be evaluated to either *true* or *false*. On *true*, the condition can be omitted, whereas on *false*, the entire transition is dropped. Thanks to that, the expressiveness of DES with conditions is still regular as long as these can be properly resolved.

If a project contains more than two automata, conditions cannot always be fully evaluated immediately. Nevertheless, when two automata are composed, the conditions of the one can be refined on the state information of the other and vice versa.

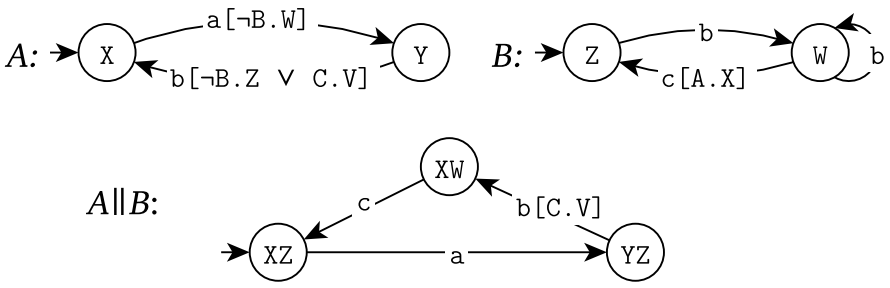


Figure 5.15: Condition refinement in plant composition (figure based on [42])

Example 5.14. Figure 5.15 shows two exemplary plant automata A and B, both having conditional transitions, and their composition. Assume $\Sigma_A = \{\mathtt{a}, \mathtt{b}\}, \Sigma_B = \{\mathtt{b}, \mathtt{c}\}$, thus $\Sigma_{A\|B} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. All occurrences of states inside A and B can be resolved within the composition, whereas `[C.V]` remains. △

In general, a variable inside a condition evaluates to *true* if it references to the source state of the respective transition or prohibition. This naturally includes composite states. For instance, the conditions $[G.q]$ and $[H.p]$ would both evaluate to *true* on transitions leaving the state $(G \parallel H).qp$. In return, a reference to another state of the same automaton evaluates to *false*, such as $G.r$ in $(G \parallel H).qp$ if $r \neq q$.

5.7.2.1 Multiple Transitions

In principle, multiple conditional transitions can be defined on the same event and state without violating determinism as long as their conditions logically exclude each other. Satisfiability checkers could be instrumentalized in order to verify that. An easier solution is to warn the user when the conditions of two outbound transitions of the same state both evaluate to true during composition, and abort.

Multiple conditional prohibitions of the same event are not necessary as these can easily be substituted by one prohibition with a disjunction of several conditions $c_1 \vee \cdots \vee c_n$.

5.7.2.2 Plant Composition

Consider two plants G_1 and G_2 which have conditional transitions and shall be composed with each other. Both plants are still synchronized via their events. Hence, transitions on different events can be handled independently from each other.

For each event e for which a transition exists at the currently considered state, several cases need to be distinguished: (1) e, is not in the alphabet of one plant $G_i, i \in \{1, 2\}$, (2) it is in both plants' alphabets but only one plant G_i defines e-transitions on the current state, and (3) the respective current states of both plants provide transitions on e. In case (1) the conditions can be refined on G_i's current state as described above since G_{3-i} can neither provide any information on e nor change its state on this event. In the second case, G_i claims that the event is not possible to occur, i.e., all e transitions can be omitted at this

state regardless of their conditions. Case (3) requires that the conditions of all e-transitions are combined pairwise. For every such pair one new transition to the respective composed target state is inserted. The two conditions are conjuncted because both must hold for the new transition to be feasible. The formal composition of plants with conditional transitions is given in Section 5.8.4.

5.7.2.3 Specification and Plant Models

When a plant model is applied onto a specification or on an existing synthesis automaton, the situation is slightly more complicated. The reason is that specifications include both transitions and prohibitions. Besides, they may reside in their current state if first no condition on any outbound transition is fulfilled, and second the event is not part of the plant's alphabet or, equivalently, the synthesis automaton's *plant alphabet* (cf. Section 5.3). Plant models serve the purpose to narrow specifications onto actually possible cases. A prohibition is only kept in the composition if either the plant does not know the event or it has a transition for it leaving the current state. Otherwise, the event is not imminent in that situation.

Removing prohibitions of such impossible events is important to ensure maximal permissiveness of the result, at least if the event is uncontrollable. Hence, prohibitions depend on the existence of corresponding transitions on the plant and, by that, on their respective conditions. On the other hand, prohibitions obviously need to be dominant over transitions to not violate the specification. These three aspects, *specification resides in current state, forbidden events need to be possible to stay prohibited, prohibition dominates transitions*, must be considered when composing a specification/synthesis automaton with a plant. As for the plant-plant case, a formal definition can be found in Section 5.8.4.

Example 5.15. Figure 5.16 shows all cases for the composition of a specification H and a plant G schematically. It is assumed that no refinements can be applied on the conditions between H and G, i.e., they all exclusively address states of other automata. △

5.7.3 Synthesis on DES with Conditions

The user is responsible for applying all relevant plants to the specification before monolithic synthesis is invoked. This also includes resolving all conditions. Transitions and forbidden events which still carry conditions in the stage of synthesis will be treated conservatively, i.e., controllable and uncontrollable events are assumed to be possible in principle and thus potentially need to be avoided, whilst conditional enforceable transitions are ignored. If a supervisor exists, it is guaranteed to be safe but unlikely to be maximally permissive.

Incremental synthesis is meant to be executed on a bare specification and includes all required plants automatically. Conditional prohibitions of uncontrollable events are always resolved by composition with the referenced automata before any further actions are taken which would change the behavior of the supervisor. The same holds for enforceable transitions before they are activated, i.e., the algorithm makes sure that preemption is indeed possible in the respective situation.

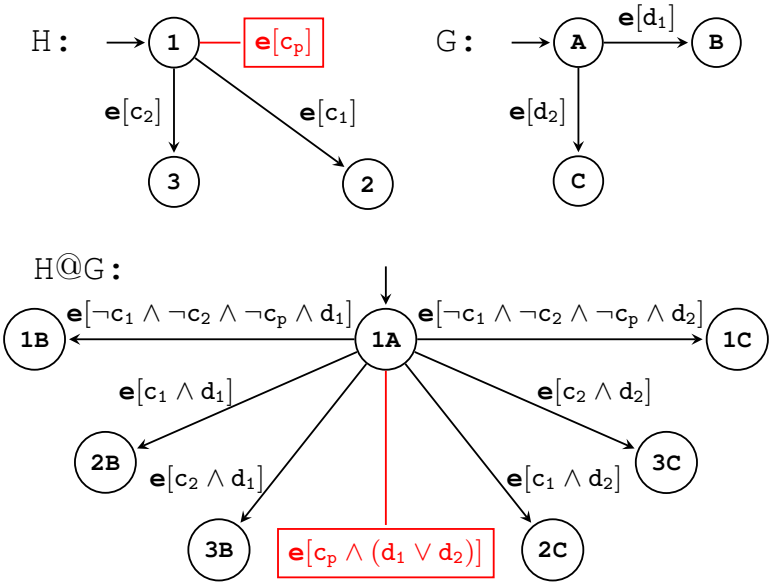


Figure 5.16: Specification-plant composition with conditions (figure taken from [42])

Anyway, the final supervisor must not contain multiple transitions per event anymore as the presented approach does not provide condition evaluation during runtime yet. However, detecting such in the resulting automaton is trivial.

5.7.4 Outlook

In general, conditional transitions provide the user with a powerful yet efficient way of modeling. They allow to restrict statements, such as something being prohibited, possible or causing the current state to change, to certain states of other components or even combinations of these. In the presented approach, conditions still need to be resolved by composition with the referenced automata before or, at the latest, during synthesis. In the future it would be desirable to allow synthesis to operate with conditional transitions rather than just resolving them by composition. When using symbolic methods which invoke satisfiability or BDD solvers like [26, 50, 116, 113], this could increase performance and, by that, the admissible size and complexity of models significantly. Ultimately, it could be possible to support conditional transitions and prohibitions in the final supervisors. However, this would require communication and synchronization of supervisors during runtime which can be problematically in terms of time predictability. Either way, additional research would be necessary to investigate the chances, risks and limitations in actually lifting synthesis to conditional DES. As long as nonblockingness is neglected, the state spaces often keep manageable sizes anyhow, such that resolution by composition is usually practicable.

5.8 Formal Model

The supervisory control theory as defined by [87, 88, 23] and sketched in Chapter 2 is a formal framework that addresses a specific problem, thus a *calculus*. The concepts and alterations introduced in the previous sections do not precisely match the definitions of SCT. Instead of transferring the existing calculus onto the practical problem, the modeling concept has been derived from an applicational point of view. A dualism between the latter and the SCT could serve as a basis to prove the correctness of the methods. However, since the classic SCT lacks in the combined representation of admissible and possible behaviors within the same automaton, the algorithms for composition and synthesis in this case had to undergo substantial transformations as well. Without these being proven to be sound as well, an implementation had to be based on the SCT models directly or their classification as a *formal method* would be at least questionable. In order to avoid that detour through SCT, a new, self-contained formalization is given in this section. The different automata types (specifications, supervisors, etc.) are formalized as separate entities with their own semantics as informally introduced before.

Compared to the Ramade-Wonham formalism, the presented framework is rather technical as it represents an ex-post formalization. However, its intention is rather to reflect the models and methods as precisely as possible in order to prove their soundness than to give an intuitive introduction to a concept. The two synthesis algorithms are defined on top of the formal model along with certain properties which are necessary to prove their correctness.

Unlike the traditional SCT, the discussed formalism is entirely based on automata as these, besides the events, form the basis of the model. The user provides both the requirements and the physical plant capabilities in the shape of automata. Hence it makes sense that also the algorithms operate on these. Formal languages as primarily used by Ramadge and Wonham only serve to define and proove properties but at no time need to be stored or handled by the algorithms.

Another difference is that the language of the entire physically possible plant behavior is not assumed to be known as in many other contributions where the generator *defines* that language. It will be referenced by L_{phy} within this section. The plant contract only applies some restrictions on L_{phy} but does not allow to produce or reconstruct it in detail. The same holds for the alphabet of all possible events, $\hat{\Sigma}$, which will be used to define and proove properties but also needs not to be known entirely.

The incremental synthesis algorithm has not been part of former publications or tools yet, neither in the presented shape nor based on traditional SCT automata.

5.8.1 Basic Concepts

In this section, the basic automaton concepts will be defined as well as their compositions. To formally capture the scope of the requirements that shall be enforced by supervision, the *environment* is introduced.

Definition 1 (Environment) A supervisory control environment E is a tuple $E = (\hat{\Sigma}, L_{phy}, \mathfrak{H}, \mathfrak{G})$ of

- a master alphabet $\hat{\Sigma}$
- the prefix-closed language $L_{phy} \subseteq \hat{\Sigma}^*$ of the physical plant's possible behavior
- a set of plant models $\mathfrak{G}$
- a set of specifications $\mathfrak{H}$.

The master alphabet represents the set of all events that are possible in the considered scenario. It is partitioned into the classes of *controllable events* $\hat{\Sigma}_c$ and *uncontrollable events* $\hat{\Sigma}_u$. Note that neither $\hat{\Sigma}$ nor L_{phy} have to be known and L_{phy} does not necessarily have to be regular. $\mathfrak{G}$ and $\mathfrak{H}$ are the sets of plant models and specifications the user has provided for the problem. As usual on modular systems, events form the basis for the synchronization of multiple automata. The master alphabet $\hat{\Sigma}$ must at least contain all events used in $\mathfrak{G}, \mathfrak{H}$ and L_{phy} but may be larger.

5.8.1.1 Safety Specifications

The starting point for supervisor synthesis is a specification automaton.

Definition 2 (Specification) A specification is a quadruple $H = (Q, f, P, q_0)$ of

- a finite set of states Q
- a partial transition function $f : Q \times \hat{\Sigma} \rightharpoonup Q$
- a prohibition map $P : Q \to 2^{\hat{\Sigma}}$
- an initial state q_0.

For f and all other partially defined functions we use the notation $f(x)!$ if f is defined on x and $f(x)\not!$ if not. Further, when for a set M it is stated that $f(x) \in M$ we implicitly include $f(x)!$ to the statement.

When clear from the context, the index notation X_Y refers to the element X in the tuple $Y = (\ldots, X, \ldots)$, e.g., Q_H addresses the state space of $H = (Q, f, P, q_0)$. Note that Definition 2 does not involve a dedicated alphabet for the specification as it reflects the automaton type as described in Section 5.3.

In the following we assume all specifications to be *consistent*, i.e., $\forall q \in Q$ holds $e \in P(q) \implies f(q,e)!$.

Definition 3 (Transitive Path Function) The transitive path function $f^* : Q \times \hat{\Sigma}^* \rightharpoonup Q$ for specifications is recursively defined by

$$
\begin{aligned}
f^*(q, \varepsilon) &= q \\
f^*(q, es) &= \begin{cases} f^*(q,s) & \text{if } f(q,e)! \ \wedge \ e \notin P(q) \\ f^*(f(q,e),s) & \text{if } f(q,e)! \\ \text{undefined} & \text{otherwise} \end{cases}
\end{aligned}
$$

where $e \in \hat{\Sigma}$, $s \in \hat{\Sigma}^*$. For brevity, we sometimes write $f^*(e)$ instead of $f^*(q_0, e)$. The transition function f is an essential element of each automaton definition, which has to be stored explicitly, whereas f^* is implicitly given. It is required for propositions and proofs only but not needed for operators and algorithms. Each specification defines two languages over $\hat{\Sigma}$.

DEFINITION 4 (Languages of a Specification) Let $H \in \mathfrak{H}$ be a specification and f^* its transitive path function. The *prohibited language* of a specification is defined by $\mathfrak{L}_{\mathcal{P}}(H) = \{ses' \in \hat{\Sigma}^* \mid s, s' \in \hat{\Sigma}^*, f^*(s)!, e \in P(f^*(s))\}$, its *safe language* by $\mathfrak{L}_{\mathcal{S}}(H) = \hat{\Sigma}^* \setminus \mathfrak{L}_{\mathcal{P}}(H)$.

Intuitively, all words that end on a prohibited event and all possible continuations of those words are contained in $\mathfrak{L}_{\mathcal{P}}(H)$, i.e., $\mathfrak{L}_{\mathcal{P}}(H)\hat{\Sigma}^* = \mathfrak{L}_{\mathcal{P}}(H)$. We hence call $\mathfrak{L}_{\mathcal{P}}(H)$ *closed under continuation* over $\hat{\Sigma}$. This suits the common definition of safety properties in the theory of ω-regular languages[7]. $\mathfrak{L}_{\mathcal{S}}(H)$ contains all event sequences of $\hat{\Sigma}^*$ that are not forbidden, by H, i.e., allowed.

LEMMA 1 (Prefix closure of $\mathfrak{L}_{\mathcal{S}}$) Let $H \in \mathfrak{H}$ be a specification. $\mathfrak{L}_{\mathcal{S}}(H)$ is prefix closed.

Proof. Obviously, $\hat{\Sigma}^*$ is prefix-closed. $\mathfrak{L}_{\mathcal{P}}$ is closed under continuation. The complement of a such a language inside a prefix closed domain is prefix-closed. □

That means that a safe event sequence must have been safe all the time before.

DEFINITION 5 (Dual SCT Specification) Every specification H can be translated to an equivalent specification H' in the sense of [87, 88, 23] where $H' = (Q, \Sigma', f', \Gamma, q_0')$, $f' : Q \times \Sigma' \rightharpoonup Q$, $\Sigma' := \bigcup_{q \in Q} \big(\{e \in \hat{\Sigma} \mid f(q,e)!\} \cup P(q)\big)$, $\Gamma(q) := \Sigma' \setminus P(q)$, $q_0' := q_0$, and

$$f'(q,e) := \begin{cases} q & \text{if } f(q,e)! \wedge e \notin P(q) \\ f(q,e) & \text{if } f(q,e)! \\ \text{undefined} & \text{if } e \in P(q) \end{cases}$$

The first case in the definition of f' inserts loops for legal events. Σ' contains all events that occur at H's transitions or that are prohibited at some state.

Definition 5 illustrates that the expressiveness of both ways of modeling does not differ. The translation to the opposite direction is omitted but works in exact analogy.

5.8.1.2 The Specification Product

In most cases it makes sense to synthesize the supervisor for each specification separately as the state space is smaller on average. However, it is sometimes reasonable to fuse multiple specifications to one, e.g., to avoid prohibited enforceable events being used for preemption (cf. Section 6.4.2). This can be done with the specification product.

DEFINITION 6 (Specification Product, $|$) Given two specifications $H_1 = (Q_1, f_1, P_1, q_{01}), H_2 = (Q_2, f_2, P_2, q_{02})$ their product "$|$" is defined by $H_1|H_2 = Ac(Q, f, P, q_0) = H$ where

[7] In ω-regular language theory, an infinite string is unsafe if it has a finite prefix that violates a safety property, cf. Section 4.3.2.

- $Q := Q_1 \times Q_2$
- $P : Q \to 2^{\hat{\Sigma}}, \quad P(q_1, q_2) := P_1(q_1) \cup P_2(q_2)$
- $f : Q \times \hat{\Sigma} \rightharpoonup Q$

$$f((q_1,q_2),e) = \begin{cases} (f_1(q_1,e), f_2(q_2,e)) & \text{if } f_1(q_1,e)! \wedge f_2(q_2,e)! \\ (f_1(q_1,e), q_2) & \text{if } f_1(q_1,e)! \wedge f_2(q_2,e)\not! \wedge e \notin P_2(q) \\ (q_1, f_2(q_2,e)) & \text{if } f_2(q_2,e)! \wedge f_1(q_1,e)\not! \wedge e \notin P_1(q) \\ \text{undefined} & \text{otherwise} \end{cases}$$

- $q_0 := (q_{01}, q_{02})$

Ac denotes the *accessible* function which reduces an automaton to the parts reachable from its initial state q_0[8]: $Ac(Q, f, P, q_0) = (Q', f', P', q_0)$ where

- $Q' = \{q \in Q \mid \exists s \in \hat{\Sigma}^* : f^*(q_0, s)!\}$
- $f' : Q' \times \hat{\Sigma} \rightharpoonup Q', \quad f'(q,e) = f(q,e)$
- $P' : Q' \to 2^{\hat{\Sigma}}, \quad P'(q) = P(q)$

LEMMA 2 Let $H_1, H_2 \in \mathfrak{H}$ be consistent. Then $H = H_1|H_2$ is also consistent.

Proof. For each $q_1 \in Q_1,\ q_2 \in Q_2,\ e \in \hat{\Sigma}$ holds:

- If $f_1(q_1,e)!$ and $f_2(q_2,e)!$ then (due to consistency of H_1, H_2) holds $e \notin P_1(q_1) \cup P_2(q_2) = P(q) \longrightarrow$ consistent.
- If $f_1(q_1,e)!$ and $e \in P_2(q_2)$ then $f((q_1,q_2),e)\not!$ by construction.
- If $f_2(q_2,e)!$ and $e \in P_1(q_1) \ldots$ (analogue).
- If $f_1(q_1,e)\not!$ and $f_2(q_2,e)\not!$ then also $f((q_1,q_2),e)\not!$.

This covers all cases that could possibly violate consistency. □

The result of the specification product is again a specification, which prohibits every word that has been prohibited by one of its operands.

LEMMA 3 Consider two Specifications $H_1, H_2 \in \mathfrak{H}$ then

$$\begin{aligned} \mathfrak{L}_{\mathcal{P}}(H_1|H_2) &= \mathfrak{L}_{\mathcal{P}}(H_1) \cup \mathfrak{L}_{\mathcal{P}}(H_2) \\ \mathfrak{L}_{\mathcal{S}}(H_1|H_2) &= \mathfrak{L}_{\mathcal{S}}(H_1) \cap \mathfrak{L}_{\mathcal{S}}(H_2) \end{aligned}$$

The intuition is: Everything forbidden by at least one specification stays forbidden in the specifications' product. Everything that has not yet been forbidden stays allowed.

[8] In a software implementation, *Ac* is implicitly given when the composition is calculated by traversal from q_{01} and q_{02} instead of the cartesian product

Proof. First line ($\mathfrak{L}_\mathcal{P}$):
From the definition of $\mathfrak{L}_\mathcal{P}$ we obtain[9]:

$$\begin{aligned}\mathfrak{L}_\mathcal{P}(H_1) &= \{ses' \in \hat{\Sigma}^* \mid s,s' \in \hat{\Sigma}^*, f_1^*(q_{01},s)!, e \in P_1(f_1^*(q_{01},s))\}\\ \mathfrak{L}_\mathcal{P}(H_2) &= \{ses' \in \hat{\Sigma}^* \mid s,s' \in \hat{\Sigma}^*, f_2^*(q_{01},s)!, e \in P_2(f_2^*(q_{01},s))\}\\ \mathfrak{L}_\mathcal{P}(H_1|H_2) &= \{ses' \in \hat{\Sigma}^* \mid s,s' \in \hat{\Sigma}^*, f_|^*(q_{01},s)!, e \in P_|(f_|^*((q_{01},q_{02}),s)\}\end{aligned}$$

$\subseteq$:
Let $w \in \mathfrak{L}_\mathcal{P}(H_1|H_2)$. Then there is a representation $ses' = w$ so that $f_|^*((q_{01},q_{02}),s)!$ and $e \in P_|(f^*((q_{01},q_{02}),s))$. Due to the definition of f^* and the consistency claim this representation is unique as s cannot contain prohibited events. In other words e is the "first" prohibited event in string w. By the definition of $f_|$ holds for all $\sigma_i,\ 1 \leq i \leq n$ where $\sigma_1 \ldots \sigma_n = s$ [10] that

$$\begin{aligned}\sigma_i &\notin P_1(f_1^*(q_{01},\sigma_1 \ldots \sigma_{i-1})) \qquad \text{and}\\ \sigma_i &\notin P_2(f_2^*(q_{02},\sigma_1 \ldots \sigma_{i-1}))\end{aligned}$$

which means that no event before e has been forbidden in H_1 or in H_2. From the definition of $P_|$ we achieve

$$e \in P_1(f_1^*(q_{01},s)) \quad \text{or} \quad e \in P_2(f_2^*(q_{02},s))$$

Hence $se \in \mathfrak{L}_\mathcal{P}(H_1)$ or $se \in \mathfrak{L}_\mathcal{P}(H_2)$ (or both). As in the definition of $\mathfrak{L}_\mathcal{P}$ the forbidden event e can be continued by an arbitrary string taken from $\hat{\Sigma}^*$ the above also holds for the suffix s' within w, i.e., $ses' = w \in \mathfrak{L}_\mathcal{P}(H_1)$ or $ses' = w \in \mathfrak{L}_\mathcal{P}(H_2)$ and hence $w \in \mathfrak{L}_\mathcal{P}(H_1) \cup \mathfrak{L}_\mathcal{P}(H_2)$.
$\supseteq$:
I. Let $w \in \mathfrak{L}_\mathcal{P}(H_1)$. Then there is a unique representation $ses' = w$ with $f_1^*(q_{01},s)$ defined and $e \in P_1(f_1^*(q_{01},s))$. Regarding s and H_2 there are two cases:

1. $s \notin \mathfrak{L}_\mathcal{P}(H_2)$. Then $f_2(q_{02},s)$ is defined. Due to construction of $f_|$ and $f_|^*$, $f_|^*((q_{01},q_{02}),s)$ is also defined and $e \in P_|(f_|^*((q_{01},q_{02}),s))$. It follows $se \in \mathfrak{L}_\mathcal{P}(H_1|H_2)$. For the same reason as in "$\subseteq$" (closed under continuation), $w = ses' \in \mathfrak{L}_\mathcal{P}(H_1|H_2)$.
2. $s \in \mathfrak{L}_\mathcal{P}(H_2)$. In return, there is a decomposition $\hat{s}\hat{e}\hat{s}' = s$ such that II. applies. Since $\mathfrak{L}_\mathcal{P}$ is closed under continuation, $s = \hat{s}\hat{e}\hat{s}'es' = w \in \mathfrak{L}_\mathcal{P}(H_2)$, thus II. applies to w, too.

II. $w \in \mathfrak{L}_\mathcal{P}(H_2)$. Since the definitions of $f_|$, $P_|$ and $f_|^*$ are commutative w. r. t. H_1 and H_2, $w \in \mathfrak{L}_\mathcal{P}(H_1|H_2)$ holds in analogy to I.

For the second line ($\mathfrak{L}_\mathcal{S}$) holds

$$\mathfrak{L}_\mathcal{S}(H_1|H_2) = \hat{\Sigma}^* \setminus \mathfrak{L}_\mathcal{P}(H_1|H_2) = \hat{\Sigma}^* \setminus (\mathfrak{L}_\mathcal{P}(H_1) \cup \mathfrak{L}_\mathcal{P}(H_2)) = \mathfrak{L}_\mathcal{S}(H_1) \cap \mathfrak{L}_\mathcal{S}(H_2) \qquad \square$$

5.8.1.3 Supervisors

A supervisor is modeled like a specification except it may disable controllable events only.

DEFINITION 7 (Supervisor) A supervisor is a quadruple $S = (Q, f, P, q_0)$ with

[9] For brevity we use $f_|, P_|, \ldots$ for $f_{H_1|H_2}, P_{H_1|H_2}, \ldots$
[10] Define $\sigma_1\sigma_0 := \varepsilon$

- a finite set of states Q
- a partial transition function $f : Q \times \hat{\Sigma} \rightharpoonup Q$
- a prohibition map $P : Q \to 2^{\hat{\Sigma}_c}$
- an initial state q_0

Definition 8 (Disabled Language of a Supervisor) The disabled language of a supervisor is defined by: $\mathfrak{L}_{\mathcal{D}}(S) = \{ses' \in \hat{\Sigma}^* \mid s, s' \in \hat{\Sigma}^*, f^*(s)!, e \in P(f^*(s))\}$ where the definition of f^* is analog to Definition 3.

Definition 9 (Supervised Plant) Let S be a supervisor. The language of the supervised (physical) plant is $L_{phy} \setminus \mathfrak{L}_{\mathcal{D}}(S)$.

Lemma 4 (Controllability of $L \setminus \mathfrak{L}_{\mathcal{D}}$) Let S be a supervisor and $L \subseteq \hat{\Sigma}^*$ an arbitrary language. Its "enabled" sublanguage $L \setminus \mathfrak{L}_{\mathcal{D}}(S)$ is controllable with respect to L.

Proof. Let $s \in L \setminus \mathfrak{L}_{\mathcal{D}}(S)$ and s' an arbitrarily chosen prefix of s, i.e., $s's'' = s$ for some s''. For every $e \in \hat{\Sigma}_U$ holds one of the following two cases:
1. $s'e \in L$. Since $s \notin \mathfrak{L}_{\mathcal{D}}(S)$ holds $s' \notin \mathfrak{L}_{\mathcal{D}}(S)$, so $f^*(s')$ is defined. We know that a supervisor does not prohibit uncontrollable events, $P(f^*(s')) \cap \hat{\Sigma}_U = \emptyset$. It follows $s'e \notin \mathfrak{L}_{\mathcal{D}}(S)$.
2. $s'e \notin L$. In that case $s'e \notin (L \setminus \mathfrak{L}_{\mathcal{D}}(S))\hat{\Sigma}_U \cap L$. Thus it can be ignored.

Finally, we obtain $\overline{L \setminus \mathfrak{L}_{\mathcal{D}}(S)}\ \hat{\Sigma}_U \cap L \subseteq L \setminus \mathfrak{L}_{\mathcal{D}}(S)$. □

Lemma 4 holds particularly for $L_{phy} \subseteq \hat{\Sigma}^*$. The distinction of $\mathfrak{L}_{\mathcal{P}}$ and $\mathfrak{L}_{\mathcal{D}}$ is meant to emphasize that lemma 4 holds for $\mathfrak{L}_{\mathcal{D}}$ but not necessarily for $\mathfrak{L}_{\mathcal{P}}$ of an arbitrary specification or synthesis automaton.

5.8.1.4 Safety and Permissiveness

A supervisor is intended to enforce a specification on the physical plant. If this is the case for some $H \in \mathfrak{H}$, we call the supervisor H-safe.

Definition 10 (Safety) Let $H \in \mathfrak{H}$ be a specification, S a supervisor and $L \subseteq \hat{\Sigma}^*$ a language. S is called *H-safe* w.r.t. L iff $L \setminus \mathfrak{L}_{\mathcal{D}}(S) \subseteq \mathfrak{L}_{\mathcal{S}}(H)$.

It follows that S is H-safe $\iff L \cap \mathfrak{L}_{\mathcal{P}}(H) \subseteq \mathfrak{L}_{\mathcal{D}}(S)$. A given specification $H \in \mathfrak{H}$ may define a safe language which is not controllable. Uncontrollable events cannot be disabled by definition. Instead, the circumstances where the event is (1) not allowed and at the same time (2) possible to occur have to be avoided by a suitable supervisor. Aspect (1) addresses the specification state containing the prohibition, aspect (2) depends on physical limitations of the plant which can be captured in *plant models* as introduced in section 5.3.

There can be specifications though which are not *realizable* on L as they prohibit events which can happen in unavoidable circumstances.

Definition 11 Let $H \in \mathfrak{H}$ a specification. H is *unrealizable* on L if $L \cap \mathfrak{L}_{\mathcal{P}}(H) \cap \hat{\Sigma}_u^* \neq \emptyset$.

For these problems exists no H-safe supervisor. Accordingly, the SCSL of H is empty.

Since a safe supervisor could be more restrictive than necessary, the optimality criterion for supervisors is maximal permissiveness, defined in analogy to [23].

DEFINITION 12 (Maximal Permissiveness) An H-safe supervisor S is *maximally permissive* w.r.t. $L \subseteq \hat{\Sigma}^*$ iff $\forall S'$: S' is H-safe w.r.t. $L \implies L \setminus \mathfrak{L}_{\mathcal{D}}(S') \subseteq L \setminus \mathfrak{L}_{\mathcal{D}}(S)$

When L is not explicitly stated, we implicitly refer to L_{phy}. In many cases it is possible to obtain a safe supervisor without any knowledge about the plant. This refers to aspect (1), prevent the circumstances where an event is prohibited. However, a reasonable supervisor should be both safe and maximally permissive, which involves taking into account where an event is able to happen.

5.8.1.5 Plant Models

In order to determine when events are physically able to happen, and only for that, it is necessary to take into account available knowledge about the plant. This can be done by providing automata which model the reachable state space of plant parts and thus provide an approximation of L_{phy}.

In contrast to specifications, plant models are defined with respect to a given alphabet. That alphabet is the most important component of a plant automaton as it defines which events are related to each other. Dependent on that alphabet the states and transitions have to define all constellations and sequences of the contained events that can possibly occur in order to comply with the plant contract. The plant automata thus yield the same languages as in other modular approaches [23, 117, 120, 35], although they do not serve as *generators*, cf. Section 2.1.

DEFINITION 13 (Plant Model) A plant model is a quadruple $G = (Q, \Sigma, f, q_0)$ with

- a finite set of states Q
- a finite alphabet $\Sigma \subseteq \hat{\Sigma}$
- a partial transition function $f : Q \times \Sigma \rightharpoonup Q$
- an initial state $q_0 \in Q$

DEFINITION 14 (f^*) For a given plant model G we define the transitive path function $f^* : Q \times \Sigma^* \rightharpoonup Q$ recursively:

$$\begin{aligned} f^*(q, \varepsilon) &= q \\ f^*(q, es) &= f^*(f(q, e), s) \end{aligned}$$

DEFINITION 15 (Languages of a Plant, $\mathfrak{L}, \mathfrak{L}^{\Pi}$) The *internal language* $\mathfrak{L}$ of a plant is defined by: $\mathfrak{L}(G) := \{s \in \Sigma^* \mid f^*(q_0, s)!\}$. Its *achievable language* $\mathfrak{L}^{\Pi}$ is defined by $\mathfrak{L}^{\Pi}(G) := \Pi^{-1}_{\Sigma_G}(\mathfrak{L}(G))$ where Π^{-1}_{Σ} is the inverse of the natural projection from $\hat{\Sigma}^*$ to Σ^* [23], element-wise lifted to sets of strings. It is defined by:

$$\Pi^{-1}_{\Sigma} : 2^{\Sigma^*} \to 2^{\hat{\Sigma}^*}, \quad L \mapsto \bigcup_{e_1 \ldots e_n \in L} M e_1 M \ldots M e_n M, \text{ where } M := (\hat{\Sigma} \setminus \Sigma)^*$$

The set $\mathfrak{L}^{\Pi}(G)$ contains all words that the G is able to capture by ignoring all events $e \notin \Sigma_G$. To guarantee correct synthesis, we must claim that all plant models are correct, meaning their achievable languages are over-approximations of L_{phy}. Therefore, Definition 16, the formalization of the *plant contract*, is established.

Definition 16 (Plant contract, correctness of plant models) Let L be a language.

- A plant model G is called *correct w.r.t.* L if $L \subseteq \mathfrak{L}^{\Pi}(G)$.
- A set B of plant models is called *correct w.r.t.* L if all $G \in B$ are correct w.r.t. L.

According to the plant contract, we assume the provided set $\mathfrak{G}$ to be correct w.r.t. L_{phy}. The interaction of two plant models is given by their parallel composition. The following conception is common in the community [23].

Definition 17 (Plant Composition, $\|$) Given two plant models $G_1 = (Q_1, \Sigma_1, f_1, q_{01})$ and $G_2 = (Q_2, \Sigma_2, f_2, q_{02})$, their parallel composition is defined by $G_1 \parallel G_2 = Ac(Q, \Sigma, f, q_0)$,

- $Q := Q_1 \times Q_2$
- $\Sigma := \Sigma_1 \cup \Sigma_2$
- $f : Q \times \Sigma \rightharpoonup Q$

$$f((q_1,q_2),e) := \begin{cases} (f_1(q_1,e), f_2(q_2,e)) & \text{if } f_1(q_1,e)! \wedge f_2(q_2,e)! \\ (f_1(q_1,e), q_2) & \text{if } f_1(q_1,e)! \wedge e \notin \Sigma_2 \\ (q_1, f_2(q_2,e)) & \text{if } f_2(q_2,e)! \wedge e \notin \Sigma_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

- $q_0 := (q_{01}, q_{02})$.

Ac is defined in analogy to Definition 6 where $\Sigma' = \Sigma$ and P is ignored.

Lemma 5 (Conservation of $\mathfrak{L}^{\Pi}$, from [23]) Let $G_1, G_2 \in \mathfrak{G}$. For $G_1 \parallel G_2$ holds

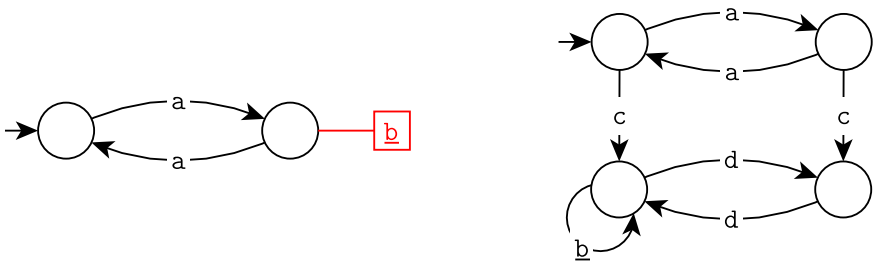
$$\mathfrak{L}^{\Pi}(G_1 \parallel G_2) \;=\; \mathfrak{L}^{\Pi}(G_1) \cap \mathfrak{L}^{\Pi}(G_2).$$

We denote the product of all plants within $\mathfrak{G}$ as $\hat{G} := \|_{G \in \mathfrak{G}} G$ and the set of all possible plant products in $\mathfrak{G}$ as $\hat{\mathfrak{G}} := \{G_1 \parallel \ldots \parallel G_n \mid G_1, \ldots, G_n \in \mathfrak{G}\}$. Obviously, $\hat{G} \in \hat{\mathfrak{G}}$.

Lemma 6 All $G \in \hat{\mathfrak{G}}$ are correct and all $\mathfrak{L}^{\Pi}(G), G \in \hat{\mathfrak{G}}$ are prefix-closed.

Proof. Correctness: Via induction.
Basis: Let G_1, G_2 be correct. Then (due to Lemma 5) holds $L_{phy} \subseteq \mathfrak{L}^{\Pi}(G_1) \cap \mathfrak{L}^{\Pi}(G_2) = \mathfrak{L}^{\Pi}(G_1 \parallel G_2)$. Thus, $G' = G_1 \parallel G_2$ is correct.
Step: Let G_1, G_2 be products of plants and correct. Then, analogously to induction basis, $G_1 \parallel G_2$ is correct. Since $\mathfrak{G}$ is assumed to be correct it follows $\hat{\mathfrak{G}}$ is also correct.

Prefix Closedness: The prefix closedness of $\mathfrak{L}(G)$ follows directly from the recursive definition of f^* (Definition 14), i.e., $f^*(q,se)! \implies f^*(q,s)!$. Since the inverse natural projection injects **all** possible sequences of events outside Σ, it also injects all prefixes of those sequences. Thus, for each word $uvu' \in \mathfrak{L}^{\Pi}(G)$, where all prefixes of uu' are in $\mathfrak{L}^{\Pi}(G)$ and an injected string $v \in (\hat{\Sigma} \setminus \Sigma)^*$ holds that $uv' \in \mathfrak{L}^{\Pi}(G)$ for every prefix v' of v. □



(a) Specification H (b) Plant G

Figure 5.17: Specification and regular plant approximation

5.8.1.6 Maximal Achievable Permissiveness

As discussed above, the criterion for an optimal supervisor is its permissiveness w.r.t. L_{phy}. Since the latter is in general not known, it is not always possible to find the maximally permissive supervisor. For that reason, we relax this requirement and settle to a supervisor that is *maximally permissive as far as we know*, i.e., with respect to the provided plant models in $\mathfrak{G}$. Hence, we define $\mathfrak{L}^{\Pi}(\mathfrak{G}) := \mathfrak{L}^{\Pi}(\hat{G})$ as the best available approximation of plant behavior.

Lemma 7 Let $H \in \mathfrak{H}$, $G \in \hat{\mathfrak{G}}$ and S a supervisor. It holds: S is H-safe w.r.t. $\mathfrak{L}^{\Pi}(G) \implies S$ is H-safe w.r.t. L_{phy}.

Proof. From the correctness of G (Lemma 6) follows $L_{phy} \subseteq \mathfrak{L}^{\Pi}(G)$ and hence $\mathfrak{L}^{\Pi}(G) \setminus \mathfrak{L}_{\mathcal{D}}(S) \subseteq \mathfrak{L}_{\mathcal{S}}(H) \Rightarrow L_{phy} \setminus \mathfrak{L}_{\mathcal{D}}(S) \subseteq \mathfrak{L}_{\mathcal{S}}(H)$. □

This allows us to derive safe supervisors without detailed knowledge about L_{phy} but on the basis of the regular over-approximations given by $\mathfrak{G}$.

Example 5.16. Consider the specification H as shown in Figure 5.17a, $\mathtt{a}, \mathtt{c}, \mathtt{d} \in \hat{\Sigma}_c$, $\underline{\mathtt{b}} \in \hat{\Sigma}_u$, and $L_{phy} = \{\mathtt{a}^n\mathtt{c}\mathtt{d}^n\underline{\mathtt{b}} \mid n \in \mathbb{N}\}$. L_{phy} is obviously not regular thus no DFA is able to recognize L_{phy}. As an approximation, we got G as depicted in Figure 5.17b. $\mathfrak{L}(G) = \overline{(\mathtt{aa})^*\mathtt{c}(\mathtt{dd})^*\underline{\mathtt{b}}^* \cup \mathtt{a}(\mathtt{aa})^*\mathtt{cd}(\mathtt{dd})^*\underline{\mathtt{b}}^*}$. Since $L_{phy} \subseteq \mathfrak{L}^{\Pi}(G)$, G is correct and thus fulfills the plant contract. However, $\mathtt{acdddb} \in \mathfrak{L}^{\Pi}(G) \setminus L_{phy}$, i.e., it is a proper **over**-approximation of L_{phy}. The best possible (non-regular) supervisor would disable $\mathtt{d}$ after $\mathtt{a}^n\mathtt{c}\mathtt{d}^{n-1}$ for odd values of n, resulting in the SCSL of L_{phy}. Using the regular approximation given by G, we get a supervisor S which is as permissive as maximally achievable with G. It disables all words in $\mathfrak{L}_{\mathcal{D}}(S) = (\mathtt{a}(\mathtt{aa})^*\mathtt{cd})\hat{\Sigma}^*$. △

In order to accomplish the best available permissiveness, the entire plant model has to be taken into account in general. In Section 5.8.3 it is shown that there are sufficient criteria for when a single plant model can be ignored by still achieving the same permissiveness though.

Note that a specification that is realizable on L_{phy} might not be realizable on $\mathfrak{L}^{\Pi}(G)$ if the over-approximation given by the plant model G is too coarse. This is the case if an undesired uncontrollable event depends on a preceding controllable one which is not known by G.

5.8.1.7 Applying Plant Models to Specifications

A specification defines certain situations in which a supervisor is supposed to restrict the physical plant's behavior to guarantee safety. This way, a specification forms the basis for the process that finally results in the synthesis of a supervisor. As the optimization criterion for that supervisor is its permissiveness, it should prohibit upcoming events only where necessary. For controllable events, the decision of whether or not to apply a prohibition is straightforward. If a specification requires a controllable event not to happen, there is no potential increase of permissiveness possible as any occurrence of that event would instantly violate the safety conditions defined by the specification.

Prohibitions of uncontrollable events have to be handled differently. Since it cannot be disabled, alternative ways must be found of how to prevent a forbidden uncontrollable event. The restrictions imposed to that end often affect the legal behavior as well. Hence, in contrast to controllable prohibitions, it does play a role for the overall permissiveness whether and under which circumstances an uncontrollable event can occur and, in return, in which situations further restrictions are not necessary.

In the following, the synthesis automaton from Section 5.3 is formally defined, the representations which unites aspects of legality and physical possibility.

Definition 18 (Synthesis Automaton) A synthesis automaton is a quintuple $C = (Q, \Sigma^{\Pi}, f, P, q_0)$ of

- a set of states Q,
- a plant alphabet Σ^{Π},
- a transition function $Q \times \hat{\Sigma} \rightharpoonup Q$,
- a prohibition map $P : Q \to 2^{\hat{\Sigma}}$ and
- an initial state q_0.

By $\mathfrak{C}$ we denote the set of all synthesis automata with $\Sigma^{\Pi} \subseteq \hat{\Sigma}$. The plant alphabet Σ^{Π} contains all events which came into the automaton "by plant", i.e., which the automaton has full transitional knowledge about. In other words, an event $e \in \Sigma^{\Pi}$ for which no transition exists from the current state, is considered to be impossible while for $e' \notin \Sigma^{\Pi}$ this is not known yet. Synthesis automata allow for a successive refinement by further plant models. This way they are suitable to represent intermediate results of the incremental synthesis procedure.

As already mentioned in earlier sections, there have been several concepts for that purpose in former contributions. The most prominent approaches to marry the legal with the physically possible in one representation are forbidden states [68] or blocking states [35]. The former are modeled and considered as potentially reachable and distinguishable parts of the state space. This, however, can lead to misconceptions, especially in the context of preemption. Forbidden states must not be confused with unstable states which are not illegal yet but need to be left immediately. Artificial blocking states, on the other hand, would require to solve the nonblockingness problem.

Definition 19 (Languages of a Synthesis Automaton, $\mathfrak{L}^{\Pi}, \mathfrak{L}_{\mathcal{P}}$) The achievable language of a synthesis automaton $C = (Q, \Sigma^{\Pi}, f, P, q_0)$ contains all possible, allowed strings. The prohibited language contains all forbidden strings that would be possible in the scope of C and thus need be avoided.

$$\mathfrak{L}^{\Pi}(C) = \{s \in \hat{\Sigma}^* \mid f^*(q_0, s)!\}$$
$$\mathfrak{L}_{\mathcal{P}}(C) = \{ses' \in \hat{\Sigma}^* \mid s, s' \in \hat{\Sigma}^* \wedge f^*(q_0, s)! \wedge e \in P(f^*(q_0, s))\}$$

where f^* is defined by ($e \in \hat{\Sigma}$, $s \in \hat{\Sigma}^*$):

$$f^*(q, \varepsilon) = q$$
$$f^*(q, es) = \begin{cases} f^*(f(q,e), s) & \text{if } f(q,e)! \\ f^*(q, s) & \text{if } f(q,e)! \wedge e \notin P(q) \wedge e \notin \Sigma^{\Pi} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The language $\mathfrak{L}^{\Pi}(C)$ is the set of all words that are considered possible by all plant models that are contained in C and allowed by the specification inside C. This may still involve forbidden uncontrollable events. Like for plants, $\mathfrak{L}^{\Pi}(C)$ is prefix-closed. A formal proof is omitted here, but works very similar. Each $H \in \mathfrak{H}$ can be represented as an equivalent synthesis automaton:

Lemma 8 Let $H \in \mathfrak{H}$ and $C_H = (Q_H, \emptyset, f_H, q_{0,H})$. Then $\mathfrak{L}_{\mathcal{S}}(H) = \mathfrak{L}^{\Pi}(C_H)$ and $\mathfrak{L}_{\mathcal{P}}(H) = \mathfrak{L}^{\Pi}_{\mathcal{P}}(C_H)$.

Thanks to that compatibility, every property that holds for a synthesis automaton also holds for a specification.

In the following, the application product is introduced. It adds the information from a plant model to a specification or an existing synthesis automaton. Thus, according to Lemma 8, C can also represent a specification C_H.

Definition 20 (Application product, @) The application of plant model $G \in \mathfrak{G}$ to a synthesis automaton $C \in \mathfrak{C}$ is a synthesis automaton $C@G = Ac(Q, \Sigma^{\Pi}, f, P, q_0) \in \mathfrak{C}$, where

- $Q := Q_C \times Q_G$
- $\Sigma^{\Pi} := \Sigma^{\Pi}_C \cup \Sigma_G$.
- $f : Q \times \hat{\Sigma} \rightharpoonup Q$,
$$f((q_C, q_G), e) := \begin{cases} (f_C(q_C, e), f_G(q_G, e)) & \text{if defined} \\ (q_C, f_G(q_G, e)) & \text{if defined } \wedge f_C(q_C, e)! \\ & \quad \wedge e \notin P_C(q_C) \wedge e \notin \Sigma^{\Pi}_C \\ (f_C(q_C, e), q_G) & \text{if def. } \wedge f_G(q_G, e)! \wedge e \notin \Sigma_G \\ \text{undefined} & \text{otherwise} \end{cases}$$
- $P : Q \to 2^{\hat{\Sigma}}$, $P((q_C, q_G)) := \{e \in \hat{\Sigma} \mid e \in P_C(q_C) \wedge (f_G(q_G, e)! \vee e \notin \Sigma_G)\}$
- $q_0 := (q_{0,C},\ q_{0,G})$

Again, *Ac* is defined in analogy to Definition 6, where Σ^{Π} is untouched.

The relationship between || and @ is captured by the following lemma.

LEMMA 9 For all $C \in \mathfrak{C}, G_1, G_2 \in \mathfrak{G}$ holds:

$$C@(G_1 \parallel G_2) \sim (C@G_1)@G_2 \sim (C@G_2)@G_1$$

where $A \sim B$ denotes that A and B have the same size, structure and languages.
The formal proof is omitted as it is very technical and needs an exhaustive enumeration of combinations of possible cases. The correctness follows by construction of || and @.

While Definition 20 defines the technical construction of a synthesis automaton, the impact of the application product is much easier to understand in terms of the languages associated to the automata, as the following lemma shows.

LEMMA 10 (Properties of $\mathfrak{L}_{\mathcal{P}}(C@G)$ and $\mathfrak{L}^{\Pi}(C@G)$) Let $C \in \mathfrak{C}$ be a synthesis automaton (or specification) and $G \in \hat{\mathfrak{G}}$ a plant model. Then the following equations hold.

$$\begin{aligned} \text{I.} \quad \mathfrak{L}_{\mathcal{P}}(C@G) &= [\mathfrak{L}_{\mathcal{P}}(C) \cap \mathfrak{L}^{\Pi}(G)]\hat{\Sigma}^* \\ \text{II.} \quad \mathfrak{L}^{\Pi}(C@G) &= \mathfrak{L}^{\Pi}(C) \cap \mathfrak{L}^{\Pi}(G) \end{aligned}$$

Proof. Within this proof we abbreviate $P_{C@G}, f_{C@G}, q_{0_{C@G}}, \ldots$ by $P_@, f_@, q_{0@}, \ldots$.

I. – Proof by induction over length of event string s.
Induction basis: Let $s = e \in \hat{\Sigma}$. It holds $e \in \mathfrak{L}_{\mathcal{P}}(C@G)$ iff $e \in P_@(q_{0@})$ iff $e \in P_C(Q_{0C})$ and (case 1) $f_G(q_{0G})$ defined OR (case 2) $e \notin \Sigma_G$. Both cases imply $e \in \mathfrak{L}^{\Pi}(G)$ and either the first or the second is in return implied by $e \in \mathfrak{L}^{\Pi}(G)$.
Induction step: We prove the claim for $se, e \in \hat{\Sigma}$ on the assumption that it holds for s. There are two cases for s:

1. let $s \in \mathfrak{L}_{\mathcal{P}}(C@G)$ and $s \in [\mathfrak{L}_{\mathcal{P}}(C) \cap \mathfrak{L}^{\Pi}(G)]\hat{\Sigma}^*$. Since $\mathfrak{L}_{\mathcal{P}}$ is closed under continuation, $se \in \mathfrak{L}_{\mathcal{P}}(C@G)$ and trivially also $se \in [\mathfrak{L}_{\mathcal{P}}(C) \cap \mathfrak{L}^{\Pi}(G)]\hat{\Sigma}^*$.
2. let $s \notin \mathfrak{L}_{\mathcal{P}}(C@G)$ and $s \notin [\mathfrak{L}_{\mathcal{P}}(C) \cap \mathfrak{L}^{\Pi}]\hat{\Sigma}^*$, i.e., s is legal. We distinguish two sub-cases:
   a) $s \in \mathfrak{L}^{\Pi}(C@G)$ and $s \in \mathfrak{L}^{\Pi}(C) \cap \mathfrak{L}^{\Pi}(G)$ (s is possible). Let $(q_c, q_G) := f^*_@((q_{0C}, q_{0G}), s)$. The rest of this sub-case is analog to the induction basis $s = e$, where q_G and q_C are substituted by q_{0G} and q_{0C}.
   b) $s \notin \mathfrak{L}^{\Pi}(C@G)$ and $s \notin \mathfrak{L}^{\Pi}(C) \cap \mathfrak{L}^{\Pi}(G)$ (s is not possible). Then holds by construction of $P_@$ and P_C and the definition of $\mathfrak{L}_{\mathcal{P}}$ that $se \notin \mathfrak{L}_{\mathcal{P}}(C@G)$ and either $se \notin \mathfrak{L}_{\mathcal{P}}(C)$ or $se \notin \mathfrak{L}^{\Pi}(G)$ – or both. That leads to $se \notin (\mathfrak{L}_{\mathcal{P}}(C) \cap \mathfrak{L}^{\Pi}(G))\hat{\Sigma}^*$.

II. – This proof is analogously structured to the proof of III.
Induction basis: Let $s = \varepsilon$. Trivially, $\varepsilon \in \mathfrak{L}^{\Pi}(C@G)$ and $\varepsilon \in \mathfrak{L}^{\Pi}(C)$ and $\varepsilon \in \mathfrak{L}^{\Pi}(G)$. Hence, $\varepsilon \in \mathfrak{L}^{\Pi}(C) \cap \mathfrak{L}^{\Pi}(G)$.
Induction step: We again distinguish two cases.

1. $s \in \mathfrak{L}^{\Pi}(C@G)$ and $s \in \mathfrak{L}^{\Pi}(C) \cap \mathfrak{L}^{\Pi}(G)$.
   For brevity, we define $q_@ := f^*_@(q_{0@}, s)$, $q_C := f^*_C(q_{0C}, s)$ and $q_G = f^*_G(q_{0G}, s)$. Let $e \in \hat{\Sigma}$ arbitrary. Three bidirectional sub-cases:

$$\begin{array}{l} se \in \mathfrak{L}^{\Pi}(C) \\ \wedge\, se \in \mathfrak{L}^{\Pi}(G) \end{array} \Longleftrightarrow \left\{ \begin{array}{ll} (1) & f_C(q_C, e)!, f_G(q_G, e)! \\ (2) & f_C(q_C, e)!, e \notin \Sigma^{\Pi}_C, e \notin P_C(q_C), f_G(q_G, e)! \\ (3) & f_C(q_C, e)!, f_G(q_G, e)!, e \notin \Sigma_G \end{array} \right.$$

$$\left.\begin{array}{l}\ldots(1)\\ \ldots(2)\\ \ldots(3)\end{array}\right\} \iff f_{@}(q_{@}, e)! \iff se \in \mathfrak{L}^{\Pi}(C@G)$$

2. Since $\mathfrak{L}^{\Pi}$ is prefix-closed, we obtain:
   - $s \notin \mathfrak{L}^{\Pi}(C) \implies se \notin \mathfrak{L}^{\Pi}(C)$
   - $s \notin \mathfrak{L}^{\Pi}(G) \implies se \notin \mathfrak{L}^{\Pi}(G)$
   - $s \notin \mathfrak{L}^{\Pi}(C@G) \implies se \notin \mathfrak{L}^{\Pi}(C@G)$

   Where the first two bullets imply $s \notin \mathfrak{L}^{\Pi}(C) \cap \mathfrak{L}^{\Pi}(G) \implies se \notin \mathfrak{L}^{\Pi}(C) \cap \mathfrak{L}^{\Pi}(G)$. □

We see that a synthesis automaton $C = H@G$ conserves the information about what is possible ($\mathfrak{L}^{\Pi}(G)$) and what is not safe ($\mathfrak{L}_{\mathcal{P}}(H)$). But it dispenses strings that are neither allowed nor possible. That comes from the fact that all considered (infinite) languages $\mathfrak{L}_{\mathcal{P}}, \mathfrak{L}^{\Pi}$, etc. are represented by finite automata. To keep those automata as compact as possible, they do not distinguish whether an impossible string was allowed or not. In other words, they concentrate on the reachable state space. Note that the sets $\mathfrak{L}^{\Pi}(H@G)$ and $\mathfrak{L}_{\mathcal{P}}(H@G)$ do in general not sum up to $\hat{\Sigma}^*$. However, they are always disjoint.

Transitions and states which represent forbidden yet impossible strings are neglected during composition of $C@G$. As a consequence, $\mathfrak{L}_{\mathcal{P}}(C@G)$ indeed prohibits potentially less event sequences than $\mathfrak{L}_{\mathcal{P}}(C)$ did. For a specification H and plant G, that can cause $\hat{\Sigma}^* \setminus \mathfrak{L}_{\mathcal{P}}(H@G) \notin \mathfrak{L}_{\mathcal{S}}(H)$, which appears problematic at the first glance. However, as we claim G being correct, $\mathfrak{L}_{\mathcal{P}}(H@G)$ prohibits all sequences with respect to L_{phy}, as formalized by the following Lemma.

Lemma 11 (Safety of @-Product) Let $G \in \hat{\mathfrak{G}}$ be a plant and C a synthesis automaton. It holds

$$L_{phy} \setminus \mathfrak{L}_{\mathcal{P}}(C@G) = L_{phy} \setminus \mathfrak{L}_{\mathcal{P}}(C) \quad \text{(Safety)}$$

Proof.

$$\begin{aligned}
& L_{phy} \setminus \mathfrak{L}_{\mathcal{P}}(C@G) \\
= \; & L_{phy} \setminus [(\mathfrak{L}_{\mathcal{P}}(C) \cap \mathfrak{L}^{\Pi}(G))\, \hat{\Sigma}^*] \\
= \; & [L_{phy} \setminus \mathfrak{L}_{\mathcal{P}}(C)] \cup [L_{phy} \setminus (\mathfrak{L}^{\Pi}(G)\, \hat{\Sigma}^*)] \\
= \; & [L_{phy} \setminus \mathfrak{L}_{\mathcal{P}}(C)] \cup [\underbrace{(L_{phy} \setminus \mathfrak{L}^{\Pi}(G))}_{=\emptyset\ (\dagger)} \setminus (\mathfrak{L}^{\Pi}(G)\, \hat{\Sigma}^*)] \\
= \; & L_{phy} \setminus \mathfrak{L}_{\mathcal{P}}(C)
\end{aligned}$$

(†) since $L_{phy} \subseteq \mathfrak{L}^{\Pi}(G)$ (correctness of G). □

5.8.2 Monolithic Synthesis

This section adapts the classical, monolithic synthesis algorithm for supervisor derivation onto the presented formalism. In contrast to the common literature, the algorithm does not consider a plant model and specification separately from each other but carries out the necessary steps on a synthesis automaton that contains information about both. The

definitions and lemmas provided in the following will be recycled to define and proof the soundness of the incremental method as well.

As already mentioned above, the plant information provided in a synthesis automaton C can be insufficient to derive a supervisor. Since the prohibited language is refined on every composition according to Lemma 10, it is realizable if $\mathfrak{L}_{\mathcal{P}}(C) \cap \hat{\Sigma}_u^* = \emptyset$, i.e., if all purely uncontrollable sequences have turned out spurious. Otherwise it is unrealizable yet. In that case, adding more plant models can further reduce $\mathfrak{L}_{\mathcal{P}}$.

LEMMA 12 (Directedness of @) Let $C \in \mathfrak{C}, G \in \hat{\mathfrak{G}}$. Then holds

$$\mathfrak{L}_{\mathcal{P}}(C@G) \subseteq \mathfrak{L}_{\mathcal{P}}(C) \tag{5.1}$$
$$\mathfrak{L}^{\Pi}(C@G) \subseteq \mathfrak{L}^{\Pi}(C) \tag{5.2}$$
$$\mathfrak{L}^{\Pi}(C@G) \subseteq \mathfrak{L}^{\Pi}(G) \tag{5.3}$$

Proof. Line 5.1: $\mathfrak{L}_{\mathcal{P}}(C@G) = [\mathfrak{L}_{\mathcal{P}}(C) \cap \mathfrak{L}^{\Pi}(G)]\,\hat{\Sigma}^* = \mathfrak{L}_{\mathcal{P}}(C)\,\hat{\Sigma}^* \cap \mathfrak{L}^{\Pi}(G)\,\hat{\Sigma}^*$
$= \mathfrak{L}_{\mathcal{P}}(C) \cap \mathfrak{L}^{\Pi}(G)\,\hat{\Sigma}^* \subseteq \mathfrak{L}_{\mathcal{P}}(C)$
Line 5.2 and 5.3 follow directly from Lemma 10. □

Lemma 12 or, more precise, the situation where $\subseteq$ becomes $\subset$, is the key for the increase of permissiveness and realizability when plants are taken into consideration instead of using the specification barely. A sufficient criterion for that will be given in Section 5.8.3.

The monolithic synthesis algorithm is a fixpoint procedure. It operates in terms of succeeding iterations, called *steps*.

DEFINITION 21 (Synthesis Step, **sys**) A synthesis step is a mapping $\mathbf{sys} : \mathfrak{C} \to \mathfrak{C}$ and $\mathbf{sys}(Q, \Sigma^{\Pi}, f, P, q_0) = (Q', \Sigma^{\Pi}, f', P', q_0')$, where

- $Q' := \{q \in Q \mid P(q) \cap \hat{\Sigma}_u = \emptyset\}$
- $f' := f_{|Q' \times \hat{\Sigma} \to Q'}$ [11]
- $P' := Q' \to 2^{\hat{\Sigma}}$, s.th. $e \in P'(q) \iff e \in P(q) \cap \hat{\Sigma}_c$ or $f(q,e) \in Q \setminus Q'$
- $q_0' = \begin{cases} q_0, & q_0 \in Q' \\ \text{undefined} & \text{otherwise} \end{cases}$

A synthesis step is the standard way to "increase the controllability" of the resulting allowed language. It represents the adaptation of one iteration of the SCSL algorithm given by [23] to the presented formal model. In contrast to the latter, it operates on a single synthesis automaton. Therefore, for a given $C \in \mathfrak{C}$, $C' := \mathbf{sys}(C)$ represents again a valid synthesis automaton which can be composed with further plant components, a property which will be used for incremental synthesis.

We call an automaton *void* if it does not have an initial state. The composition of a void automaton with another automaton is always void too due to the absence of a proper initial state. For a void automaton A, $Q_{Ac(A)} = \emptyset$.

[11] The reduction of f to those $(q,e) \in Q' \times \hat{\Sigma}$ for which $f(q,e) \in Q'$

LEMMA 13 Let $C \in \mathfrak{C}$ and $C' := \mathbf{sys}(C)$.

1. If C' is void, then C is not realizable.
2. If C is realizable, then C' is realizable.

Proof. Line 1: $q_{0C'}! \implies q_{0C} \notin Q' \implies P(q_{0C}) \cap \hat{\Sigma}_u \neq \emptyset \implies \mathfrak{L}_{\mathcal{P}}(C) \cap \hat{\Sigma}_u^* \neq \emptyset$.
Line 2: By construction of **sys** holds for all $q \in Q_{C'}$ that $P'(q) \cap \hat{\Sigma}_u \subseteq P(q) \cap \hat{\Sigma}_u$. □

Since in the realizable case, all prohibited events either stay prohibited or become unreachable by P', **sys** is directed in the following sense.

LEMMA 14 (Directedness of **sys**) Let $C \in \mathfrak{C}$ be a realizable synthesis automaton. It holds

$$\begin{aligned} \mathfrak{L}_{\mathcal{P}}(C) &\subseteq \mathfrak{L}_{\mathcal{P}}(\mathbf{sys}(C)) \\ \mathfrak{L}^{\Pi}(C) &\supseteq \mathfrak{L}^{\Pi}(\mathbf{sys}(C)) \end{aligned}$$

Proof. Let $C' = \mathbf{sys}(C)$. By the construction of **sys** we see that $\forall q \in Q_{C'} : P_C(q) \subseteq P_{C'}(q)$ and $f_{C'}(q,e)! \Rightarrow f_C(q,e)!$. Further holds for all $q \in Q_C \setminus Q_{C'}$ (the eliminated states) and $q' \in Q_{C'}$ that $f_C(q',e) = q \Rightarrow e \in P_{C'}(q')$, thus the event leading to q has been forbidden. Hence a previously prohibited string $s\hat{e}e$ with $e \in P_C(f_C^*(s\hat{e}))$ stays forbidden as $\hat{e} \in P_{C'}(f_{C'}^*(s))$ and $\mathfrak{L}_{\mathcal{P}}$ is closed under continuation. From Definition 19 follow the propositions. □

When multiple synthesis steps are applied subsequently, a fixpoint is reached after finitely many steps. This is always the case since Q' shrinks monotonically and f', P', q_0' purely depend on Q', which is finite. Thus, if $Q' = Q$, then $f' = f, P' = P, q_0' = q_0$ and hence $Q_{\mathbf{sys}(\mathbf{sys}(C))} = Q_C$ and inductively $Q_{\mathbf{sys}(\ldots\mathbf{sys}(C))} = Q_C$.

Let $\mathbf{sys}^i(C) := \underbrace{\mathbf{sys}(\ldots\mathbf{sys}(}_{i \text{ times}} C)\ldots)$. Then for some $n \in \mathbb{N} : \mathbf{sys}^n(C) = \mathbf{sys}^{n+1}(C)$. We denote that fixpoint by $\mathbf{sys}^{\text{fix}}(C)$. By definition of Q' (Def. 21), we see that

$$\forall q \in Q_{\mathbf{sys}^{\text{fix}}(C)} : \; P_{\mathbf{sys}^{\text{fix}}(C)}(q) \cap \hat{\Sigma}_u = \emptyset$$

since if there was an uncontrollable, prohibited event left, **sys** would change the automaton and a fixpoint would not have been reached yet. This fact allows us to use $\mathbf{sys}^{\text{fix}}(C)$ as supervisor $S = (Q, f, P, q_0)$, where $(Q, \Sigma^{\Pi}, f, P, q_0) := \mathbf{sys}^{\text{fix}}(C)$, by simply dropping the permissiveness alphabet Σ^{Π}. For brevity we write $S := \mathbf{sys}^{\text{fix}}(C)$ and implicitly omit Σ^{Π}.

Adding plant models can increase but never decrease the effective permissiveness of the result of a synthesis step. The reason for that is that an additional plant model can potentially eliminate prohibitions of uncontrollable events, or at least do so in certain states (*state splitting*, cf. [23]). As a consequence, those prohibitions do not have to be considered by **sys** anymore.

THEOREM 15 (Correctness of synthesis) Let $G \in \hat{\mathfrak{G}}$ and $H \in \mathfrak{H}$ such that H is realizable on $\mathfrak{L}^{\Pi}(G)$. Then $S := \mathbf{sys}^{\text{fix}}(H@G)$ is a supervisor that is

- not void,

- H-safe w.r.t. L_{phy},
- H-safe and maximally permissive w.r.t. $\mathfrak{L}^{\Pi}(G)$.

Proof. Realizability: Assume that S was void. Choose i s.th. $C := \mathbf{sys}^{i}(H@G) \neq \mathbf{sys}(C) = \mathbf{sys}^{\text{fix}}(H@G) = S$. Then, by Lemma 13(1), C is not realizable. Given that H is realizable on $\mathfrak{L}^{\Pi}(G)$ we obtain from Lemma 13(2) that C is also realizable. Contradiction.

Safety: Follows directly from lemmas 11 and 14.

Maximal Permissiveness. To show: For all H-safe supervisors S' holds $\mathfrak{L}^{\Pi}(G) \setminus \mathfrak{L}_{\mathcal{D}}(S') \subseteq \mathfrak{L}^{\Pi}(G) \setminus \mathfrak{L}_{\mathcal{D}}(S)$.

Assume, there is a H-safe S' that is more permissive than S. Then exists a string $v \in \mathfrak{L}^{\Pi}(G) \setminus \mathfrak{L}_{\mathcal{D}}(S')$ and $v \notin \mathfrak{L}^{\Pi}(G) \setminus \mathfrak{L}_{\mathcal{D}}(S)$. Then, $v \in \mathfrak{L}_{\mathcal{D}}(\mathbf{sys}^{\text{fix}}(H@G))$. Let v_0 be the shortest forbidden prefix of v, i.e., $v_0 \in \mathfrak{L}_{\mathcal{D}}(\mathbf{sys}^{\text{fix}}(H@G))$ and i_0 the iteration of **sys** where the fixpoint has been reached first, i.e., $\mathbf{sys}^{i_0}(H@G) = \mathbf{sys}^{i_0+1}(H@G) \neq \mathbf{sys}^{i_0-1}(H@G)$. There are two cases that lead to v_0 being prohibited by S due to the construction of **sys**:

case 1 – $v_0 \in \mathfrak{L}_{\mathcal{P}}(H)$. In this case, S' is obviously not H-safe. Contradiction.

case 2 – $v_0 \notin \mathfrak{L}_{\mathcal{P}}(H)$. Then exists an event $e_0 \in \hat{\Sigma}_u$, s.th. $v_0 e_0 \in \mathfrak{L}^{\Pi}(G)$, i.e., inside G e_0 is considered possible after the legal string v_0. Further exists a previous iteration $i_1 \in \mathbb{N}, 0 \leq i_1 < i_0$, such that $v_0 e_0 \in \mathfrak{L}_{\mathcal{P}}(\mathbf{sys}^{i_1}(H@G))$ and $v_0 \notin \mathfrak{L}_{\mathcal{P}}(\mathbf{sys}^{i_1}(H@G))$ and $v_0 \in \mathfrak{L}_{\mathcal{P}}(\mathbf{sys}_{i_1+1}(H@G))$. In iteration $i_1 + 1$, **sys** prohibited v_0 to prevent the uncontrollable e_0 from occurring. Note, that this happens only when such e_0 exists inside $\mathfrak{L}_{\mathcal{P}}(G)$, otherwise only case 1 is applicable. Since e_1 is uncontrollable, S' cannot disable it either: $v_0 e_0 \notin \mathfrak{L}_{\mathcal{D}}(S')$.

Define $v_j := v_{j-1} e_{j-1}$. For v_1, we inductively obtain the same two cases again, as for every further v_j, as long as case 2 applies. Note, that $i_j = i_{j-1} - 1$ holds by construction for $j > 1$, since **sys** isolates a state containing an uncontrollable prohibition immediately in the next iteration. Due to that, eventually, $i_m = 0$ is reached, where case 2 collapses to $v_m e_m \in \mathfrak{L}_{\mathcal{P}}(\mathbf{sys}^{0}(H@G)) = \mathfrak{L}_{\mathcal{P}}(H@G) \subseteq \mathfrak{L}_{\mathcal{P}}(H)$. The last equality holds due to lemmas 8 and 10. Since $v_m e_m = v_0 e_0 \dots e_m$ and all $e_j \in \hat{\Sigma}_u$, S' can still not disable any of them, ergo $v_m e_m \notin \mathfrak{L}_{\mathcal{D}}(S')$ and S' is not H-safe. Contrad. □

The supervisor with the maximal achievable permissiveness can always be obtained by taking all available knowledge about the plant into account, thus by basing the synthesis on $C = H@(\|_{G \in \mathfrak{G}} G)$. This is commonly referred to as *monolithic synthesis*. Corollary 16 explicitly addresses the special case where the plant models together form a generator, thus define the *entire* possible behavior instead of an over-approximation.

Corollary 16 Let $H \in \mathfrak{H}$ be realizable on $\mathfrak{L}^{\Pi}(\hat{G}) = L_{phy}$. Then $S := \mathbf{sys}^{\text{fix}}(H@\hat{G})$ is a supervisor that is H-safe and maximally permissive.

Proof. Follows directly from Theorem 15. □

Algorithm 1 sketches the monolithic supervisor synthesis for the computation of $\mathbf{sys}^{\text{fix}}$ procedurally.

Algorithm 1 Monolithic supervisor synthesis without preemption, mon(C)

Input: $C = (Q, \Sigma^{\Pi}, f, P, q_0)$

```
 1: fixpoint := false
 2: while ¬ fixpoint do
 3:    fixpoint := true
 4:    for all q ∈ Q do
 5:       if P(q) ∩ Σ̂_u ≠ ∅ then
 6:          fixpoint := false;
 7:          ∀e remove (q, e) from f;
 8:          Q := Q \ {q};
 9:          for all (q', e') ∈ f^{-1}(q) do
10:             remove (q', e') from f;
11:             P(q') := P(q') ∪ {e'};
12:          end for
13:          remove all unreach. states from Q and their outg. trans. from f;
14:       end if
15:    end for
16: end while
17: if q_0 ∈ Q then
18:    return (Q, f, P, q_0);
19: else
20:    return "unrealizable!";
21: end if
```

The method is very straightforward. It searches for uncontrollable prohibited events and isolates all states that contain such by removing all incoming transitions and prohibiting the respective events instead. That corresponds to one synthesis step **sys**( ). To increase performance all states that have become unreachable can instantly be removed, too. A previously constructed reachability map that tracks, which state is reachable via which other states, helps. Alternatively, it is also possible to remove all unreachable states at the very end of the algorithm: However, it may take longer until a fixpoint is reached in that case. Line 10 ensures that consistency is not violated: The (q', e') transition is removed before an e' prohibition is inserted.

5.8.3 Incremental Synthesis

In order to get the best achievable permissiveness, it seems reasonable to take as many plant models into account as possible. However, the size of the intermediate state space increases dramatically by doing so. An intelligent approach thus should add a plant model only if it can possibly increase permissiveness in the current situation. Besides, it would do that as late as possible as synthesis steps can decrease the state space of intermediate results and, by that, the size of further compositions.

Lemma 9 allows us to replace the parallel composition of several plants by succeeding application products. For a delayed product computation, we must also show that it is legitimate to interleave product computations and synthesis steps. Moreover, a criterion for the cases where the latter does not affect the permissiveness of the result is needed.

The presented incremental synthesis automates the decision on which plant models are suitable to increase permissiveness and when. To that end, it is essential to recapitulate what permissiveness actually means. Basically, it describes the goal that nothing more is prohibited than actually required to ensure safety. In the following we present a necessary criterion for a plant automaton to increase the permissiveness of the supervisor resulting from the current synthesis automaton. Due to explicit prohibitions and the fact that everything not stated in a specification is implicitly allowed, it is very easy (and performant) to locate states where uncontrollable events are prohibited. When a prohibition is found, two cases have to be distinguished.

- The forbidden event is *uncontrollable* ($e \in \hat{\Sigma}_u$). This means the event cannot be disabled. Before the corresponding specification state is isolated it has first to be checked whether that state has logically to be split into several states (w. r. t. the plant) or if the undesired event is even impossible to occur.

- The forbidden event is *controllable* ($e \in \hat{\Sigma}_c$). In this case the event can simply be disabled without worrying about any plant models. There is no real increase of permissiveness possible that would not instantly violate the specification.

The following lemma summarizes that condition formally.

LEMMA 17 (Ineffective plant) Let $C = (Q_C, \Sigma_C^\Pi, f_C, P_C, q_{0C}) \in \mathfrak{C}$ be a synthesis automaton and $G = (Q_G, \Sigma_G, f_G, q_{0G}) \in \mathfrak{G}$ be a plant model. For a string $v \in \mathfrak{L}^\Pi(G)$ and a succeeding prohibited, uncontrollable event $e \in P_C(f_C^*(v)) \cap \hat{\Sigma}_u$ holds

$$e \notin \Sigma_G \ \wedge\ \mathbf{sys}(C@G) \text{ not void} \implies v \in \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C@G))$$

Proof. Let $v \in \mathfrak{L}^\Pi(G)$. From $e \notin \Sigma_G$ follows $ve \in \mathfrak{L}^\Pi(G)$. Further, $f_C^*(v)!$ implies $v \in \mathfrak{L}^\Pi(C)$ and $e \in P_C(f_C^*(v))$ implies $ve \in \mathfrak{L}_\mathcal{P}(C)$.
Together, that makes $ve \in \mathfrak{L}_\mathcal{P}(C@G) = [\mathfrak{L}_\mathcal{P}(C) \cap \mathfrak{L}^\Pi(G)]\hat{\Sigma}^*$ and $v \in \mathfrak{L}^\Pi(C@G) = \mathfrak{L}^\Pi(C) \cap \mathfrak{L}^\Pi(G)$. That means $f_{C@G}^*(v)!$ and $e \in P_{C@G}(f_{C@G}^*(v))$.
We define $C' := \mathbf{sys}(C@G)$ and $we' := v$. By the definition of **sys** (Def. 21) and $q_{0C'} \in Q'_C$ follows that $e' \in P_{C'}(f_{C'}^*(w))$[12] and $f_{C'}(f_{C'}^*(w), e')!$. Hence, from the definition of $\mathfrak{L}_\mathcal{P}$ (Def. 19), we obtain $we' = v \in \mathfrak{L}_\mathcal{P}(C') = \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C@G))$. □

Based on this result, the following lemma allows us to delay plant compositions.

LEMMA 18 (Postponed @-Product) Let $C = (Q_C, \Sigma_C^\Pi, f_C, P_C, q_{0C}) \in \mathfrak{C}$ be a synthesis automaton and $G = (Q_G, \Sigma_G, f_G, q_{0G}) \in \mathfrak{G}$ a plant model. Then it holds

$$\bigcup_{q \in Q_C} P_C(q) \cap \Sigma_G \cap \hat{\Sigma}_u = \emptyset \quad \implies \quad \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C@G)) = \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C)@G)$$

Proof. This proof shows the equality of both languages by showing the two inclusions $\subseteq, \supseteq$ separately. Note, that the "interesting", non-trivial case is the second case of "$\supseteq$", where the plant model may be "drawn outside" the **sys** function although **sys** actually changes the automaton.

"$\subseteq$": Let $v \in \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C@G))$. Without loss of generality, we assume for $we := v$ that $w \notin \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C@G))$, since $\mathfrak{L}_\mathcal{P}$ is closed under continuation on both sides of the equation and every word has a shortest forbidden prefix.
There are two cases for v in relation to $C@G$:

case 1: $v \in \mathfrak{L}_\mathcal{P}(C@G)$ (v was already forbidden before the synthesis step). Since its longest proper prefix w is not forbidden after the synthesis step, $w \notin \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C@G))$, e must be controllable, $e \in \hat{\Sigma}_c$. Due to the definition of $\mathfrak{L}_\mathcal{P}$ (Def. 19) follows $w \in \mathfrak{L}^\Pi(C@G) = \mathfrak{L}^\Pi(C) \cap \mathfrak{L}^\Pi(G)$ and $e \in P_C(f_C^*(w))$. As e is controllable, we get $w \in \mathfrak{L}^\Pi(\mathbf{sys}(C))$ and $e \in P_{\mathbf{sys}(C)}(f_{\mathbf{sys}(C)}^*(w))$. It follows $we \in \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C)) \cap \mathfrak{L}^\Pi(G) \subseteq \mathfrak{L}_\mathcal{P}([\mathbf{sys}(C) \cap \mathfrak{L}^\Pi(G)]\hat{\Sigma}^*) = \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C)@G)$.

case 2: $v \notin \mathfrak{L}_\mathcal{P}(C@G)$. Since $v \in \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C@G))$ holds $v \in \mathfrak{L}^\Pi(C) \cap \mathfrak{L}^\Pi(G)$, i.e., both C and G consider e possible to occur. There must be $e' \in \hat{\Sigma}_u$ s. th. $ve' \in \mathfrak{L}_\mathcal{P}(C) \cap \mathfrak{L}^\Pi(G)$. Due to e' being uncontrollable, $v \in \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C))$ and finally $v \in \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C)) \cap \mathfrak{L}^\Pi(G) \subseteq \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C)@G)$.

"$\supseteq$": Let $v \in \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C)@G)$. Again, w.l.o.g., we assume for $we := v$ that $w \notin \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C)@G)$. The fact that w is not in that prohibited language, but we is, leads to $f_{\mathbf{sys}(C)}^*(w)!, f_G^*(v)!$ and $e \in P_{\mathbf{sys}(C)}(f_{\mathbf{sys}(C)}^*(w))$. There are two cases again:

[12] Remember: $f^*(\varepsilon) := q_0$

case 1: $v \in \mathfrak{L}_\mathcal{P}(C)$ and $e \in \hat{\Sigma}_c$ (as above). It follows that $v \in \mathfrak{L}_\mathcal{P}(C@G)$ and, due to the directedness of **sys** (Lem. 14), $v \in \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C@G))$.

case 2: $v \notin \mathfrak{L}_\mathcal{P}(C)$. Since $v \in \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C))$, $v \in \mathfrak{L}^\Pi(C)$. There must be $e' \in \hat{\Sigma}_u$, s. th. $ve' \in \mathfrak{L}_\mathcal{P}(C)$, which means $f_C^*(v)!$ and $e' \in P_C(f_C^*(v))$. Further, from $f_G^*(v)!$, we know $v \in \mathfrak{L}^\Pi(G)$. Because we assume $\bigcup_{q \in Q_C} P_C(q) \cap \hat{\Sigma}_u \cap \Sigma_G = \emptyset$ to hold, e must be unknown to G, $e \notin \Sigma_G$. By Lemma 17 finally follows $v \in \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C@G))$.

"$v \rightarrow v\hat{\Sigma}^*$":

For both directions we have shown that for the respective shortest forbidden words $v \in \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C@G)) \implies v \in \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C)@G)$ and vice versa. Since $\mathfrak{L}_\mathcal{P}$ is closed under continuation that also holds for all $vw, w \in \hat{\Sigma}^*$ in both directions, which lifts the proof from the shortest forbidden prefixes to arbitrary strings. Hence both languages are equal. □

Note that the "⊆" direction of the proof does not make use of the condition $\bigcup_{q \in Q_C} P_C(q) \cap \hat{\Sigma}_u \cap \Sigma_G = \emptyset$, which means that $\mathfrak{L}_\mathcal{P}(\mathbf{sys}(C@G)) \subseteq \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C)@G)$ holds in general.

Lemma 18 unveils that it makes no difference whether a plant is applied before a synthesis step or afterwards as long as the set of uncontrollable, prohibited events that are shared with the plant, $\bigcup_{q \in Q_C} P_C(q) \cap \Sigma_G \cap \hat{\Sigma}_u$, is empty. However, the performance of the delayed product calculation can benefit from the fact that each synthesis step (except the fixpoint is reached) shrinks the state space and the number of transitions. When we interpret $\mathbf{sys}(C) := C'$ we can apply the lemma again, $\mathfrak{L}_\mathcal{P}(\mathbf{sys}(C')@G) = \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C'@G)) = \mathfrak{L}_\mathcal{P}(\mathbf{sys}(\mathbf{sys}(C@G)))$, as long as the condition holds for $Q_{C'}$ and $P_{C'}$ too.

If the product with a plant model has been repeatedly delayed and a fixpoint, i.e., a valid supervisor has been found before the plant has been used, it can be completely neglected.

Lemma 19 (Early Fixpoint) Let $C = (Q_C, \Sigma_C^\Pi, f_C, P_C, q_{0C}) \in \mathfrak{C}$ be realizable and $G = (Q_G, \Sigma_G, f_G, q_{0G}) \in \mathfrak{G}$. Let further $\bigcup_{q \in Q_C} P_C(q) \cap \Sigma_G \cap \hat{\Sigma}_u = \emptyset$ and $C' := \mathbf{sys}(C)$. Then

$$\forall q \in Q_{C'} : P_{C'}(q) \subseteq \hat{\Sigma}_c \quad \implies \quad L_{phy} \setminus \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C)) = L_{phy} \setminus \mathfrak{L}_\mathcal{P}(\mathbf{sys}(C@G))$$

Proof. Follows directly from Lemmas 18 and 11. □

The presented check of whether a plant model G is a reasonable candidate to improve permissiveness under the assumption that state q of a prohibition $P(q)$ is reachable (i.e., $ve \in L_{phy}$, $q = f^*(q_0, v)$ exists) fits perfectly into the backwards-orientated standard synthesis algorithm: The procedure of repeatedly executing the steps

1. search for permissiveness-increasing candidates (plants G with $e \in \Sigma_G$ for some forbidden e)
2. @-product calculation
3. single synthesis step

results in an *incremental synthesis technique* that computes product automata only if reasonable and as late as possible. Additionally, the interleaved synthesis step shrinks the state spaces of intermediate results.

The pseudo code in Algorithm 2 describes the incremental synthesis procedure in detail. It starts with a pure specification H and extracts all uncontrollable prohibitions. For each respective event $e \in \hat{\Sigma}_u$ it searches for all available plant models $G_i \in \mathfrak{G}$ whose alphabets contain that event ($e \in \Sigma_{G_i}$) and applies the specification to all of them: $C_1 := H@G_1@\ldots@G_n$. After that step there is no plant left that could increase permissiveness at the currently considered states. Next, for all uncontrollable prohibitions which remain in C_1 one synthesis step is executed. Since this might introduce new uncontrollable prohibitions, the procedure is looped until a fixpoint is found. The set $\tilde{G}_{remain}$ ensures that no plant model is used twice. Doing so would have no effect on the result. Still, it would dramatically increase the runtime. In the current tool implementation, the plants for product calculation are picked in arbitrary order (line 15). In the future, heuristics for an intelligent selection order could improve this algorithm's performance. In that case, even an event-wise iteration over all three steps could be reasonable. The central question in that context is typically whether a product calculation and subsequent synthesis step enlarge or reduce the size of the state space. While Step 2 can have either effect, Step 3 always decreases it. The synthesis step itself is carried out in lines 20–31, where all states containing uncontrollable, prohibited events are removed and their incoming transitions are transformed into prohibitions respectively. The absence of such prohibitions is also the fixpoint criterion.

Theorem 20 (Correctness of Incremental Synthesis) Let $H \in \mathfrak{H}$ be a specification that is realizable on $\mathfrak{L}^{\Pi}(\mathfrak{G})$. The supervisor $S = \mathbf{inc}(H, \mathfrak{G})$ is

- not void,
- H-safe w.r.t. L_{phy},
- H-safe and maximally permissive w.r.t. $\mathfrak{G}$.

Proof. Realizability: Assume S was void. Since a composition is only void if one of the composed automata is void and the plants $G \in \mathfrak{G}$ are not void, there must have been $C \in \mathfrak{C}$ s.th. C is not void and $S = \mathbf{sys}(C)$. Then exists $e_u \in P(q_{0C}) \cap \hat{\Sigma}_u$. By construction of **inc** (ll. 9–11), all G where such an e_u exists in Σ_G have been considered in C in terms of previous compositions. Hence (Lemma 13), C is unrealizable. As C is either already the specification H or a result of interleaved synthesis steps and compositions between H and the $G \in \mathfrak{G}$, where none of these operations can introduce unrealizability (thanks to lemmas 12 and 13), H is not realizable on $\mathfrak{L}^{\Pi}(\mathfrak{G})$. Contradiction.

Safety: We have to show that $L_{phy} \setminus \mathfrak{L}_{\mathcal{D}}(S) \subseteq \mathfrak{L}_{\mathcal{S}}(H)$. The algorithm performs two types of operations on H. Products with plants and synthesis steps. The order varies but plays no role for safety. Let C be an intermediate result of Algorithm 2 at Line 4, i.e., at the very beginning of a loop iteration. From Lemma 11 (safety of @-Product) we know that for all $G \in \mathfrak{G}$: $L_{phy} \setminus \mathfrak{L}_{\mathcal{P}}(C@G) = L_{phy} \setminus \mathfrak{L}_{\mathcal{P}}(C)$. Lemma 14 (monotonicity of **sys**) states $\mathfrak{L}_{\mathcal{P}}(C) \subseteq \mathfrak{L}_{\mathcal{P}}(\mathbf{sys}(C))$. It follows $L_{phy} \setminus \mathfrak{L}_{\mathcal{P}}(\mathbf{sys}(C)) \subseteq L_{phy} \setminus \mathfrak{L}_{\mathcal{P}}(C)$. Hence, all for all intermediate results C holds $L_{phy} \setminus \mathfrak{L}_{\mathcal{P}}(C) \subseteq L_{phy} \setminus \mathfrak{L}_{\mathcal{P}}(H)$ and consequently for the final supervisor S: $L_{phy} \setminus \mathfrak{L}_{\mathcal{D}}(S) \subseteq L_{phy} \setminus \mathfrak{L}_{\mathcal{P}}(H)$.

Algorithm 2 Incremental supervisor synthesis without preemption, **inc**$(H, \mathfrak{G})$

Input: $H = (Q, f, P, q_0)$, $\mathfrak{G}$

fixpoint := **false**
$\tilde{G}_{remain} := \mathfrak{G}$
$C :=$ convertToSynthAut(H)
while ¬ fixpoint **do**
  fixpoint := **true**

  // Step 1: Gather plants to consider in $\tilde{G}$
  $\tilde{G} := \emptyset$
  for all $q \in Q_C$ **do**
    $\tilde{G} := \tilde{G} \cup \{G \in \tilde{G}_{remain} \mid P(q) \cap \hat{\Sigma}_u \cap \Sigma_G \neq \emptyset\}$
  end for
  $\tilde{G}_{remain} := \tilde{G}_{remain} \setminus \tilde{G}$

  // Step 2: Compute application products
  for all $G \in \tilde{G}$ **do**
    $C := C@G$
  end for

  // Step 3: Perform one synthesis step
  for $q \in Q_C$ **do**
    if $P(q) \cap \hat{\Sigma}_u \neq \emptyset$ **then**
      fixpoint := **false**
      $Q := Q \setminus \{q\}$
      $\forall e$ remove (q, e) from f
      for all $(q', e') \in f^{-1}(q)$ **do**
        remove (q', e') from f
        $P(q') := P(q') \cup \{e\}$
      end for
      remove all unreachable states from Q and their respective outgoing transitions from f.
    end if
  end for
end while
if $q_0 \in Q$ **then**
  return (Q, f, P, q_0);
else
  return "unrealizable!";
end if

Permissiveness: From Lemma 15 we know that the monolithic synthesis computes the maximally permissive supervisor for H w.r.t. $\mathfrak{G}$ by

$$C = \mathbf{sys}^{\text{fix}}(H@(\mathop{\Big\|}_{G\in\mathfrak{G}} G)) \quad \text{and finally } S := C$$

Corresponding to Lemma 9 that is equivalent to

$$\mathbf{sys}^{\text{fix}}(H@G_1@\ldots@G_n), \quad \{G_1,\ldots,G_n\} := \mathfrak{G}$$

Lemma 18 allows us to draw plants outside a synthesis step if $\forall q \in Q_C : P_C(q) \cap \Sigma_G \cap \hat{\Sigma}_u = \emptyset$ holds. $(\star)$
For $H@G_1@G_2$, that means that in order to check the eligibility of G_2 for being drawn outside it suffices to check the prohibitions of H, i.e., iterate over $q \in Q_H$ and check $P_H(q)$, since G_1 does not introduce new prohibited events. Accordingly, that holds for all further G_is. That checks are performed in ll. 9–11 where the plants that stay inside the **sys**() braces are gathered. The remaining plant models (l. 12) stay in $\tilde{G}_{remain}$ ergo they are ensured to be checked in the next iteration again. Lines 20–31 implement the synthesis step function **sys**, cf. Definition 21. The result of the first iteration is $C' = \mathbf{sys}(H@G_{11}@\ldots@G_{1n})$ where $G_{11},\ldots,G_{1n}$ are those plants where $(\star)$ does not hold. All subsequent iterations follow the same principle, except H is now replaced by C' and only the plant inside $\tilde{G}_{remain}$ are checked for $(\star)$ again. Lemma 19 allows to terminate as soon as a fixpoint is found. The sufficient condition for that fixpoint is $\forall q \in Q_C : P(q) \cap \hat{\Sigma}_u = \emptyset$, which is checked in Line 21. Finally, it is returned as the supervisor. □

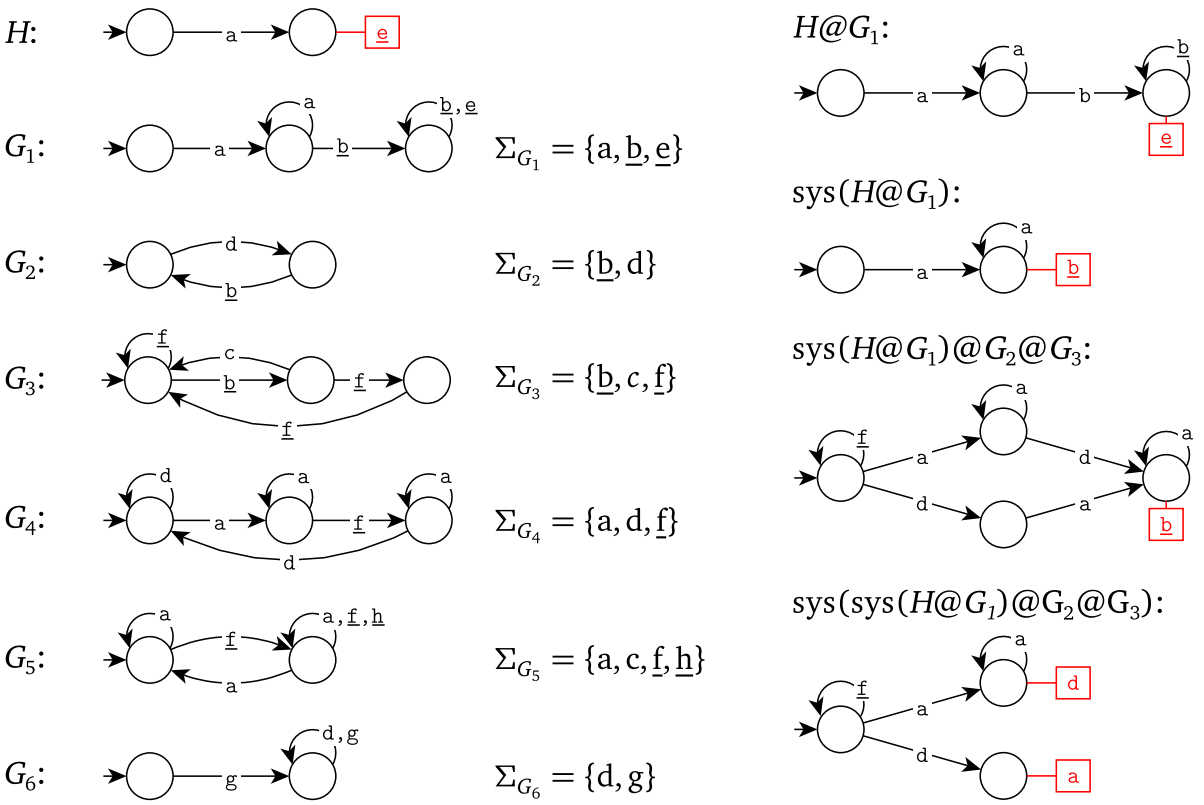


Figure 5.18: Example for incremental synthesis. Only three plant automata are required to obtain a maximally permissive supervisor.

Example 5.17. Let $\mathfrak{H} = \{H\}$, $\mathfrak{G} = \{G_1, \ldots, G_6\}$, as depicted on the left-hand side of Figure 5.18, $\underline{\mathtt{b}}, \underline{\mathtt{e}}, \underline{\mathtt{f}} \in \hat{\Sigma}_u$, $\mathtt{a}, \mathtt{c}, \mathtt{d}, \mathtt{g} \in \hat{\Sigma}_c$, and L_{phy} unknown. The first iteration starts with H. The only uncontrollable forbidden event is $\underline{\mathtt{e}}$. Accordingly, all plants G where $\underline{\mathtt{e}} \in \Sigma_G$ are gathered in $\tilde{G}$, i.e., $\tilde{G} = \{G_1\}$. All other automata have no knowledge about when $\underline{\mathtt{e}}$ is or is not possible. Then, $H@G_1$ is computed. It is shown on the right-hand side. Since the resulting synthesis automaton contains a prohibition of an uncontrollable event, $\underline{\mathtt{e}}$, a fixpoint is not yet reached and the algorithm invokes a synthesis step, yielding $\mathbf{sys}(H@G_1)$.

In the next iteration we obtain $\tilde{G} = \{G_2, G_3\}$ since $b \in G_2, b \in G_3$ and $b \notin G_4 \cup G_5 \cup G_6$ and the composition $\mathbf{sys}(H@G_1)@G_2@G_3$ is computed. After the next synthesis step, all existing prohibitions address controllable events, namely $\mathtt{a}$ and $\mathtt{d}$. Hence, a fixpoint is reached. Thanks to Theorem 20, $S := \mathbf{sys}(\mathbf{sys}(H@G_1)@G_2@G_3)$ is the maximally permissive supervisor. Note that in the final result the $\mathtt{a}$-loop plays no role anymore and could be deleted. However, $G_1 \parallel G_2 \parallel G_3 \parallel G_4 \parallel G_5 \parallel G_6$, which consists of 26 states and 91 transitions in this example, does not have to be computed to obtain S. △

Comparison to Modular Synthesis by Åkesson et al. In Section 5.4, it was already mentioned that the presented incremental method appears at the first glance similar to the modular approach by Åkesson et al. [3]. However, there are significant differences. The latter picks the relevant plant models using the transitive hull of the function *Dep*, defined

by "$Dep(F, \Sigma') = \{F_i \mid \exists \sigma \in \Sigma' \cap \Sigma^{F_i}\}$" [3]. Transferred to the formalisms of this chapter, this would correspond to $Dep(\mathfrak{G}, \Sigma) = \{G \in \mathfrak{G} \mid \Sigma_G \cap \Sigma \neq \emptyset\}$. First, Σ is initialized to the set of *all uncontrollable events* of the specification[13] and then incrementally extended by $\{e \in \Sigma_G \mid G \in Dep(\mathfrak{G}, \Sigma)\} \cap \hat{\Sigma}_u$ until a fixpoint Σ_{fix} is reached. The specification is finally composed with all plants in $Dep(\mathfrak{G}, \Sigma_{\text{fix}})$ and a monolithic synthesis is performed on the result.

Consider again Example 5.17 and the automata on the left-hand side of Figure 5.18. The transitive closure of *Dep* would involve G_1, G_2 and G_3 because of e and b as well, but also G_4 and G_5 as the uncontrollable event f is shared amongst G_3, G_4 and G_5. This *Dep*-based criterion for the relevance of a plant is strictly coarser than the one presented in Lemma 18.

Åkesson et al. over-approximate the set of plants that *during synthesis could eventually become relevant*. The presented incremental method instead postpones this decision until an event is actually prohibited and performs compositions lazily.

A comparison of the runtime performance between the monolithic, the modular and the incremental method is given in Section 7.5.

5.8.4 Conditions

Conditions relate the transitions and prohibitions of an automaton to the states of other automata. Within the formal model, the scope of what is addressable is given by the supervisory control environment E. In the following, first conditions are formalized. Thereafter, the definitions for the composition operators $\|$ and @ on conditional automata are given.

5.8.4.1 Formalization, Evaluation and Refinement

Although the implementation allows to reference states of already calculated compositions, this ability can be neglected here as a reference to a composite state (q_1, q_2) can always be replaced by the conjunction $q_1 \wedge q_2$. $\hat{Q}_E$ contains all atomic states.

Definition 22 ($\hat{Q}_E$) Let $E = (\hat{\Sigma}, L_{phy}, \mathfrak{H}, \mathfrak{G})$. The *global state domain* is defined by $\hat{Q}_E := \{q \in Q_A \mid A \in \mathfrak{G} \cup \mathfrak{H}\}$

Using the elements of $\hat{Q}_E$ as variables, conditions can be defined.

Definition 23 (from [42]) A *condition* over $\hat{Q}_E$ is a propositional formula with the syntax

$$c \;\rightarrow\; c \vee c \mid c \wedge c \mid \neg c \mid q \mid \mathbf{t} \mid \mathbf{f}$$

where $q \in \hat{Q}_E$. $\mathcal{P}_{\hat{Q}_E}$ denotes the set of all such conditions over $\hat{Q}_E$.

In order to resolve or refine conditions, they need be evaluated on the state where the corresponding transition or prohibition is located. This can have three possible outcomes: $\top$ (*true*), $\bot$ (*false*) and $\boxminus$ (*maybe*). The latter is the case if the current state still allows both evaluations as more information is needed for a definite result. Thus, conditions form a *three-valued logic*.

[13] In [3] the specification is called *"uncontrollable supervisor"*

DEFINITION 24 For a composite state $q = (q_1, \ldots, q_n)$ the set of its atomic substates is defined by $\mathsf{atom}(q) := \{q_1, \ldots, q_n\}$. For an atomic state $q \in \hat{Q}_E$, $\mathsf{atom}(q) := \{q\}$.

DEFINITION 25 (Evaluation, Refinement adapted from [42]) The evaluation of a condition c on a state p, denoted by $[\![c]\!]p$, is defined by:

$$[\![\mathtt{t}]\!]p := \top,\ [\![\mathtt{f}]\!]p := \bot,$$

$$[\![\mathtt{q}]\!]p := \begin{cases} \top & \mathtt{q} \in \mathsf{atom}(p) \\ \bot & \exists p' \in \mathsf{atom}(p), p' \neq q \text{ and } p, q \text{ are in the same automaton} \\ \boxminus & \text{otherwise} \end{cases},$$

$$[\![\neg c]\!]p := \begin{cases} \top & [\![c]\!]p = \bot \\ \bot & [\![c]\!]p = \top \\ \boxminus & \text{otherwise} \end{cases},$$

$$[\![c_1 \vee c_2]\!]p := \max\{[\![c_1]\!]p, [\![c_2]\!]p\},$$

$$[\![c_1 \wedge c_2]\!]p := \min\{[\![c_1]\!]p, [\![c_2]\!]p\},$$

where $\bot < \boxminus < \top$. The refinement of a condition $c \in \mathcal{P}_{\hat{Q}_E}$ on a state p is defined by the refinement function ref_p,

$$\mathrm{ref}_p(\mathtt{q}) = \begin{cases} \mathtt{t} & [\![q]\!]p = \top \\ \mathtt{f} & [\![q]\!]p = \bot \\ \mathtt{q} & \text{otherwise} \end{cases}, \qquad \mathrm{ref}_p(\neg c) = \begin{cases} \mathtt{f} & [\![c]\!]p = \top \\ \mathtt{t} & [\![c]\!]p = \bot \\ \neg \mathrm{ref}_p(c) & \text{otherwise} \end{cases},$$

$$\mathrm{ref}_p(c_1 \vee c_2) = \begin{cases} \mathtt{t} & [\![c_1 \vee c_2]\!]p = \top \\ \mathtt{f} & [\![c_1 \vee c_2]\!]p = \bot \\ \mathrm{ref}_p(c_1) \vee \mathrm{ref}_p(c_2) & \text{otherwise} \end{cases},$$

$$\mathrm{ref}_p(c_1 \wedge c_2) = \begin{cases} \mathtt{t} & [\![c_1 \wedge c_2]\!]p = \top \\ \mathtt{f} & [\![c_1 \wedge c_2]\!]p = \bot \\ \mathrm{ref}_p(c_1) \wedge \mathrm{ref}_p(c_2) & \text{otherwise} \end{cases}.$$

5.8.4.2 Compositions

Conditions allow multiple transitions of the same event at the same state. Hence, the definition set of the transition function from Definition 13 need to be supplemented with the condition. In order to achieve a more compact definition for the composition of two conditional plants, the transition function f is converted to a relation $T \subseteq Q \times \Sigma \times \mathcal{P}_{\hat{Q}_E} \times Q$ [42]. Each element $t = (q_1, e, c, q_2) \in T$ represents one transition from q_1 to q_2 on the event e and the condition c. Unconditional transitions can be realized using $c = \mathbf{t}$. The transition function f of an entirely unconditional plant is represented by $T = \{(q_1, e, \mathbf{t}, q_2) \mid f(q_1, e) = q_2\}$. $\tilde{Ac}$ is a modified version of the accessible function Ac which additionally removes all transitions (q, e, c, q') and prohibitions (q, e, c) where $c = \mathbf{f}$

Definition 26 (Plant composition, from [42]) Let $G^1 = (Q^1, \Sigma^1, T^1, q_0^1)$ and $G^2 = (Q^2\Sigma, T^2, q_0^2)$ two plant automata. Their composition is defined by $G^1||G^2 = \tilde{Ac}(Q^1 \times Q^2, \Sigma^1 \cup \Sigma^2, T', (q_0^1, q_0^2))$. The new transition relation T' is the smallest set such that for all $q_1^1, q_2^1 \in Q^1,\ q_1^2, q_2^2 \in Q^2,\ e \in \Sigma^1 \cup \Sigma^2,\ c^1, c^2 \in \mathcal{P}_{\hat{Q}_E}$ the following three implications are satisfied

$$
\begin{aligned}
(q_1^1, e, c^1, q_2^1) \in T^1, e \notin \Sigma^2 &\implies ((q_1^1, q_1^2), e, \mathrm{ref}_{q_1^2}(c^1), (q_2^1, q_1^2)) \in T', \\
(q_1^2, e, c^2, q_2^2) \in T^2, e \notin \Sigma^1 &\implies ((q_1^1, q_1^2), e, \mathrm{ref}_{q_1^1}(c^2), (q_1^1, q_2^2)) \in T', \\
(q_1^1, e, c^1, q_2^1) \in T^1 \text{ and } (q_1^2, e, c^2, q_2^2) \in T^2 &\implies ((q_1^1, q_1^2), e, \mathrm{ref}_{q_1^2}(c^1) \wedge \mathrm{ref}_{q_1^1}(c^2), (q_2^1, q_2^2)) \in T'.
\end{aligned}
$$

$\tilde{Ac}$, the *accessible* function for automata with transition relations, is defined by $\tilde{Ac}(Q, \Sigma^\Pi, T, P, q_0) = (Q', \Sigma^\Pi, T', P', q_0)$, where

- $\tilde{T}$ is the largest set so that $\tilde{T} \subseteq T$ and for all $(q, e, c, q') \in \tilde{T}$ exist $(q_0, e_1, c_1, q_1), \ldots, (q_n, e_n, c_n, q_{n+1}) \in \tilde{T}$ and $q = q_{n+1}$.
- $Q' = \{q \in Q \mid \exists (q', e, c, q) \in \tilde{T}\}$
- $T' = \tilde{T} \cap (Q' \times \hat{\Sigma} \times c \times Q')$

The first two lines cover the case where the event is only known by one of the two plants. In each case, the condition is refined on the other one. The third implication handles events which are in both plants' alphabets. The conditions are refined on each other's states and conjuncted as in this case both must hold for the transition to be valid. Note that this line creates one element in T' for each combination of e-transitions in T^1 and T^2 and thus for their conditions. Example 5.14 on Page 79 demonstrates the composition of two plants.

The application product needs to consider that the plant contract does not hold for specifications, i.e., an event not contained in the synthesis automaton's plant alphabet Σ^Π can occur in absence of a transition. In a conditional setting, this requires that the conditions of competing transitions all evaluate to false as otherwise one of those would be taken. Hence, these conditions are state- and event-wise gathered in $\mathrm{cc} : Q \times \hat{\Sigma} \rightharpoonup 2^{\mathcal{P}_{\hat{Q}_E}}$ to finally form a *counter condition* for staying in the current state.

Definition 27 (Application product, from [42], extended) Let $C = (Q^C, \Sigma^{\Pi C}, T^C, P^C, q_0^C)$ be a synthesis automaton and $G = (Q^G, \Sigma^G, T^G, q_0^G)$ a plant model. The application of G onto C is given by $C@G = \tilde{Ac}(Q^C \times Q^G, \Sigma^{\Pi C} \cup \Sigma^G, T', P', (q_0^C, q_0^G))$. T', P' and $\mathrm{cc}(q, e)$ are the smallest sets such that for all globally known events $e \in \hat{\Sigma}$:

- if $e \notin \Sigma^G$ then
  - for all $(q_1^C, e, c^C, q_2^C) \in T^C, q^G \in Q^G$: $((q_1^C, q^G), e, \mathrm{ref}_{q^G}(c^C), (q_2^C, q^G)) \in T'$
  - for all $(q^C, e, c^P) \in P^C, q^G \in Q^G$: $((q^C, q^G), e, \mathrm{ref}_{q^G}(c^P)) \in P'$
- if $e \in \Sigma^G$, then for all $(q_1^G, e, c^G, q_2^G) \in T^G$ holds:
  - for all $(q_1^C, e, c^C, q_2^C) \in T^C$:
    $((q_1^C, q_1^G), e, \mathrm{ref}_{q_1^G}(c^C) \wedge \mathrm{ref}_{q_1^C}(c^G), (q_2^C, q_2^G)) \in T'$ and $\mathrm{ref}_{q_1^G}(c^C) \in \mathrm{cc}(q_1^C, e)$

- for all $(q^C, e, c^P) \in P^C$:
  $\mathrm{ref}_{q_1^G}(c^P) \in \mathrm{cc}(q^C, e)$ and $((q^C, q_1^G), e, \mathrm{ref}_{q_1^G}(c^P) \wedge \mathrm{ref}_{q^C}(c^G)) \in P'$
- if $e \notin \Sigma^{\Pi C}$ then for all $q^C \in Q^C$:
  $((q_1^C, q_1^G), e, c_{\mathrm{cc}}, (q_1^C, q_2^G)) \in T'$, where $c_{\mathrm{cc}} := c^G \wedge \bigwedge_{c \in \mathrm{cc}(q_1^C)} \neg c$

As for the unconditional case (definitions 17 and 20), the composition operator can be implemented in an exploratory way, traversing both automata from their respective initial states. Thus, *Ac* is implicitly applied as unreachable states are automatically avoided. Further, as soon as a condition evaluates to $\bot$, the corresponding transition or prohibition is removed, finally yielding the semantics of $\tilde{Ac}$. Note that consistency of C is crucial for Definition 27 to yield the expected result [42], i.e., that no prohibition and transition with non-excluding conditions are defined at the same state for the same event.

Figure 5.16 on Page 81 illustrates the above definition for a specification, i.e., $\Sigma^{\Pi C} = \emptyset$. The transitions to 1B and 1C involve the counter condition $\neg \mathtt{c}_1 \wedge \neg \mathtt{c}_2 \wedge \neg \mathtt{c}_\mathrm{p}$. In case of $e \in \Sigma^{\Pi C}$, these two transitions would be dropped.

5.8.5 Preemption

Adding preemption to the formal model requires several adaptations. To not overload the formalisms, this section considers only straight, unconditional transitions and prohibitions using the functional notion of f and P from Section 5.8.1. For the monolithic method this is unproblematic as all conditions are assumed to be eliminated until synthesis is invoked. Remember that those transitions which are enforced-by-specification are technically possible but their use is discouraged, cf. Section 6.4.2. Hence, they will not be covered in this section either. Due to the missing synchronization of enforced events amongst several supervisors in the runtime framework (cf. Section 6.4.2), the following formalisms are based on the assumption that only one supervisor is synthesized. Hence, all given specifications $H_1, \ldots, H_n$ need be composed to one $H \in \mathfrak{H}$ first.

5.8.5.1 Extensions of the Model

In order to allow preemption, $\hat{\Sigma}$ is extended by the class of enforceable events, $\hat{\Sigma}_f$. While specifications and plants need not be touched, synthesis automata must be able to distinguish between enforced and not (yet) enforced transitions.

Definition 28 (Preemptive synthesis automaton) A preemptive synthesis automaton is a tuple $(Q, \Sigma^\Pi, f, \gamma, P, q_0)$ where the partial function $\gamma : Q \rightharpoonup \hat{\Sigma}_f$ defines which event $e_f = \gamma(q)$ is enforced at q. The rest is analog to Definition 18.

For all states q, where no event is enforced yet, $\gamma(q)$ is undefined. Throughout this section, $\mathfrak{C}_\delta$ addresses the set of all preemptive synthesis automata.

As preemption aims for more permissive, yet safe, solutions, it is necessary to relax the definition of safety in a sense that considers preempted events as successfully averted. But first, it needs to be declared when an event is regarded as preempted, based on the second aspect of the *preemption contract* (cf. Section 5.5.2)

Definition 29 (Preemption contract, enforcement completeness) Let L be a language and B a set of plants that are correct w.r.t. L. B is *enforcement complete w.r.t.* L iff for all $s \in L, e \in \hat{\Sigma}_f$ holds

$$se \in \bigcap_{G \in B} \mathfrak{L}^{\Pi}(G) \implies se \in L.$$

The preemption contract claims that $\mathfrak{G}$ is enforcement complete w.r.t. L_{phy}, i.e., all enforceable events are realizable.

Definition 30 (δ) Let $C = (Q, \Sigma^{\Pi}, f, \gamma, P, q_0)$ a synthesis automaton. The events *preempted* in a state q are given by $\delta_C : Q \to 2^{\hat{\Sigma}_u \cup \hat{\Sigma}_f}$ where

$$\delta_C(q) := \begin{cases} \hat{\Sigma}_u \cup \hat{\Sigma}_f \setminus \gamma(q) & \text{if } \gamma(q)! \\ \emptyset & \text{if } \gamma(q)\not! \end{cases}$$

Note that all unenforced enforceable events are treated as preempted. In a 1-supervisor setting, that makes sense as those events cannot occur anymore. Practically, however, a second supervisor which is executed in parallel could still enforce one of them as long as it is not prohibited.

The consistency requirement is extended by the claims that first only available events can be enforced, i.e., for all $q \in Q$ holds $\gamma(q)! \implies f(q, \gamma(q))!$ and second there are no transitions on preempted events, i.e., for all $q \in Q, e \in \delta_C(q) : f(q, e)\not!$. The operators and functions defined in the following always yield consistent results on consistent inputs.

Definition 31 (Preempted Language $\mathfrak{L}_\delta$) Let $C = (Q, \Sigma^{\Pi}, f, \gamma, P, q_0) \in \mathfrak{C}_\delta$ a preemptive synthesis automaton. The *language preempted by* C is defined by

$$\mathfrak{L}_\delta(C) := \{ses' \mid f^*(q_0, s)!,\ e \in \delta_C(f^*(q_0, s)),\ s' \in \hat{\Sigma}^*\}$$

Like $\mathfrak{L}_{\mathcal{P}}$, $\mathfrak{L}_\delta$ is obviously closed under continuation by construction. Note that $\mathfrak{L}_\delta(C) = \emptyset$ for a non-preemptive C since γ is undefined on all states and δ thus empty on all states.

The application product needs slight modification to fit preemptive automata.

Definition 32 (Supplement of application Product for $\mathfrak{C}_\delta$) Let $C \in \mathfrak{C}_\delta$ and $G \in \mathfrak{G}$. $C@G = Ac(Q, \Sigma^{\Pi}, f, \gamma, P, q_0) \in \mathfrak{C}_\delta$ is defined as in Definition 20, except

- $\gamma(q_C, q_G) = \begin{cases} \gamma_C(q_C) & \text{if } f_G^*(q_G, \gamma_C(q_C))! \\ \text{undefined} & \text{otherwise} \end{cases}$
- $f((q_C, q_G), e) = \begin{cases} \vdots & \\ (q_C, f_G(q_G, e)) & \cdots \wedge e \notin \delta_C(q_C) \quad \text{(add this constraint to 2nd case)} \\ \vdots & \end{cases}$

Definition 32 is straightforward: An enforced event stays enforced if possible on the plant and preempted events are removed from the transition function.

For the sake of brevity, the formal definition of a preemptive supervisor is omitted. It is basically a supervisor in the sense of Definition 7 (no uncontrollable prohibitions, no

alphabet) augmented with the enforcement function γ as in Definition 28. The functions of preempted events δ and the preempted language $\mathfrak{L}_\delta$ are defined exactly as on synthesis automata (definitions 30 and 31).

Based on these foundations, safety can be defined for preemptive supervision.

DEFINITION 33 (Safety) Let $H \in \mathfrak{H}$ a specification, S a supervisor and $L \subseteq \hat{\Sigma}^*$ a language. S is called H-safe w.r.t. L iff $L \setminus (\mathfrak{L}_\mathcal{D}(S) \cup \mathfrak{L}_\delta(S)) \subseteq \mathfrak{L}_\mathcal{S}(H)$

While for safety only the absence of undesired strings is relevant, be it via disabling or preemption, permissiveness is more challenging to grasp. Of course, a supervisor which disables more events than necessary shall still be conceived worse than one that is less restrictive in these regards. Further, it is obvious that preemption of an event e in a state q is more permissive than isolating q by prohibiting all incoming transitions. When there are multiple enforceable events available in q which all would preempt e, the situation is less clear. Indeed, since optimization and nonblockingness are not addressed, one of them may be picked arbitrarily. The consequence is that more than one legit, maximally permissive solution may exist. The following relation is introduced to capture this aspect when comparing two languages.

DEFINITION 34 ($\subseteq_F, \supseteq_F$) Let $L_1, L_2 \subseteq \hat{\Sigma}^*$ two languages. L_1 is a *sublanguage modulo enforcement* of L_2, denoted by $L_1 \subseteq_F L_2$ iff for all $s \in L_1$ holds

- $s \in L_2$ or
- $\exists s', s'' \in \hat{\Sigma}^*,\ e, e' \in \hat{\Sigma}_f$ s.th. $s = s'es'',\ s'e \in L_1 \setminus L_2$ and $s'e' \in L_2 \setminus L_1$.

The second criterion allows a string of L_1 to be not in L_2 if it contains an enforced event and there is a string with the same prefix in L_2 where an alternative event is enforced instead.

DEFINITION 35 (Maximal permissiveness) Let S be a H-safe supervisor and $L \subseteq \hat{\Sigma}^*$. S is maximally permissive w.r.t. L if for all supervisors S' holds

$$S' \text{ is } H\text{-safe w.r.t. } L \implies L \setminus (\mathfrak{L}_\mathcal{D}(S') \cup \mathfrak{L}_\delta(S')) \subseteq_F L \setminus (\mathfrak{L}_\mathcal{D}(S) \cup \mathfrak{L}_\delta(S))$$

5.8.5.2 Preemptive Synthesis

In the following, supervisor synthesis with preemption is introduced. In analogy to Section 5.8.2, a function **sysp** is defined to embody one iteration of the algorithm.

As for non-preemptive synthesis, a specification can be unrealizable on the given set of plants or even on L_{phy} itself. Apparently, realizability for non-preemptive synthesis is a sufficient criterion for the preemptive method to succeed. It is not necessary though. There are examples that are conventionally unrealizable, i.e., $\mathfrak{L}_\mathcal{P}(H@G) \cap \hat{\Sigma}_u^* \neq \emptyset$, but where for each string of this set a suitable preemption can be triggered, making the problem preemptively realizable. Unfortunately, this is a recursive property and thus badly definable on languages. Basically, it needs to ensure that for each uncontrollable forbidden string there is an available preemption such that the same holds for the state reached thereafter.

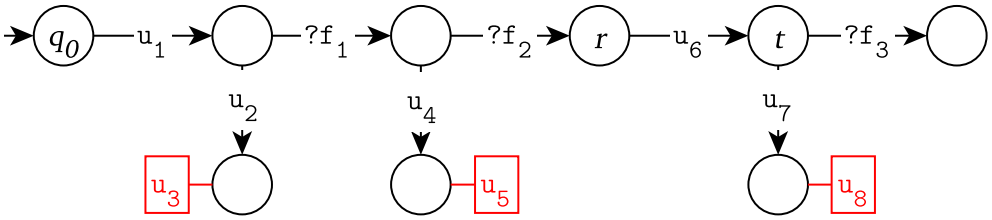


Figure 5.19: Synthesis automaton that is preemptively but not conventionally realizable, $u_1, \ldots, u_8 \in \hat{\Sigma}_u$, $f_1, \ldots, f_3 \in \hat{\Sigma}_f$.

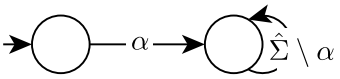


Figure 5.20: Auxiliary plant G_{aux}

Example 5.18. Figure 5.19 illustrates that. All three enforceable events need be enforced to guarantee safety. $?f_1$ and $?f_2$ together establish safety for q_0. This, in return, requires $?f_3$ as outlet from state t as otherwise executing $?f_2$ would not be safe. △

The algorithm presented later in this section is able to handle cases like that as it operates backwards from uncontrollable prohibitions towards an entirely safe statespace in which all critical states are either left immediately or unreachable. However, the property sketched above cannot be defined on the languages of H and $\mathfrak{G}$ straightforwardly. Fortunately, realizability can easily be reduced to a permissiveness problem as shown in the following. For that, we introduce an auxiliary controllable event α and a plant model: $G_{aux} := (\{q_0, q_1\}, \hat{\Sigma}, f_{aux}, q_0)$ where

$$f_{aux}(q,e) := \begin{cases} q_1 & q = q_0 \ \wedge \ e = \alpha \\ q_1 & q = q_1 \ \wedge \ e \neq \alpha \\ \text{undefined} & \text{otherwise} \end{cases}$$

Since its alphabet is $\hat{\Sigma}$, it specifies that α must be the very first event to happen for all strings in $\mathfrak{L}^{\Pi}(G_{aux})$. Since α is controllable, it can always be disabled as ultima ratio. Thus, the synthesized supervisor is never void. If it disables α, however, this means that the specification was unrealizable on the original plants in $\mathfrak{G}$.

Disabling α is apparently more restrictive than allowing it. Hence, by introducing G_{aux}, an unrealizable problem can without loss of generality be transformed to one that yields the *maximally restrictive* supervisor. In return, if the provenly maximally permissive supervisor disables α, there is no H-safe supervisor for the original problem at all.

When using the monolithic procedure, the scope of synthesis is set by the user. Hence the user is also responsible to ensure compliance with the preemption contract. The incremental method can be extended to support preemption too. Unfortunately, the directedness of @ regarding $\mathfrak{L}_{\mathcal{P}}$ (Line 5.1 of Lemma 12) does not hold for $\mathfrak{L}_{\delta}$. The reason is that an

additional plant model G can theoretically unveil the infeasibility of an event enforced in an automaton C, causing the previously preempted events $\delta_C(\,)$ and their continuations to be not preempted anymore in $C@G$.

Nonetheless, the incremental method can safely be used when all such dependencies are resolved by composition *before* an event is enforced. Preemption is hence always based on full information about the events to enforce, as in the monolithic case, which will be presented and proven sound in the following.

Definition 36 (Preemptive synthesis step, **sysp**) A preemptive synthesis step is a mapping $\mathbf{sysp} : \mathfrak{C}_\delta \to \mathfrak{C}_\delta$ and $\mathbf{sysp}(Q, \Sigma^\Pi, f, \gamma, P, q_0) = (Q', \Sigma^\Pi, f', \gamma', P', q_0')$, where

- Q' is the biggest set s.th. $Q' \subseteq Q$, and for all $q \in Q' : P(q) \cap \hat{\Sigma}_u = \emptyset$ or $\exists e \in \hat{\Sigma}_f : f(q,e) \in Q'$.
- $\gamma' : Q' \rightharpoonup \hat{\Sigma}_f$ s.th.

$$\gamma'(q) = \begin{cases} \gamma(q) & \text{if } \gamma(q)! \ \wedge \ f(q,\gamma(q)) \in Q' \\ \mathbf{choose}(F_q) & \text{if } \hat{\Sigma}_u \cap P(q) \neq \emptyset \ \wedge \ F_q \neq \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

  where $F_q := \{e \in \hat{\Sigma}_f \mid f(q,e) \in Q'\}$ is the set of enforceable events available in q. $\mathbf{choose}(F)$ picks one arbitrary event from F.

- $f' : Q' \times \hat{\Sigma} \rightharpoonup Q'$ s.th.

$$f'(q,e) := \begin{cases} f(q,e) & \text{if } f(q,e) \in Q' \text{ and } e \in \hat{\Sigma}_c \\ f(q,e) & \text{if } f(q,e) \in Q' \text{ and } \gamma'(q)! \\ f(q,e) & \text{if } f(q,e) \in Q' \text{ and } e = \gamma'(q) \\ \text{undefined} & \text{otherwise} \end{cases}$$

- $P' : Q' \to 2^{\hat{\Sigma}}$ s.th. $e \in P'(q) \iff e \in P(q) \cap \hat{\Sigma}_c$ or $f(q,e) \in Q \setminus Q'$
- $q_0' = \begin{cases} q_0 & \text{if } q_0 \in Q' \\ \text{undefined} & \text{otherwise} \end{cases}$

Monolithic synthesis applies **sysp** subsequently until a fixpoint is reached.

Lemma 21 (Fixpoint) Let $C \in \mathfrak{C}_\delta$. There is $n \in \mathbb{N}$ s.th. $\forall m > n : \ \mathbf{sysp}^n(C) = \mathbf{sysp}^m(C)$.

Proof. **sysp** shrinks down Q monotonically. Further, it can be seen for $C' = \mathbf{sysp}^{n-1}(C)$ that if $Q_{C'} = Q_{\mathbf{sysp}(C')}$, then $P_{C'} = P_{\mathbf{sysp}(C')}$, i.e., no further prohibitions are established. Indeed can $\gamma_{\mathbf{sysp}(C')}$ change compared to $\gamma_{C'}$. As there are no new prohibitions, a fixpoint is reached then since γ only reacts on new newly imposed forbidden events. Hence, $\mathbf{sysp}(C') = \mathbf{sysp}(\mathbf{sysp}(C'))$. □

We label that fixpoint for a given C by $\mathbf{sysp}^{\text{fix}}(C)$. Like in the non-preemptive case, we can ensure the directedness of **sysp**. In that context, it is important that the two languages $\mathfrak{L}_\mathcal{P}$ and $\mathfrak{L}_\delta$ are considered together because a string can move from one language to the other if one of its prefixes becomes forbidden or preempted. Remember that neither $\mathfrak{L}_\mathcal{P}$ nor $\mathfrak{L}_\delta$ are in general prefix-closed.

LEMMA 22 (Directedness of **sysp**) Let $C \in \mathfrak{C}_\delta$ be realizable.

$$\mathfrak{L}_\mathcal{P}(C) \cup \mathfrak{L}_\delta(C) \quad \subseteq \quad \mathfrak{L}_\mathcal{P}(\mathbf{sysp}(C)) \cup \mathfrak{L}_\delta(\mathbf{sysp}(C))$$

Proof. Throughout this proof, define $C' := \mathbf{sysp}(C)$.
Let $s \in \mathfrak{L}_\mathcal{P}(C)$. There is a decomposition $s'es'' = s$ s.th. $s' \in \mathfrak{L}^\Pi(C), e \in P_C(f_C^*(q_{0C}, s'))$.

- If $e \in \hat{\Sigma}_c \cup \hat{\Sigma}_f$ then, by construction of **sysp**, $e \in P_{C'}(f_{C'}^*(q_{0C'}, s'))$ and hence $s'e \in \mathfrak{L}_\mathcal{P}(C')$. Due to the closedness under continuation, $s'es'' \in \mathfrak{L}_\mathcal{P}(C')$.
- If $e \in \hat{\Sigma}_u$, there exist two sub-cases. Let $q := f_C^*(q_{0C}, s')$.
  - *Case 1:* $\exists e_f \in \hat{\Sigma}_f$ s.th. $f_C(q, e_f)!$. Then $f_{C'}^*(q_{0C'}, s') = q \in Q'$ and $\gamma_{C'}(q)!$. It follows $e \in \delta_{C'}(q)$ and $s'e \in \mathfrak{L}_\delta(C')$ and finally $s'es'' = s \in \mathfrak{L}_\delta(C')$.
  - *Case 2:* $\forall e_f \in \hat{\Sigma}_f : f_C(q, e_f)!$ (there is no enforceable event in q). Then, by the definition of **sysp**, $q \notin Q'$ and thus either $q_0 \notin Q'$ or $\exists u \in \hat{\Sigma}^*$ s.th. $s' = ue'$ and $e' \in P_{C'}(f_{C'}^*(q_0, s'))$. It follows $s', s'e, s'es'' \in \mathfrak{L}_\mathcal{P}(C')$.

Let $s \in \mathfrak{L}_\delta(C)$. Then there is a decomposition $s'es'' = s$ s.th. $s' \in \mathfrak{L}^\Pi(C)$ and $e \in \delta_C(f_C^*(q_{0C}, s'))$. Again, two sub-cases emerge:

- *Case 1:* $s' \notin \mathfrak{L}^\Pi(C')$. Then $s' \in \mathfrak{L}_\mathcal{P}(C') \cup \mathfrak{L}_\delta(C')$ and hence $s'es'' \in \mathfrak{L}_\mathcal{P}(C') \cup \mathfrak{L}_\delta(C')$.
- *Case 2:* $s' \in \mathfrak{L}^\Pi(C')$. Then $e \in \delta_{C'}(f_{C'}^*(q_{0C}, s'))$. It follows $s'e$, $s'es'' \in \mathfrak{L}_\delta(C')$. □

LEMMA 23 (Safety of **sysp**) Let $L \in \hat{\Sigma}^*$ a language, $H \in \mathfrak{H}$ a specification and $C \in \mathfrak{C}_\delta$ a synthesis automaton such that $L \subseteq \mathfrak{L}^\Pi(C)$ (C fulfills the plant contract) and $\forall s \in L, e \in \hat{\Sigma}_f : se \in \mathfrak{L}^\Pi(C) \implies se \in L$ (C fulfills the preemption contract). Let further $\mathfrak{L}^\Pi(C) \subseteq \mathfrak{L}_\mathcal{S}(H)$ (C is H-safe). Then $S := \mathbf{sysp}^{\text{fix}}(C)$ is an H-safe supervisor if q_{0S} is defined.

Proof. Supervisor: Assume S is not a supervisor. Then exists $q \in Q_S$ s.th. $P(q) \cap \hat{\Sigma}_u \neq \emptyset$. In that case, by construction of **sysp**, we get $\mathbf{sysp}(S) \neq S = \mathbf{sysp}^{\text{fix}}(S)$. Contradiction.

Safety: From $\mathfrak{L}^\Pi(C) \subseteq \mathfrak{L}_\mathcal{S}(H)$ and $\mathfrak{L}^\Pi(C) \cap (\mathfrak{L}_\delta(C) \cup \mathfrak{L}_\mathcal{P}(C)) = \emptyset$ follows that $L \setminus (\mathfrak{L}_\delta(C) \cup \mathfrak{L}_\mathcal{P}(C)) \subseteq \mathfrak{L}_\mathcal{S}(H)$. Thanks to Lemma 22, that yields $L \setminus (\mathfrak{L}_\delta(\mathbf{sysp}^{\text{fix}}(C)) \cup \mathfrak{L}_\mathcal{P}(\mathbf{sysp}^{\text{fix}}(C))) \subseteq \mathfrak{L}_\mathcal{S}(H)$. □

THEOREM 24 (Correctness of preemptive synthesis) Let $H \in \mathfrak{H}$ a specification and $S := \mathbf{sysp}^{\text{fix}}(H@\hat{G})$.

1. If S is void, then there is no H-safe supervisor.
2. If S is not void, it is H-safe w.r.t. L_{phy}.
3. If S is not void, it is H-safe and maximally permissive w.r.t. $\hat{G}$.

Proof. **2.** Follows directly from the correctness and the completeness w.r.t enforceable events of $\mathfrak{G}$, and Lemma 23.

3. Safety can be proven in analogy to 2. For the maximal permissiveness, show that for all supervisors S' holds S' H-safe $\implies \mathfrak{L}^\Pi(\hat{G}) \setminus (\mathfrak{L}_\mathcal{D}(S') \cup \mathfrak{L}_\delta(S')) \subseteq_F \mathfrak{L}^\Pi(\hat{G}) \setminus (\mathfrak{L}_\mathcal{D}(S) \cup \mathfrak{L}_\delta(S))$. Assume there is some S' more permissive than S and H-safe. Define the following abbreviations: $L_S := \mathfrak{L}^\Pi(\hat{G}) \setminus (\mathfrak{L}_\mathcal{D}(S) \cup \mathfrak{L}_\delta(S))$, $L_{S'} := \mathfrak{L}^\Pi(\hat{G}) \setminus (\mathfrak{L}_\mathcal{D}(S') \cup \mathfrak{L}_\delta(S'))$.
Then exists $v \in L_{S'}, v \notin L_S$, ergo $v \in \mathfrak{L}_\mathcal{D}(S) \cup \mathfrak{L}_\delta(S)$, so that there is no decomposition

$v = s'es'',\ s', s'' \in \hat{\Sigma},\ e, e' \in \hat{\Sigma}_f$, where $s'e' \in L_S \setminus L_{S'}$ and $s'e \in L_{S'} \setminus L_S$, i.e., S and S' do not differ just in the choice of an enforced event. Let v_0 be the shortest forbidden or preempted prefix of v, i.e., every proper prefix of v_0, if it exists, is legal.
Case 1: $v \in \mathfrak{L}_{\mathcal{D}}(S)$. Two sub-cases exist:
Case 1a: $v_0 \in \mathfrak{L}_{\mathcal{P}}(H)$. Then S' is not H-safe. Contradiction.
Case 1b: $v_0 \notin \mathfrak{L}_{\mathcal{P}}(H)$. Let i_0 denote the iteration where the fixpoint is reached for the first time, i.e., $\mathbf{sysp}^{i_0}(H@G) = \mathbf{sysp}^{i_0+1}(H@G)$. Then there has been a previous iteration $0 \leq i_1 < i_0$, where v_0 was legal. Let $C_1 := \mathbf{sysp}^{i_1}(H@G)$ and $C_0 := \mathbf{sysp}(C_1)$. Then $v_0 \notin \mathfrak{L}_{\mathcal{P}}(C_0),\ v_0 \in \mathfrak{L}_{\mathcal{P}}(\mathbf{sysp}(C_0))$. For the prohibition of v_0 in $\mathbf{sysp}(C_0)$ the definition of **sysp** requires that $q^0 := f^*(q_{0C_0}, v_0) \notin Q_{C_0}$, which is only the case if there is $e_0 \in \hat{\Sigma}_u$ so that $e_0 \in P(q^0)$ (e_0 is forbidden in q^0) and preemption of e_0 was not possible, $\nexists e_f, f(q^0, e_f)!$. Define $v_j := v_{j-1}e_{j-1}$. The same two cases (1a or 1b) apply again inductively for $v_1 = v_0e_0$ and every further v_j. Since $i_j = i_{j-1} - 1$ for $j > 1$, eventually $i_m = 0$ is reached where Case 1b collapses to $v_me_m \in \mathfrak{L}_{\mathcal{P}}(\mathbf{sysp}^0(H@G)) = \mathfrak{L}_{\mathcal{P}}(H@G) \subseteq \mathfrak{L}_{\mathcal{P}}(H)$. Since $v_me_m = v_0e_1 \dots e_m$ and all e_j are uncontrollable and preemption is not available for any of them, $v_me_m \notin \mathfrak{L}_{\mathcal{D}}(S') \cup \mathfrak{L}_{\delta}(S')$. Thus, S' is not H-safe. Contradiction.
Case 2: $v_0 \in \mathfrak{L}_{\delta}(S)$. Again, two sub-cases are obtained.
Case 2a: $v_0 \in \mathfrak{L}_{\mathcal{P}}(H)$ then S' is not H-safe. Contradiction.
Case 2b: $v_0 \notin \mathfrak{L}_{\mathcal{P}}(H)$. Again, let i_0 address the iteration where the fixpoint is found and $i_1,\ 0 \leq i_1 < i_0$, the iteration where v_0 was legal for the last time, i.e., for $C_1 := \mathbf{sysp}^{i_1}(H@G), C_0 := \mathbf{sysp}(C_1)$ holds $v_0 \notin \mathfrak{L}_{\delta}(C_1),\ v_0 \in \mathfrak{L}_{\delta}(C_0)$. Let $v_0' \in \hat{\Sigma}^*, e \in \hat{\Sigma}$, s.th. $v_0 = v_0'e$. For v_0 being preempted in $\mathbf{sysp}(C_1)$, there must be an enforced $e_f \in \hat{\Sigma}_f$ s.th. for $q := f_{C_0}(q_{0C_0}, v_0')$ holds $\gamma_{C_0}(q) = e_f$ and $f_{C_0}(q, e_f) \in Q_{C_0}$. Since $v_0 \notin \mathfrak{L}_{\delta}(C_1)$ and thus $\gamma_{C_1}(q)!$, it is required that there is an uncontrollable prohibited event $e_0 \in \hat{\Sigma}_u \cap P_{C_1}(q)$ which is preempted in C_0 but not yet in C_1. For $v_1 := v_0'e_0$ holds $v_1 \in \mathfrak{L}_{\mathcal{P}}(C_1)$, i.e., Case 1 applies for v_1. This, again, inductively leads to some $v_me_m \in \mathfrak{L}_{\mathcal{P}}(H@G)$. Again, the prohibition of all $e_j, 0 < j \leq m$, in $v_me_m = v_0'e_0 \dots e_m$ requires that there is no enforceable event available to allow for preemption. Since $v \in \mathfrak{L}^{\Pi}(G) \setminus (\mathfrak{L}_{\mathcal{D}}(S') \cup \mathfrak{L}_{\delta}(S'))$ and the prefix-closedness of the latter set, $v_0 \notin \mathfrak{L}_{\mathcal{D}}(S') \cup \mathfrak{L}_{\delta}(S')$. Due to that and $v_0e_0 \dots e_m \in \mathfrak{L}_{\mathcal{P}}(H)$ where all $e_0, \dots, e_m \in \hat{\Sigma}_u$, it must be $e \in \hat{\Sigma}_f$ and $e = \gamma_{S'}(f^*_{S'}(q_{0S'}, v_0'))$, i.e., e is the enforced event after v_0' to preempt the undesired e_0. As e preempts e_f in S' and e_f preempts e in S, with the decomposition $v = v_0s'' = v_0'es''$ we get that $v_0' \in L_{S'},\ v_0'e_f \notin L_{S'}$ and $v_0'e \notin L_S$ and finally $v_0'e_f \in L_S \setminus L_{S'}$ and $v_0'e \in L_{S'} \setminus L_S$. Contradiction.
Hence, S is maximally permissive w.r.t. $\hat{G}$.

1. Follows from the maximal permissiveness of S when G is replaced by $G \parallel G_{aux}$ as described on Page 111. □

Algorithm 3 sketches the monolithic synthesis with preemption procedurally.

It should be mentioned that the presented method is not able to detect *enforcement cycles* as mentioned in Section 5.5.5. Formally, safety is established as every forbidden event uncontrollable in that cycle is considered as preempted and the corresponding string is contained in $\mathfrak{L}_{\delta}(S)$. In real scenarios, this is obviously more problematic as time is not considered in the formal model. Detecting cycles can be realized straightforwardly. It is, however, non-trivial to algorithmically decide which of the enforced transitions shall

Algorithm 3 Monolithic supervisor synthesis with preemption, monP(C)

Input: $C = (Q, \Sigma^{\Pi}, f, \gamma, P, q_0)$

```
 1: fixpoint := false;
 2: while ¬ fixpoint do
 3:    fixpoint := true
 4:    for all q ∈ Q do
 5:       if P(q) ∩ Σ̂_u ≠ ∅ then
 6:          if γ(q)! ∧ ∃e_f ∈ Σ̂_f : f(q, e_f)! then
 7:             γ(q) := choose{e_f ∈ Σ̂_f | f(q, e_f)!};   // picks arbitrary element from set
 8:          else if γ(q)! then
 9:             fixpoint := false;
10:             ∀e remove (q, e) from f;
11:             Q := Q \ {q};
12:             for all (q', e') ∈ f^{-1}(q) do
13:                remove (q', e') from f;
14:                remove q' from γ;
15:                P(q') := P(q') ∪ {e'};
16:             end for
17:             remove all unreach. states from Q and their outg. trans. from f;
18:          end if
19:       end if
20:    end for
21: end while
22: for all q ∈ Q where γ(q)! do
23:    P(q) := P(q) ∩ (Σ̂_c ∪ Σ̂_f);   // remove preempted prohibitions
24: end for
25: if q_0 ∈ Q then
26:    return (Q, f, γ, P, q_0);
27: else
28:    return "unrealizable!";
29: end if
```

be replaced by another one or even dropped to achieve the maximally permissive result. Accordingly, the definitions of δ and $\mathfrak{L}_\delta$ needed to be strengthened in an appropriate way to declare cycles of enforced transitions as unacceptable.

Chapter 6

SynTACS

The Synthesis Tool for Automation Controller Supervision (SynTACS) is a software tool which utilizes DES for the monitoring and supervision of controllers. It prototypically realizes the supervisory control concept presented in the previous chapter and this way allows to analyse and evaluate its suitability for the addressed scenario. Therefore, SynTACS explicitly addresses the enforcement of high-priority side conditions which must not be violated at any time. It offers an end-to-end solution from convenient and efficient modeling support to a smooth and easy integration of synthesized supervisors to the final controller code.

This chapter puts a focus on technical aspects such as the management of modeling artifacts, the software architecture in general and, particularly, the runtime framework which integrates the supervisors with the controller under supervision on the target hardware.

The first international presentation of the tool was on the 2016 Workshop on Discrete Event Systems [43].

6.1 Working with SynTACS

When using SynTACS the user needs to provide specfications and plant models as described in Section 5.3. The tool can compute compositions (synthesis automata) and supervisors based on these input automata. To that end, it provides an implementation of all concepts and methods discussed in Chapter 5.

6.1.1 User Interface

Graphical user interfaces (GUIs) and even graphical programming languages (cf. Section 2.3.4) are widely spread and gain larger acceptance in the area of industrial automation compared to textual ways of programming. SynTACS follows this paradigm and can entirely be controled through its GUI.

The screenshot in Figure 6.1 shows the main window of SynTACS. On its very left, the project structure is displayed, containing *modules*, automata and event definitions. The graphical automaton editor is situated in the center. States can be added and freely moved

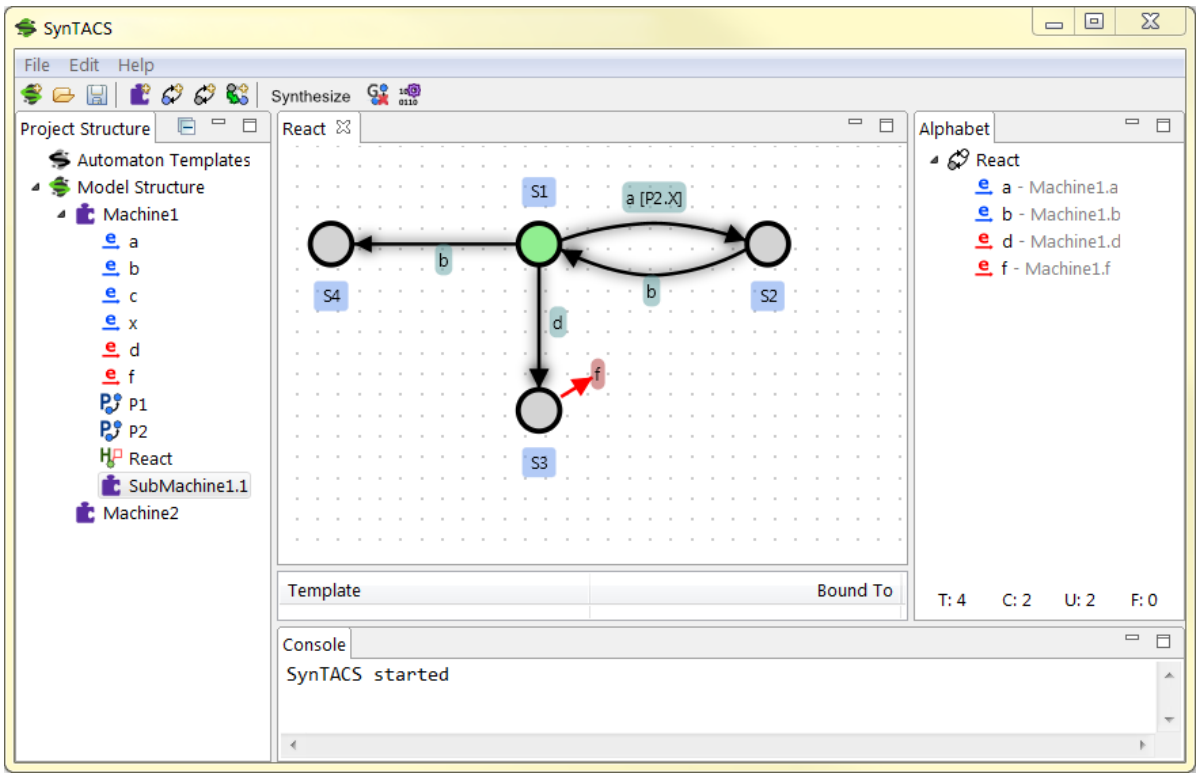


Figure 6.1: Screenshot of SynTACS

around with the mouse. The same holds for transitions between two states. On the right, the automaton's alphabet is shown.

Drag and Drop is widely supported within the GUI to allow for a quick and intuitive workflow. Access to all functions (minimization, synthesis, composition, etc.) is provided through the UI.

After providing all required events, specifications and plant models, synthesis can be started. The user can, for instance, select a specification and trigger incremental synthesis on it. SynTACS then computes one supervisor for each given specification automatically, using the method described in Section 5.4.3 and formalized in Section 5.8.3. Finally, the generation of PLC code can be invoked through a wizard.

6.1.2 Project Structure

SynTACS subsumes all user-provided and generated entities, such as events and automata, in a project tree. On the top layer, the project consists of two groups: *Automaton Templates* and *Model Structure*. Templates allow to reuse the structure of an automaton to represent or address multiple parts of actual hardware. They will be the subject of Section 6.1.4.

SynTACS follows a strictly modular approach. To support that, the user can group together elements as automata or events in modules. These may also contain arbitrarily many levels of further (sub-)modules. Since the module structure also establishes a namespace hierarchy, multiple modules may contain events or automata with identical names as long as no name

occurs twice in the very same module, outside its submodules. The *qualified name* of a module, automaton, event or state is constructed by concatenation of the names of the superordinate containers, separated by a dot. For instance, `Machine1.React.S2` refers to the state `S2` in automaton `React` of module `Machine1` (cf. Figure 6.1). In the conditions of transitions and prohibitions (cf. Section 5.7), states are also referenced by their qualified name. Qualified names can be used relatively. The condition `[P2.X]` on the `a` transition from `S1` to `S2` inside the `React` automaton in Figure 6.1 for example refers to the state `X` of the plant automaton `P2` within the same module `Machine1`. In case of a name collision, absolute qualified names overrule relative ones.

Modules are meant to represent different parts of the plant. In many academic publications, single components are represented by one automaton each. This, however, can be confusing for several reasons [42]. First, ordinary automata cannot reflect the hierarchical structure of a plant where each component can also be made of further sub-components and so forth. Second do events have a much stronger connection to the actual hardware than automata do. This is because they represent concrete actions or happenings, whereas automata only relate those events to each other, i.e., their are only indirectly connected to the hardware through the events. For these reasons, it makes sense to include the events into the modular organization. As automata synchronize via events, they do not actually host them in the sense of an event being “local” or “homed” in one particular automaton. Instead, there can even be multiple automata defining several behavioral aspects of the same component by establishing different relationships on its events, which is the third reason for the module hierarchy.

6.1.3 Accessing PLC Timers

Some of the requirements which shall be realized with SynTACS involve spans of time. When time is entirely neglected, the supervisor often needs to be more restrictive to guarantee compliance with the specification, similarly to the non-preemptive case sketched in Example 5.4. It is obviously not possible to represent timing coherences in untimed models as ordinary DES. It is possible, though, to measure time in an absolute manner outside of the models to realize delays. Although this rarely influences permissiveness fundamentally, it can be used to realize a more tolerant kind of supervision, e.g., by delaying enforced reactions.

Like events and automata, timers can be added to any module of a SynTACS project [41]. Each timer definition consists of the following elements.

- A duration, given in hours, minutes, seconds and milliseconds
- One associated event which starts the timer
- Arbitrarily many associated events which stop and reset the timer
- One dedicated *timeout* event

The term “associated” means that existing events can be referenced in order to start or stop the timer. An event can be associated with multiple timers and even serve as start event

for one and as stop event for another timer. A dedicated uncontrollable timeout event is automatically created for every timer.

When the *start* event of a timer occurs in a supervisor, the corresponding timer begins measuring time. Once the specified duration is reached, the timeout event is triggered and tracked by every running supervisor that has a transition for it in the currently active state. When one of the *stop* events is detected before the timeout, the timer is stopped and reset. All these events can be handled as usual and particularly be shared among several supervisors.

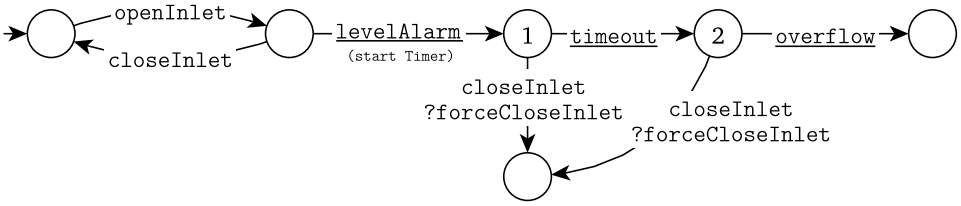
Time measurements can be relevant in situations in which a correct behavior of the controller cannot be detected immediately after some sensor event.

Example 6.1. Consider again the plant model and preemptive supervisor shown in Figure 5.8 and the specification from Figure 5.7b (simply prohibits `overflow`). It was assumed that the controller has to react immediately by closing the inlet valve once `levelAlarm` has been detected. If the corresponding controllable event `openInlet` is not recognized within the same cycle, the enforced event `!closeInlNow` would be triggered to preempt the overflow. There may be situations where this reaction is still too harsh. Assume that, due to the maximum inflow, it is known that an overflow cannot occur within two minutes after the level sensor is reached. Upstream of the valve, a pump is installed which the controller regularly shuts down before closing the inlet. The emergency action `!closeInlNow` can potentially damage the pump when applied too often and hence should be avoided. Thus, supervision should give the controller some additional time to execute the proper reaction sequence before closure is ultimately enforced.

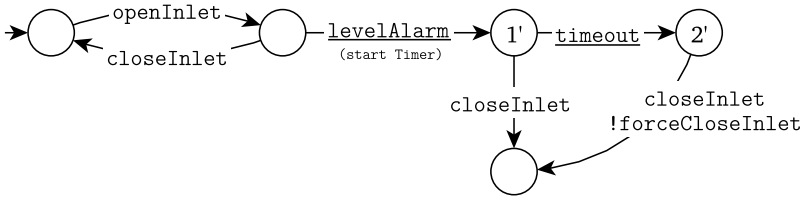
Figure 6.2 shows the plant model augmented with a timer set to 15 seconds, sufficient time for the controller to shut down the pump and close the valve regularly but still sufficiently early to reliably prevent a critical filling level in the tank. The uncontrollable `levelAlarm` starts the timer. The new plant model includes the physical aspect that the overflow will not occur before the timeout. Hence, synthesis does not enforce `?forceCloseInlet` in state $1'$ but in $2'$ where the prohibited `overflow` is imminent. For the sake of simplicity, stop-and-reset events are not considered here. An event indicating that the level has fallen below the sensor again would be a good candidate for that. △

6.1.4 Templates

Use cases that deal with realistic plants often involve several instances of the same hardware type, e.g., valves, tanks, pumps, etc. Therefore, it is usually necessary to provide multiple automata for these components. Since the connection between the modeling world and the real hardware is established via the triggers and actions of the events (cf. Section 5.3.2), these automata usually include diverse events to address different pieces of hardware. Often, they share the same behaviors and requirements among the same hardware type though. That means the user has to define almost the same automaton several times, containing the same states and equivalent transitional structure, yet on different events. To reduce that effort, SynTACS provides the concept of *automaton templates*. It has been published in [42], which this section is largely based on.



(a) Plant model



(b) Maximally permissive preemptive supervisor

Figure 6.2: Example with preemption and timers

A template can be understood as a generic automaton for a given hardware or specification type. It comprises states, transitions and, in case of specifications, prohibitions like every ordinary automaton. In contrast to the latter, transitions and prohibitions in a template can be labeled with placeholders, called *binding events*. These are defined right in the template and are not visible from outside.

A template can be *instantiated* once or multiple times at any arbitrary module in the SynTACS project hierarchy. The instance must provide a mapping from every binding event onto a proper event somewhere in the project.

Binding events can be grouped in *namespaces*. On instantiation, an entire namespace can be assigned to a module in the project at once. That module then inherits its events to the binding events of the namespace, meaning that SynTACS automatically associates all contained binding events with the events of the module which match the name, if they exist. Missing events need to be mapped manually. The user can override an inherited event mapping, e.g., if one specific event does exist in the chosen namespace but another event shall be used instead.

Namespaces can be used to define abstract *roles* within the template. This suites the notion of modules representing certain pieces of hardware. A namespace represents the abstract “interface” of a component, declaring the necessary events and the template itself provides the behavior or requirement based on this interface. A module containing an instance of that template represents real hardware to fullfill that role. The binding events of the template are matched to the module’s events which finally provide the connection to the hardware through their triggers and actions.

When all events required in a namespace are found in a module, the entire binding can be inherited and the template instance needs not being touched at all.

The following example is quoted verbatim, yet syntactically adapted to the style of this thesis, from the original paper [42] and demonstrates how templates can be used. Altered, adapted or corrected words and phrases are written in brackets.

Example 6.2. Consider the example shown in Figure 6.3 containing three tanks and five valves which are connected as shown in Part 6.3a. In this scenario, three safety requirements are claimed, which are: (1) a reflux of liquid from T1 to T2 or vice versa may never happen, (2) it is not allowed that the inlet and drainage valve of a tank are open at the same time to avoid a straight flow through the tank and (3) only one of the two liquids A and B may be drained into T3 at each time to avoid them to react. The subfigures 6.3e and 6.3f show the specification automata that prohibit the events `reflux` and `reaction` to occur at any time. Both are [underlined] to indicate that they are uncontrollable. Transitions are not required in these specifications since the state of the requirements does never change. In other words, there is no state where `reflux` or `reaction` would be allowed.

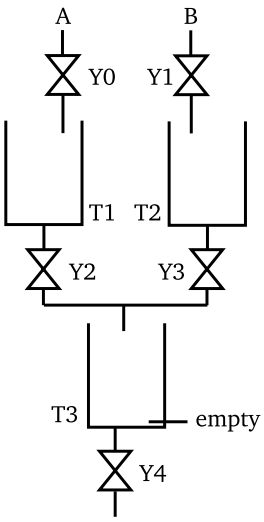
In order to make the problem controllable, plant models are needed to define in which situations those undesired events are possible to occur. To that end, the user has to provide the plant automata `jnPps` (Fig. 6.3c) and `T3` (Fig. 6.3g). According to the first, a reflux is only possible when Y2 and Y3 are open at the same time. The second claims that, after the drainage [of] one of the tanks T1 or T2 has been started, the level sensor event `empty` has to occur before the other one may be drained to avoid a reaction. While `empty` and `reaction` are local events inside T3's namespace [(module)[1]], the events of Y2 and Y3 are referenced using the namespaces in which they are defined. Figure 6.3b shows the plant automaton template for arbitrary valves. In contrast to the concrete automata `JnPps` and `T3`, the template `Vlv` has to be instantiated once for every concrete valve. The left-hand side of Fig. 6.3h shows these five instantiations into the namespaces `Y0` to `Y4`. In all five cases, the automaton instance itself is called `behavior`. Note that each of `Y0` to `Y4` must contain the events `op` (open) and `cl` (close) [to allow their inheritance from the module]. Since `Vlv` does not use roles, a role mapping is not required.

Figure 6.3d shows the specification template `VlvMutex` that forbids two valves, addressed by the role names `local` and `remote`, to be open at the same time. The instantiations of `VlvMutex`, depicted on the right-hand side of Fig. 6.3h, are created in the [module] `Mtx` for the valve pairs Y0 & Y2, Y1 & Y3, Y2 & Y4, Y3 & Y4 and vice versa. For every instance, the roles `local` and `remote` are mapped to the [module] of the according valve. Without templates, all those instances would have to be defined as separate automata.

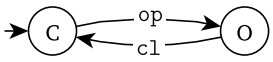
After the user has provided all necessary event definitions, the automata and templates as shown in Figure 6.3 and the instantiation mappings, the synthesis of the supervisors can be triggered. △

Figure 6.4 exemplarily shows the instance `Mtx.Y0Y2` of the template `VlvMutex` as it would be defined in SynTACS. The namespaces `remote` and `local` are bound to the modules `Y2` and `Y0` respectively. It is clearly visible that the bindings for the comprised binding events are inherited from these modules. To change the bindings, the user can click

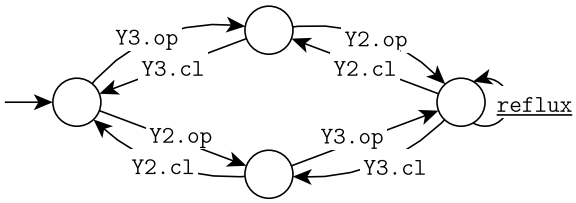
[1] In the original paper, modules and namespaces were not strictly distinguished yet. The term "module" is technically correct here.



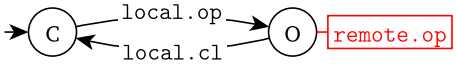
(a) P&ID



(b) Plant automaton template for a valve: `Vlv`

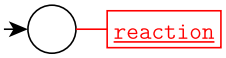


(c) Plant automaton for joining pipes: `JnPps`

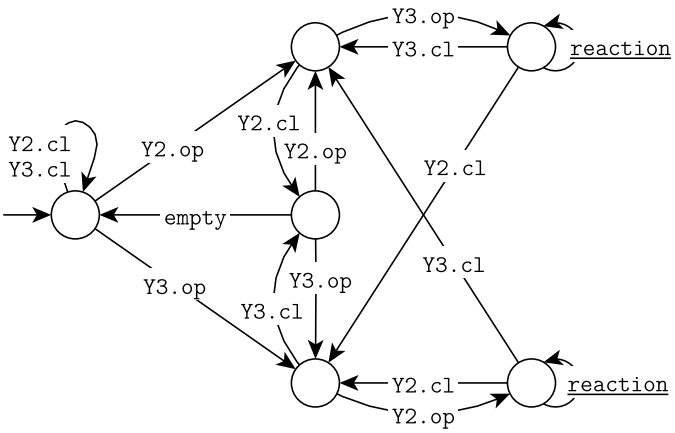


(d) Specification template for mutual exclusion of two open valves: `VlvMutex`

(e) Specification prohibiting reflux



(f) Specification prohibiting the liquids to react



(g) Plant Automaton for `T3`

`Y0.behavior : Vlv`
`Y1.behavior : Vlv`
`Y2.behavior : Vlv`
`Y3.behavior : Vlv`
`Y4.behavior : Vlv`

`Mtx.Y0Y2 : VlvMutex`	
`local`	`Y0`
`remote`	`Y2`

`Mtx.Y2Y0 : VlvMutex`	
`local`	`Y2`
`remote`	`Y0`

`...`

(h) Instantiations of templates. left: valve plant models. right: mutex specifications with namespace mapping to valve modules.

Figure 6.3: Template example – Excerpt of a chemical plant containing three tanks and five valves: Reflux from T1 to T2 and vice versa, both liquids in T3 at the same time, and straight flow through any of the tanks are forbidden (figure based on [42])

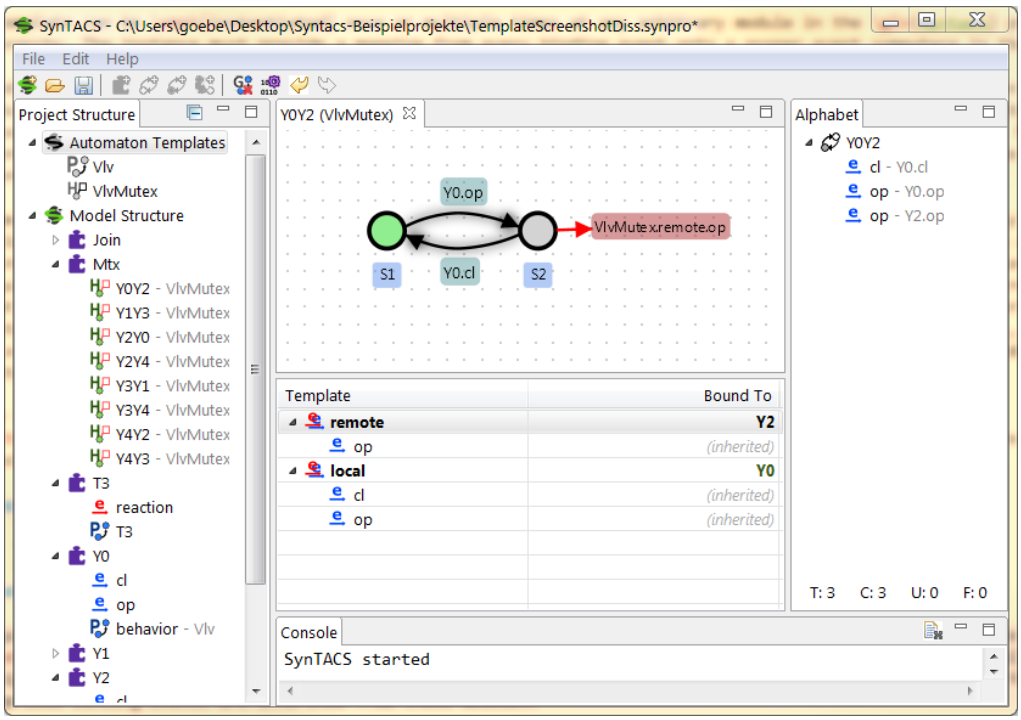


Figure 6.4: Instance of `VlvMutex` from Example 6.2 with namespace mapping in SynTACS

on the corresponding cell in the displayed table and a drop-down menu appears, offering all suitable modules or events. The editor above shows an immutable[2] version of the template automaton where all binding events are substituted by their associated concrete events for the sake of illustration.

6.2 Software Architecture

The first version of SynTACS was realized in 2014 as a plugin for *Eclipse 3*, a conveniently extendable and highly customizable IDE. In 2015, the new and current version of the tool was developed on top of the Eclipse 4 rich client platform (RCP)[3]. The latter embodies a framework for standalone applications rather than for extensions to the original IDE and thus offers significantly better flexibility. Besides, concepts as *dependency injection* and the unified *eclipse application model* improve the usability of the RCP compared to older versions.

SynTACS is modularly organized. Figure 6.5 shows its architecture schematically. The four main components are briefly described in the following.

Core Contains a basic model of states, transitions and a simple abstraction of events. Further, all algorithms including the mechanisms for composition and condition

[2] States and transitions may moved around and bended to get a better overview but no structural modifiactions can be applied as these would affect the template itself.

[3] Project website: `https://eclipse.org/`

resolution (cf. Section 5.8.4) are part of the core. It distinguishes the introduced automaton types *plant, specification, supervisor* and *synthesis automaton*. This module has no dependencies to the other components and thus is still runnable when decoupled from the rest of SynTACS, e.g., as a library.

Model The SynTACS model captures all aspects of how the elements of a SynTACS project, modules, automata, events and timers, are organized. In particular, it provides the set of available automata to the incremental synthesis method and offers a *state path resolver* to transform the qualified state names in conditions to proper object references. The Model is further responsible for storing and loading projects from the file system. The template system presented in 6.1.4 is entirely realized and encapsulated in this component too.

Editor The automaton editor of SynTACS is implemented upon the Graphical Editing Framework (GEF) 4, which follows the model–view–controller principle. Every displayable and editable core object such as states, transitions, etc. is supplemented with view data using the *decorator* pattern [40], mainly about the positioning of objects. The implemented *EditParts*, *policies* and *behaviors*[4] provide GEF with the necessary code for an appropriate user interaction with the displayed model on the one hand and a correct propagation of changes to the core components on the other. Besides, the EditParts trigger drawing the views as JavaFX elements using the view data.

Code generator This component translates the synthesized supervisors to executable PLC code using the *Apache Velocity* engine. In addition to the supervisor, an entire runtime framework is created, including the handling of events, timers, in- and output signals. Several dialects of Structured Text are already supported. However, thanks to a template managing system, the introduction of new languages is easily possible[5].

6.3 SynTACS Runtime Framework

SynTACS aims to make the methods of SCT available for monitoring and supervision of existing PLC programs. For this reason, strong emphasis was put on the possibility of generating an executable representation of the supervisor directly from the tool with minimal necessary user interaction. The result is the *SynTACS Runtime Framework (SyRF)* which roughly implements the work flow sketched in Figure 5.2 on Page 53.

The framework introduces pre- and succeeding operations to the actual controller. Thanks to the cyclic execution model of PLCs it can be guaranteed that these steps are carried out in perfect synchronization with the controller without the need of interrupts or manipulations on the operating system. More precisely, the framework is able to capture every changed input that the controller recognizes as well as all its modifications on the outputs. In that

[4] For detailed information on these concepts, refer to the project website: `https://eclipse.org/gef/`
[5] Code access is required to migrate the new templates into SynTACS.

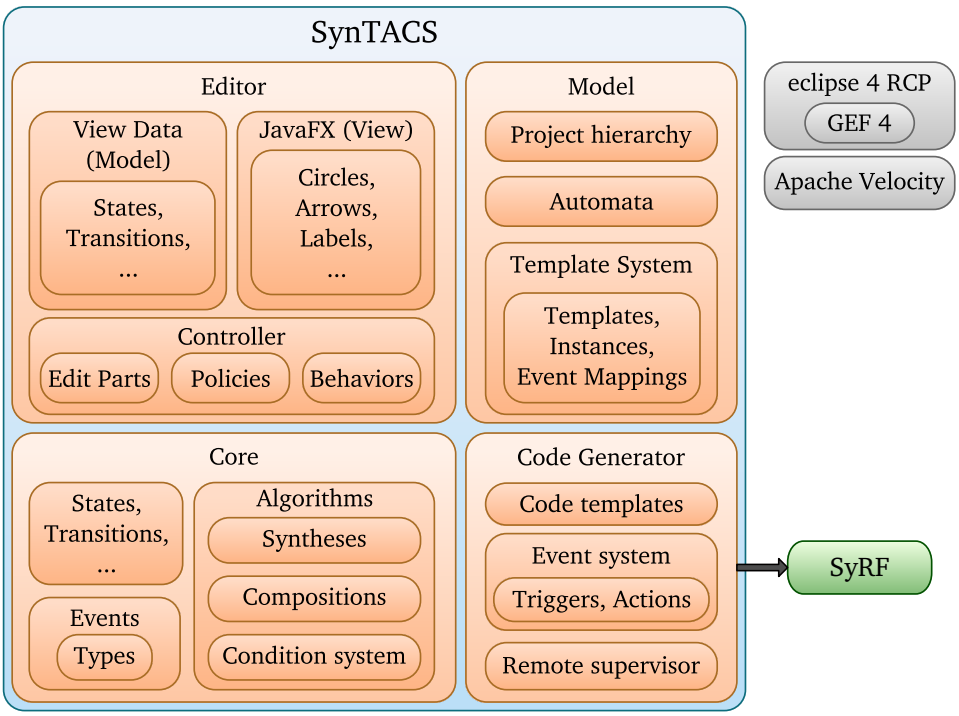


Figure 6.5: Architecture of SynTACS

context, it is vital that the latter are applied to the hardware no earlier than at the end of the entire cycle, which allows the framework to effectively block doubtful actions.

Large parts of SyRF have been developed in the context of [Timmermanns, 2015] and [Gatto, 2016].

6.3.1 Event Management

SyRF allows to use events with different trigger and action classes, as described in Section 5.3.2 at the same time. Further is it technically possible to use preemption (Section 5.5) and cyclic events (Section 5.6) in parallel or even in the same supervisor. Since the controller is assumed to react on what the plant did, where the supervisor evaluates the controller's decisions and potentially reacts these in turn, the logic order of events on the supervisor is always *uncontrollable events first, controllable events second, enforceable events last*. In the following, the technical realization of trigger and action classes is discussed. Figure 6.6 gives an overview about the SynTACS event scheme in full detail.

Trigger Classes In Chapter 5, two different tigger classes were introduced: events that are *detected* by the supervision framework and those which are explicitly *executed* by the controller. In case of the former, the controller manipulates the PIO of the PLC as usual. The user specifies a trigger condition on one or several values of the PIO as an ST expression. She can also access global controller variables with the macro `\var{`*varname*`}`. Additionally, the trigger direction needs to be specified, i.e., whether the event shall be triggered on a

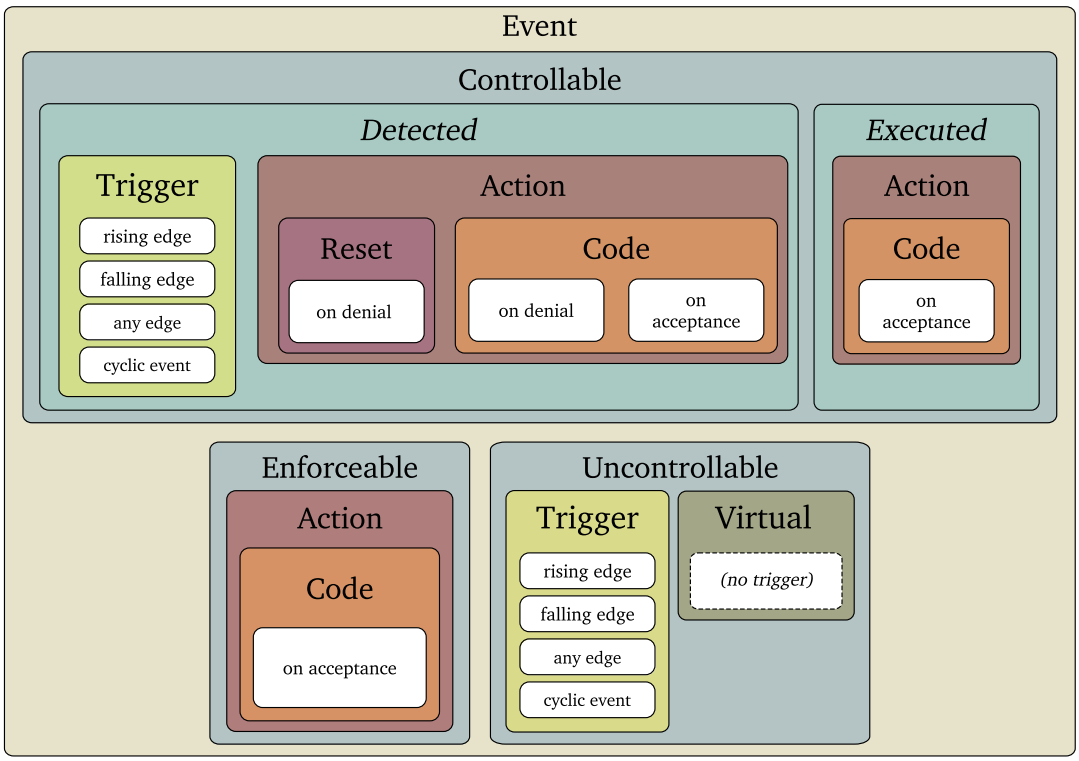


Figure 6.6: SynTACS event scheme, figure based on [Gatto, 2016]

rising edge, a falling edge or both. Rising edge means that the condition is true and has been false at least in the cycle before, while falling edge addresses the opposite. If the built-in edge detection is not sufficient, it is also possible to access the PIO values of the previous cycle using the macro `\last{`*address*`}`. Note that this macro cannot be nested as accessing the cycle before the previous one is not supported. Besides the two edges and their combination "rising or falling edge", a fourth option exists to realize *cyclic events* (see Section 5.6). These need to be re-approved by the supervisor in every cycle, i.e., the event is triggered every time that the condition evaluates to true.

Detected events allow the supervisor to work with arbitrary controllers which have not been specifically designed for the framework nor know about being supervised, one of the objectives stated in Section 5.1. If the controller is developed in parallel to the SynTACS model, it is also possible to *execute* controllable events directly from the controller. Every event is given a unique ID in the framework, which is stored in a constant, named after the *qualified name* of the event. To execute an event, a special function block must be called, passing the event's ID as argument. The event is then checked on the supervisor right away as the controller is running and executed in case of approval.

Uncontrollable events are always *detected* by the framework in the same manner and using analogous triggers as controllable ones, except they are usually formulated over the addresses of the PII. Cyclic uncontrollable events are technically possible but should be avoided as they are logically not sensible. Enforceable events apparently have no trigger.

Virtual Events It is often the case that automata provided by the user contain uncontrollable events which are prohibited in all states where they can occur as they are always undesired, e.g., over-heating or damage. Usually there is no dedicated sensor hardware to detect this kind of events, thus a trigger cannot be formulated. Since synthesis is meant to transform the uncontrollable specification into a controllable supervisor, these events will be dropped during the process anyway. They are called *virtual events* as they will never happen under supervision. A supervisor may of course not contain virtual events anymore. If one still does, an error is displayed during code generation.

Action Classes Recall the three action classes from Section 5.3.2, *Reset on denial, Code on acceptance* and *Code on denial*. When the user enters a trigger condition, SynTACS lexically scans that string for PIO addresses and displays one checkbox for each address below. By these she can select which outputs shall be reset in case that the event has been denied while the others stay unaltered. For the action classes *Code on acceptance* and *Code on denial*, the user needs to explicitly specify the ST code that is to be executed in the respective situation.

6.3.2 Supervisors

The code representation of synthesized supervisors is the core element of SyRF. The first version of the code generator, realized in the context of [Timmermanns, 2015], translated every supervisor to a simple lookup table in the shape of a two-dimensional array. This implementation had several drawbacks though. On the one hand, it turned out that the states of realistic supervisors are not very strongly connected, meaning that the majority of table entries represented loops. On the other hand are arrays on most PLCs stored in the main memory.

In the revised version [Gatto, 2016] the transitional structure is entirely implemented in terms of conditional jumps (usually via an extensive *switch/case* construction). This way the entire supervisor is kept in the usually much larger program storage. Further can the memory consumption of implicit loops, i.e., the absence of a transition or prohibition for an event, be reduced to zero.

All supervisors are accessed via one PLC function (POU of type FUN) which calls the functions of the corresponding supervisors and evaluates their results.

6.3.3 Timer Management

In Section 6.1.3, the definition and usage of timers within the SynTACS models has been explained. Apparently, the employed timers must be functionally realized in the SyRF too. Similar as for the supervisors, one function block manages all timers. Internally, each timer is represented by an on-delay timer (TON) instance. The *timer manager* instantiates all timers, starts and resets them and calls them on every cycle to update their status. When a timer has reached its user-defined value, the *timeout* event is triggered.

6.3.4 Procedure

Figure 6.7 shows the control flow and internal communication of SyRF during one PLC cycle schematically. At the beginning, i.e., right after the PII has been sampled, some initializations are performed. This includes refreshing the timers by calling the corresponding TONs. Further, all signals from the last cycle are captured that are relevant for edge detection or for triggers that use the `\last{}` macro on the corresponding signals. In the second step, the triggers of all uncontrollable events are evaluated and the supervisor FUN is called on all detected occurrences of these, including timeouts that are due.

Next, the embedded, user-defined controller routine is executed. Meanwhile, the framework propagates *executed* controllable events immediately to the supervisor and receives the result (authorized or denied). Authorized events are further transmitted to the timer manager which, if applicable, starts the timers associated with these events. After the controller cycle has finished, the framework checks the triggers for the remaining controllable events and processes the *detected* ones accordingly. The following step resets the signals that were manipulated by *detected events* which the supervisor has rejected, except they provide *code on denial* instead. Thereafter, the actions of granted controllable events are carried out, followed by the enforced ones. For the events which were explicitly *executed* by the controller, the original execution order is kept. For *detected events* this is not possible. Therefore, the user should be very careful when using events that operate on the same variables/signals and are not order invariant. Finally, the framework executes the specified *code on denial* for rejected events.

The workflow sketched above describes the generic shape of the SyRF. The actual code varies slightly depending on the selected PLC dialect. For instance does Siemens STEP 7 not support functions with a non-elementary return type. This is why in that case the supervisor is generated as an FB POU with empty data instead of a FUN.

6.3.5 Generation

The entire SyRF is generated dynamically from a SynTACS project. Technically, this is realized by using the *Apache Velocity Engine*[6].

The user can trigger code generation using a button in the toolbar. A wizard shows up in which he can select an output format, the supervisors to include in the framework and a target folder. The currently implemented dialects are IEC-compliant Structured Text, Siemens SCL for S7-300/400, Siemens SCL for S7-1200/1500, CoDeSys-compliant ST and PLCopen XML with embedded IEC ST. In addition, for S7-1200/1500 a *remote supervisor* option exists, which is introduced in Section 6.5.

Velocity uses textual *templates*[7] for code generation. Besides plain code, these contain *directives* for case distinctions, repetitions (loops) and to access, evaluate or insert data from the *velocity context*, an object that has to be filled with key-value pairs before generation. SynTACS uses it to store all information which is relevant to generate the SyRF, such as transitional structures, event definitions, actions, triggers, and so forth.

[6] Project website: `http://velocity.apache.org`

[7] Do not confuse with automaton templates (Section 6.1.4).

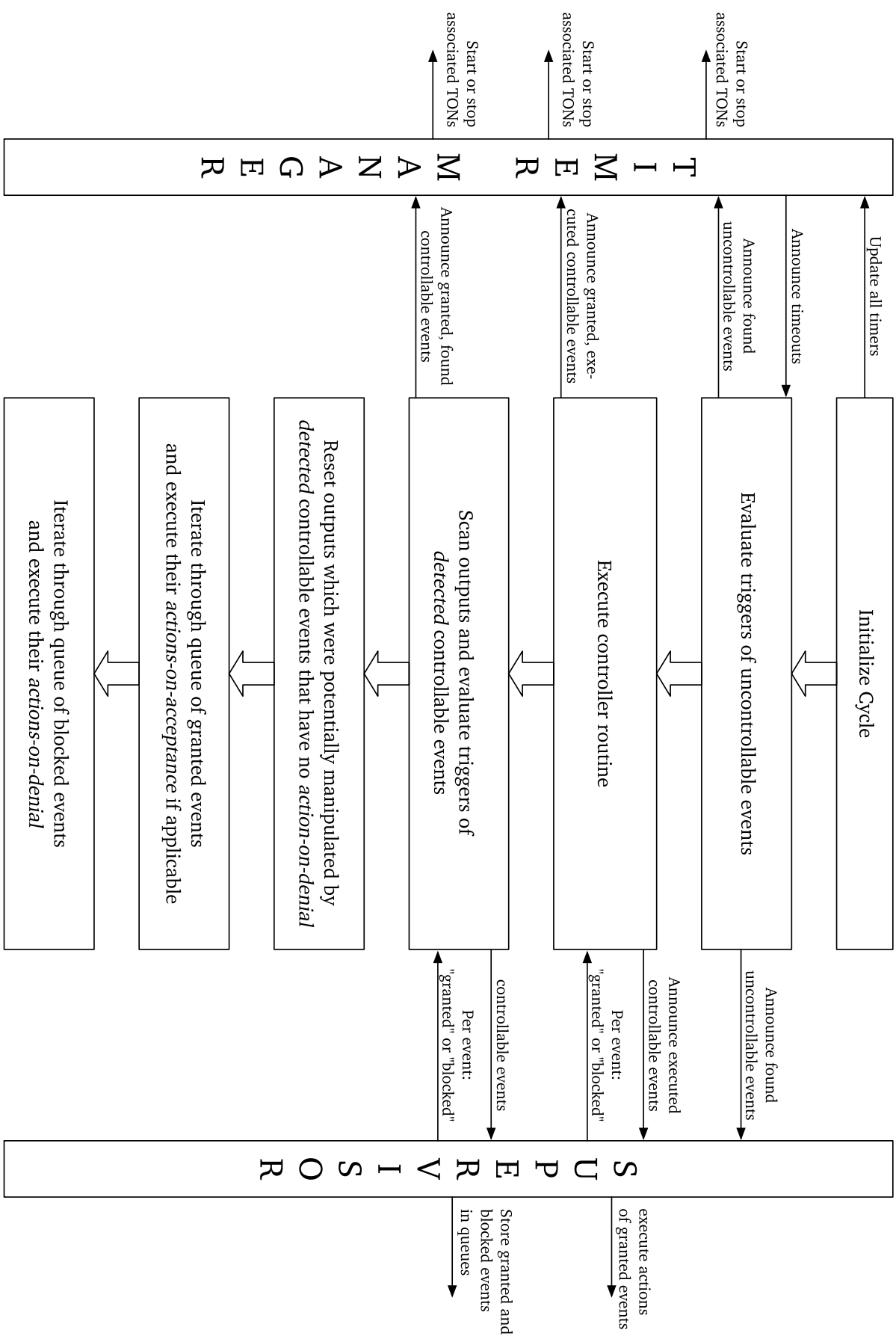


Figure 6.7: The SynTACS Runtime Framework
(Figure taken from [Gatto, 2016], translated from German)

6.4 Limitations

The SynTACS Runtime Framework allows a minimally restrictive supervision of existing controllers thanks to the maximal permissiveness of the implemented syntheses on the one hand and the non-invasive event detection on the other. Particularly the latter comes with some drawbacks though, which should be mentioned.

6.4.1 Event Order

The framework is able to keep *executed* controllable events in their proper order. Apparently, that is not possible for *detected* events, neither for controllable nor uncontrollable ones. However, this limitation is fortunately not as significant as it may seem at the first glance. Consider two physical events which occur in quick succession. If controls shall depend on which of them occurred first, the controller must in any case be hosted by sufficiently fast hardware in order to reliably determine the chronological order of these events. In return, SynTACS assumes that two events which occurred within the same cycle, and hence are detected simultaneously, are order-insensitive.

The same holds for the controller's output signals. Again, the hardware would have to be fast enough to propagate order-sensitive signals and operations in their accurate order, i.e., at least in two subsequent cycles. Since the control designer should be aware of the fact that the entire PIO is applied to the hardware outputs of the PLC simultaneously, SynTACS in this case also assumes the order insensitivity of the corresponding event pairs.

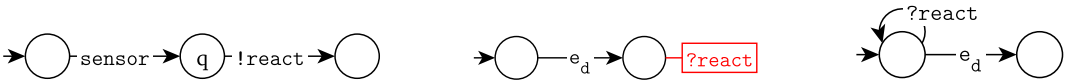
Nonetheless, the supervisor needs to fulfill the property of *interleave insensitivity*, discovered by Fabian and Hellgren [33], to tolerate arbitrary orders of these events. SynTACS does not include a method for that yet but an algorithmic check is proposed by these authors, which could be integrated in the tool.

Currently, the order of events being detected and executed is arbitrarily determined during code generation. Future versions of SynTACS should allow the user to define a customizable event prioritization.

6.4.2 Synchronization of Enforced Events

In Section 5.5.1, it is motivated why allowing unstable states is crucial for the supervision of realistic systems. Preemption by enforced events is one way to cope with that issue and has been successfully implemented in SynTACS. Although it yields well-usable results and behaves as expected in practice (cf. Section 7.2), some limitations exist.

Supervisors do not synchronize on enforceable events. Thus, although technically possible, enforceable events should not be prohibited in specifications. Nevertheless, if an enforceable transition leads to a state hosting an uncontrollable prohibition, synthesis would automatically illegalize this transition, i.e., convert it to a forbidden event. The situation of a prohibited enforceable event which is at the same time enforced by another supervisor cannot be resolved during runtime anymore. It is possible though to check the sets of enforced and prohibited events of all supervisors for disjointness and show a warning to the user if they share an event. In that case, she can still compose the corresponding



(a) Specification H (b) Specification H_d (c) Plant Model G_d

Figure 6.8: Problematic enforcement-by-Design

specifications first and let the tool synthesize one supervisor for both. This way, the conflict is avoided and the event would only be enforced in states in which it is uncritical. Note that these two sets being disjoint is a sufficient but by no means a necessary condition for conflict-free supervisors. However, using the composition can involve larger state spaces and higher cost of computation but is obviously never less permissive within the scope of legal solutions.

Within the current version of the tool, one supervisor is not able to track the events enforced by another supervisor. This has several implementation-related reasons. First, the necessary communication between the supervisors could require multiple iterations until all enforcements are processed. For instance, when one supervisor enforces an event f_1, a second tracks it and decides to enforce f_2 thereafter, which again induces the first supervisor to enforce another event f_3, and so on. These constellations make it difficult to estimate the runtime of the framework and thus the cycle time. Unbounded loops (`while` loops) are in general undesired in PLC programs for the sake of runtime predictability. The second reason is that the runtime framework cannot distinguish between enforceable and enforced transitions. Hence, all enforceable but not yet enforced transitions are removed from the automaton before it is finally converted to a supervisor.

6.4.3 Manually Enforced Events

SynTACS allows the user to directly define explicitly enforc*ed* events. This is done by adding an enforceable event to a specification's transition. This way of explicit enforcement should be used carefully as it can be delicate.

Enforcing an event via the specification does not guarantee that it is eventually executed as this is not covered by the preemption contract. In particular, the user should not prescribe critical reactions on events right away. The following example illustrates that.

Example 6.3. Consider the specification depicted in Figure 6.8a. It specifies an instant reaction on the event `sensor` in the shape of the enforced event `react`. But, if it is composed with the specification shown in Figure 6.8b, the reaction would be disabled once e_d occurred. Even if no specification forbids any enforceable events, as recommended above, there can still be a plant model which indicates that the reaction is not even physically possible after e_d, such as the one in Figure 6.8c. In both cases `react` cannot be executed, something the designer of H did probably not take into consideration. △

This example shows that enforcement does not replace a proper declaration of undesired happenings. If the user added an uncontrollable prohibition to state q in Figure 6.8a,

synthesis would disable e_d in both cases to ensure that `react` can take place, provided that H_d or G_d are composed with H. Still, it is recommended to omit enforced-by-design transitions and instead provide enforceable transitions to safe states within the plant models. The reason is that incremental synthesis automatically consults all plant models regarding the feasibility of enforceable transitions before choosing one. When monolithic synthesis is invoked, the user can apply the relevant plant automata manually and this way obtain a correct solution.

6.4.4 Conflicting Transitions

SynTACS allows multiple transitions with the same event leaving the same state as long as both transitions are conditional. The logic mutual exclusion of both conditions is not checked ex-ante although this would technically be possible using satisfiability checkers. It is not necessary though as all conditions need be resolved by composition before or during synthesis anyway. As soon as SynTACS detects two transitions at the same state which eventually both are evaluated to true, i.e., proper non-determinism, an error message is displayed to the user referring to the problematic transitions so she can fix the conditions.

Unconditionally conflicting transitions are not allowed. When an *e*-transition is added to a state where an outbound *e*-transition exists already, SynTACS prompts the user to enter a condition for the new transition. The original one automatically gets the negated condition by default to avoid inconsistent models.

6.5 Remote Supervisor

On conventional PLCs, both program and data storage is usually very limited. Particularly the transition function can become very large for complex scenarios. In the context of [Osetinski, 2017] it was evaluated whether it is possible to detach the transition function from the rest of the framework and relocate it onto an external device, which adaptations need to be performed for that and which restrictions apply. For the case study, the Texas Instruments *AM3359 ICEv2*[8], in the following ICEv2, was used to host the transition table (the *remote supervisor*). It is shipped with the proprietary real-time OS *SYS/BIOS* and a C compiler. The remainder of SyRF and the controller were to be executed on a Siemens S7-1516 PLC.

6.5.1 Communication

For the exchange of data between both devices, a communication protocol was designed on top of the *process field bus (for) decentralized peripherals* (PROFIBUS DP). High data rates are not required for this purpose. The real-time capabilities of the underlying bus system, however, are crucial as an indefinite delay of control actions is not tolerable. This, along with the good availability of PROFIBUS on PLCs made it the means of choice.

[8] Technical identification: TMDSICE3359

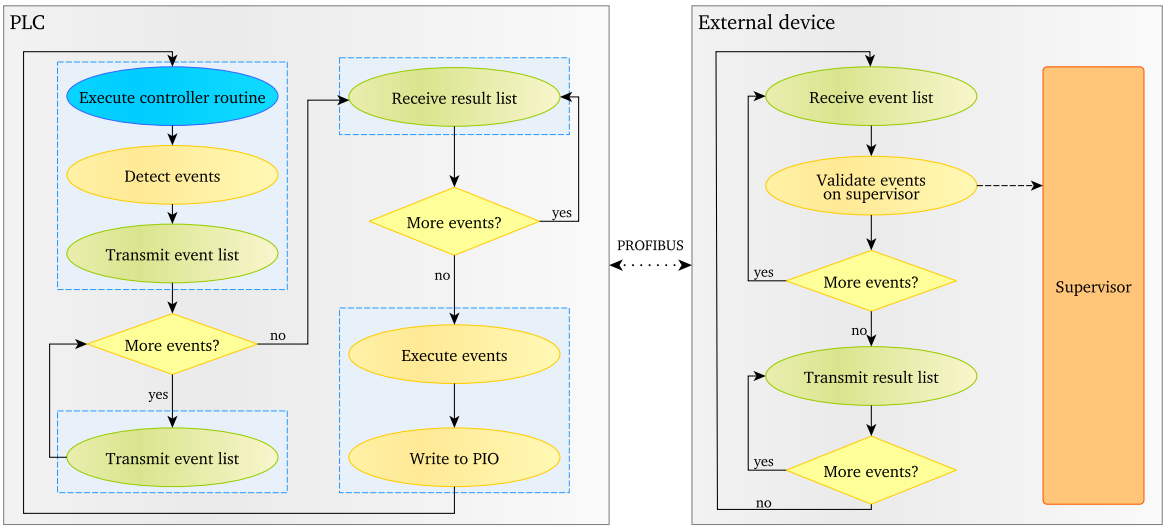


Figure 6.9: SyRF in the remote supervisor setting
(Figure taken from [Osetinski, 2017], translated from German)

Both parts of the framework need to be generated at the same time so they fit to each other. When the communication is initialized, this is checked using a randomized fingerprint. If the result is positive, the actual supervision can start. The PLC transmits the list of event occurrences to the remote supervisor, which executes them on its transition table. The result, a list of *"authorized"* and *"rejected"* notifications followed by the list of enforced events, is sent back to the PLC, which performs the necessary operations. For technical reasons[9], very long event lists of more than 238 occurrences have to be split into multiple *segments* which are transmitted subsequently. The same holds for the result and enforcement lists.

6.5.2 Cyclic Execution

The main challenge for the adaptation of the original SyRF is the asynchronous communication paradigm of PLCs when communicating with peripherals. The PLC treats data that shall be submitted to these as output signals and received data as inputs. As a consequence, data is sent and received only once a cycle in analogy to the digital and analog I/O ports. This method is tailored to field devices such as sensors and actuators, which are often connected via PROFIBUS. In the considered scenario, however, bidirectional communication would be necessary within the same cycle.

For that reason, SyRF is transformed to a sequence of multiple steps which are, similar to SFC, distributed over several succeeding cycles, as sketched in Figure 6.9. Technical details such as executing actions, resets or timers are omitted in the illustration for reasons of clarity. Every dashed box on the left-hand side represents one or even multiple cycles of the host PLC, including the corresponding in-/output phases. These are used to transmit

[9] A closed-source PROFIBUS driver on SYS/BIOS with a maximal supported segment length of 240 Bytes.

and receive the event and result lists mentioned above. Note that the controller is executed only once during the entire sequence which makes the latter a *logic cycle*.

Thanks to, first, both hardware components being real-time capable as well as SYS/BIOS and PROFIBUS and, second, that the maximal count of event occurrences is limited by the number of different events[10], the maximal duration of one logic cycle is bounded by a constant.

6.5.3 Evaluation

The presented framework for remote supervision has been evaluated on several examples. In particular, it was tested on generated supervisors with large state spaces in order to show its suitability to tackle the addressed issue of memory consumption. These supervisors had n_z states, $q_1, \ldots, q_{n_z}$, and were defined over n_e different events. Each state had outbound transitions to three other states, labeled by randomly picked events, such that all of them were reachable. The only exceptions were $q_1, q_2, q_{n_z-2}, q_{n_z-1}$ and q_{n_z} which had less transitions.

As expected, it could be observed that the size of the PLC part of the framework grows roughly linearly over n_e while n_z only affects the size of the transition table, stored on the external device [Osetinski, 2017]. The largest tested supervisor had $n_z = 60{,}000$ states and 179,994 transitions over $n_3 = 10$ different events. While the generated PLC framework consumed only a few kilobytes of memory (9 KB of POUs and 2 KB of data) during runtime, the binary of the transition table was larger than 6 megabytes and, by that, would on its own exceed the memory capacity of most ordinary PLCs.

Using that supervisor, the average duration of one *logic cycle* was 12 ms, fast enough for plenty of factory- or process-related applications. In further experiments, a linear correlation between the number of events that were detected in a cycle and its duration was observed [Osetinski, 2017]. This is not surprising as the size of transmitted messages mainly depends on that number. For 100 event occurrences, the average cycle time was 16 ms on average, for 1000 it was between 557 ms and 590 ms. In contrast, the state space size and transition density had nearly no impact on the performance thanks to the very efficient realization of *switch/case* statements by most C compilers[11].

Since the ICEv2 only has a flash memory of 8 MB, it was not possible to compile significantly larger supervisors for that device. However, thanks to the fact that the generated C code neither involves any hardware-related aspects nor does it require specific drivers, it should be possible to adapt the remote supervisor framework to other devices without too much effort.

In general, it can be stated that outsourcing the memory-intensive transition tables to an external device is feasible if a real-time capable communication channel is used. The performance decreases only locarithmically over the size of the supervisors but suffers from large numbers of event occurrences within a short time.

[10] Given that an executed event is not triggered multiple times within the same cycle, which does neither make sense from the controls's point of view nor is it supported by SyRF

[11] Theoretically, *switch/case* has logarithmic runtime on large blocks of succeeding integer entries

Chapter 7

Evaluation

In order to estimate the suitability of the approach and the tool SynTACS for the purposes described in chapters 4 and 5, an evaluation of the following aspects is required.

- How well are unexperienced users able to express requirements and behaviors using the modeling principle introduced in Section 5.3?
- How is the latter improved by the advanced modeling concepts such as conditional transitions or templates?
- How good is the general usability of SynTACS?
- Is it able to supervise a real controller that is attached to a physical plant; does it behave as expected and which restrictions apply?

These questions have been investigated by [Aydin, 2017]. The first three of them were covered by a user study. Its setting as well as the findings are summarized in the following Section 7.1. The general applicability of the methods, the tool and finally the generated code have been demonstrated on the M3P.AC, a didactic process plant situated at the RWTH institute of process control engineering [62]. This study mainly addresses the last question.

7.1 User Study

In early 2017, a user study was conducted to evaluate their suitability for the designated purpose. It has been presented in [41]. Thus, this section is mainly based on that publication.

7.1.1 Object of Investigation

The study was intended to independently assess the three aspects mentioned above in the context of controller supervision: appropriateness of automata/SCT in general, improvement by the advanced concepts mentioned above and usability of the tool. It consisted of three tasks complemented by two questionnaires, one to query the background and

foreknowledge of the participants before accomplishing the tasks and the second to collect their experiences thereafter.

In the first task, the subjects were asked to model a simple specification and a plant monolithically only by pen and paper. The second one involved a slightly more complex scenario that should be expressed with modular models including conditional transitions and prohibitions, again handwritten. The reason why these tasks were designed as pen-and-paper exercises is that any distortion of the results by limitations and benefits of the tool should be avoided. Finally, the participants had to implement specifications and plant models in SynTACS. This included the usage of automaton templates and instances. This time, all events and automata were given to exclude modeling difficulties and thus provide the same circumstances for all participants when using the tool for the sake of more credible results.

7.1.2 Procedure

After filling out the first questionnaire, the participants were given a short introduction to the use case, to DES and SCT[1], and finally to SynTACS. Five A4-sized cheat sheets were used to support the explanations. The participants were allowed to use these sheets during the entire case study. Before each exercise, the subjects were presented a sample solution for another but similar example to get a hint about the semantics of the elements.

Exercise 1 – Collision Avoidance System In the first task, the necessary models to synthesize a supervisor for a crane controller should be created. The crane trolley was assumed to be mounted on a rail at the ceiling, able to move to the left or to the right in terms of discrete steps. At both ends of the rail, sensors were installed to indicate that the left- or rightmost legal position has been reached. All five events (`stepRight`, `stepLeft`, `sensorRight`, `sensorLeft`, `crashLeft`, `crashRight`) were given including their triggers. The subjects were asked to provide a plant model for one of the two sides that correctly defines the situations in which a crash can potentially occur. Additionally, the specification that illegalizes that crash should be modeled. For a proper solution, the experimentees had to consider several aspects: They needed to distinguish between what is (il)legal, what is (im)possible and how these statements are mapped onto the model. According to the plant contract, a plant model needs to *over-approximate* the entire possible behavior (plant contract) with respect to its alphabet. Yet, it must be precise enough to make the specification realizable.

Exercise 2 – Workpiece Singularizer The second exercise involved two hardware components: A conveyor belt and a reservoir equipped with a clasp. When the clasp was opened, one workpiece was dropped onto the conveyor. After that, the clasp should automatically close again, indicated by a sensor. The belt was able to be started or stopped. The subjects should provide a specification that prevents objects to stack on the belt. The necessary modular plant model needed to capture that stacking is only

[1] The adapted shape of SCT as introduced in Section 5.3

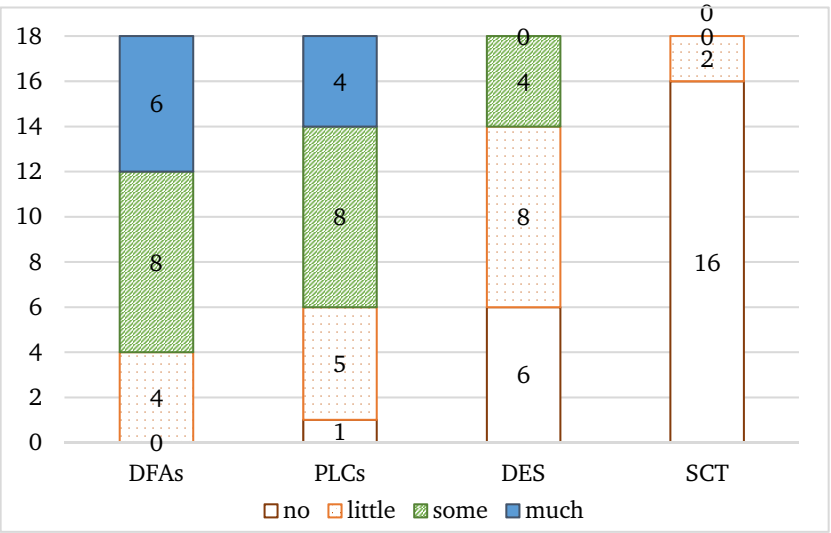


Figure 7.1: Foreknowledge of the participants (Figure based on [41])

possible while the conveyor is standing and the clasp is open. This time, the events had to be identified and defined on the described actors and sensor signals by the participants on their own.

Exercise 3 – Using SynTACS In the last exercise, a sample solution for the crane example from the first task was given for both sides, left and right. The subjects should use SynTACS to synthesize the corresponding supervisors. For that they were provided with the scaffold of a SynTACS project, consisting of one module and four predefined events. First, they had to add the two missing events to the project.

Since the considered problem is symmetric, the given plant models and specifications have the same shapes for both sides. Thus, the participants were asked to set up one template for each kind and instantiate these for either side by using the respective events. Finally, they should synthesize a supervisor from each specification instance using both available methods: Manual composition followed by monolithic synthesis, and the incremental procedure.

At the end, each attendee filled out the final survey.

7.1.3 Participants

18 persons attended the user study. 10 of which hold a Master's degree while 8 were students. All of them study or work in the fields of engineering or computer science. Nobody had previous experience with SynTACS nor were the subjects involved in its development by any means. While many had basic knowledge about classical automata theory, only few of them were familiar with DES or even SCT. Figure 7.1 shows the foreknowledge, broken down to these four categories.

7.1.4 Significance

The user study was carried out subsequently with one attendee at a time to avoid mutual influencing. To prevent disturbance, a separate room was used. Every subject got the same introduction and was provided with identical materials. It cannot be excluded for sure that information was shared amongst the participants between the experiments. However, a significant bias appears unlikely.

The strict decoupling of pure modeling tasks on the one hand and those involving the tool on the other hand allows a clear association between the results of the exercises and the objects of investigation. Concrete questions from the subjects were answered. Apart from that, only small hints were given if somebody got obviously stuck. In particular, no feedback about the correctness or quality of the provided solution was given.

7.1.5 Results – Modeling

The crane exercise was for all participants the first time that they had to design a DES model for a given system on their own. Seven of them created erroneous solutions with varying severity, while nine provided correct and sufficiently precise models. In all cases, the subjects had more difficulties in modeling the plant than the specification. Three common types of mistakes could be identified:

Over-simplification In many cases it could be observed that the expressiveness of the plant model was over-estimated. Models with too few states or transitions were the result. Two logically distinct states that were treated as one was a common mistake. That state usually contained the outbound transitions of both where the participants implicitly assumed that in one case the first and in another case the second transition would be taken. Such assumptions, however, need be made explicit in the models in order to yield a correct solution. If the plant model describes the behavior of the plant too restrictively, the resulting supervisor would be too permissive and hence eventually violate the specification.

Missing backwards transitions The provided models were often designed too tightly around the specification. In principle, that is uncritical as long as the plant contract is not violated. This, however, was the case quite frequently, especially due to forgotten backwards transitions. For instance, it was assumed that, once the crane from the first exercise has moved back from the end position, that position would become unreachable, i.e, according to the model a crash was not possible anymore.

Missing dependencies Some of the models were sound in the sense of the plant contract but incomplete, i.e., the abstraction was too coarse. The reason was that physical dependencies were not considered accurately. In that case, the result is either a safe but too restrictive supervisor or even a void supervisor if controllability cannot be established.

The second exercise was correctly solved by all eighteen attendees although it required more automata and events. Since the participants did not receive feedback about wrong

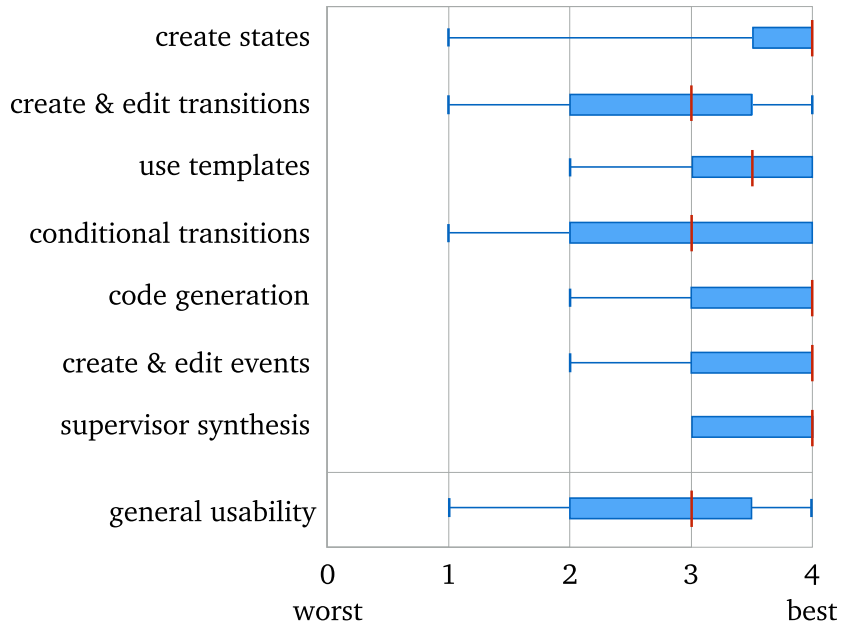


Figure 7.2: Usability rating for SynTACS (figure taken from [41])

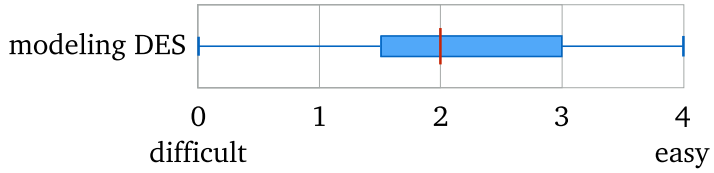


Figure 7.3: Degree of difficulty in modeling DES (figure taken from [41])

answers in the first exercise, it is unlikely that the significantly better solutions to the second are already the result of training or experience. Instead, it clearly points out how the manual process of modeling benefits from modularity and the possibility of restricting transitions to other automata's states by conditions.

7.1.6 Results – User Experience

The subjects used SynTACS for the third exercise. All users were able to accomplish the task with only minor difficulties. In the final survey, their experiences with the tool were polled in terms of seven categories: *create states, create and edit transitions, using templates, using conditional transitions, code generation, create and edit events* and *supervisor synthesis*. Further, the attendees were asked to rate the overall usability of SynTACS. All disciplines could be graded from 0 (worst) to 4 (best). Figure 7.2 shows the results as boxplots. Additionally, the participants were asked to grade the difficulty of modeling plants and specifications using DES as performed in the first two exercises. The corresponding boxplot is shown in Figure 7.3. Again, it must be emphasized that the participants only got a short introduction and had no former experience in modeling DES.

Figure 7.4: M3P.AC

Apparently, not only the first intuition and learning curve in using DES and SynTACS are relevant but also the experiences a user makes on the longer term after the initial barriers have been overcome. These will be discussed in the context of the case studies described in the following section.

7.2 Case Study: M3P.AC

In order to validate the suitability of SynTACS for the supervision of actual controller code, several case studies were carried out in the scope of [Aydin, 2017], addressing different applicational contexts. A summary of the results has been published in [41]. In the first and most comprehensive study, the safety measures of M3P.AC should be synthesized, a didactic laboratory plant which consists of real components as used in industrial settings. In the following, the case study on M3P.AC is summarized.

7.2.1 Setting

The M3P.AC is a simplified model of a wastewater treatment plant near Cologne, Germany [62] and usually serves for educational purposes, particularly in process engineering courses. It comprises 3 tanks, 5 pumps and 7 valves which are connected by pipes. Further does it provide 5 level, 6 temperature, 2 pressure and 4 flow sensors to monitor the current status of the plant. Figure 7.4 shows a photograph of the M3P.AC. The components are attached to the controllers via connectors which allow to quickly detach it from one PCS and connect it to another. For the case study, a Siemens S7-400 PLC was used.

The communication between the PLC's CPU and the field devices was managed by an *ET 200M* interface module which was connected to the CPU via PROFIBUS. The controller

under supervision was developed with SIMATIC STEP7 TIA-Portal V13 SP1 using the language SCL, Siemens' ST dialect. The same IDE served to finally migrate the generated code onto the controller.

7.2.2 Safety-Critical Requirements

While operating M3P.AC, it can come to several safety-critical situations. In the regular courses, the students are provided a list of aspects which they are supposed to implement safety measures for. Examples are preserving tanks from overfilling and pumps from running dry or working against closed valves. Furthermore, the liquid is not allowed to enter one of the tanks if the temperature difference is too significant.

Some of these measures are simple interlocks while others require unstable states being left actively, cf. Section 5.5.1. This already shows the necessity of either preemption or cyclic events when realizing such measures with SCT.

The productivity requirements to establish the desired function on the plant will not be discussed here as they are not relevant for supervision but only for the implementation of the controller. A short summary can be found in [Aydin, 2017].

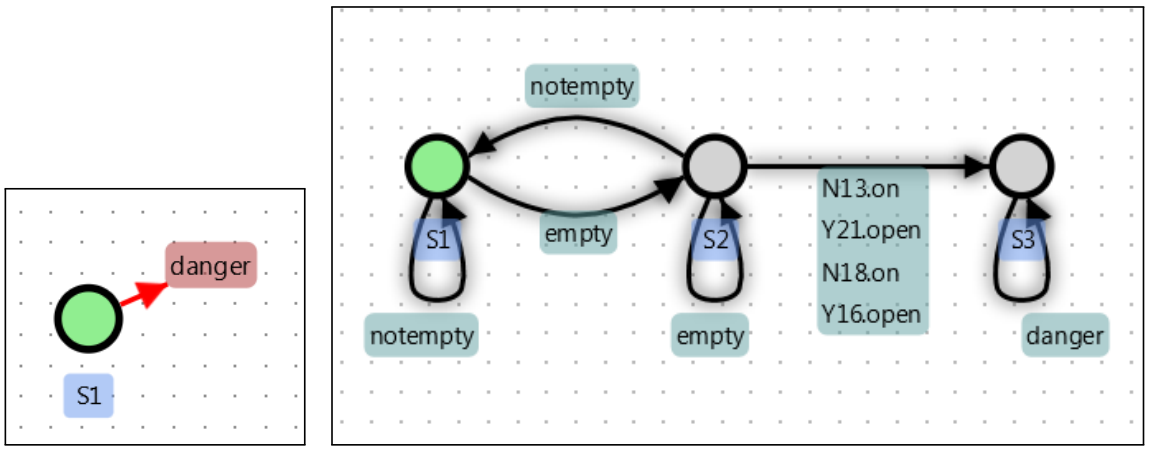
7.2.3 Synthesized Safety System

12 specifications were derived from the side conditions mentioned above. The dependencies of the overall 64 events were captured by 15 plant models. As an example, Figure 7.5 shows a simple 1-state specification prohibiting the virtual uncontrollable event `danger`, which combines several incidents. Next to it, a plant model is depicted which narrows down the situations where that event is imminent, namely one of the pumps `N13` or `N18` is started or one of the valves `Y21` or `Y16` is opened in the state `S2`. The latter is entered once the level sensor `L10`, evaluated by the triggers of the events `empty` and `notempty`, reports that reservoir `B1`, which is connected to these components, has fallen dry.

Some recurring behavioral models were successfully realized using templates and instances thereof.

From the specifications, 12 supervisors were synthesized and passed on to the code generator. The resulting SyRF was supposed to implement *all* side conditions required by the process documentation. Their integration with the controller succeeded without any problems. On a Siemens system, that involves the following steps: The content of the original main block[2] needs to be moved to another new function block as SyRF brings its own `OB1`. This new FB then must be called at the designated position inside `FB_Cycle` of the generated SyRF. The procedure for other vendors' PLCs looks similar.

[2] The *Organizational Block 1* (`OB1`) represents the entry point of the user-defined control program in STEP7 and can be compared to the PROG POU specified by the IEC [12, 55]



(a) Specification (b) Plant model

Figure 7.5: Example of a specification and a plant that is relevant to realize that specification. Original screenshots from SynTACS (Figure taken from [Aydin, 2017]).

7.2.4 Execution Procedure

The combined framework and controller were first tested on an industrial simulator, which mimics the entire behavior of M3P.AC, before they were executed on the real plant. The operation of the supervisor was monitored using the *online mode* of TIA-Portal, which allows to inspect the variables of the PLC during runtime. This way, and thanks to the long time constants of the plant in the order of minutes, it was possible to track blocked events as well as the actual output signals produced by the PLC.

In a second step, some bugs were introduced to the controller to deliberately violate one or several requirements. Additionally, the conventionally implemented safety measures were removed or disabled. The blocked events were again monitored along with the PLCs outputs sent to the actuators. The behavior of the physical plant was captured with a video camera. Eight different, possibly harmful situations were tested this way one after another or combined, covering every considered safety aspect at least once.

7.2.5 Evaluation

Given a correct controller, proper operation was possible without any interference by SyRF. However, it was observable that all critical events were indeed listed as disabled in the respective situations. In the second step, SyRF successfully blocked all operations which were either harmful or could lead to such uncontrollably. In these cases it was clearly visible that the output operations imposed by the controller were revoked by the supervisor before being applied to the outputs. As expected, the actuators did not respond in these cases.

Nearly all aspects involved safety-critical reactions in addition to interlocks. These were all reliably triggered and executed in the induced situations. That also worked in more complex cases which involved multiple supervisors as in the following observed situation:

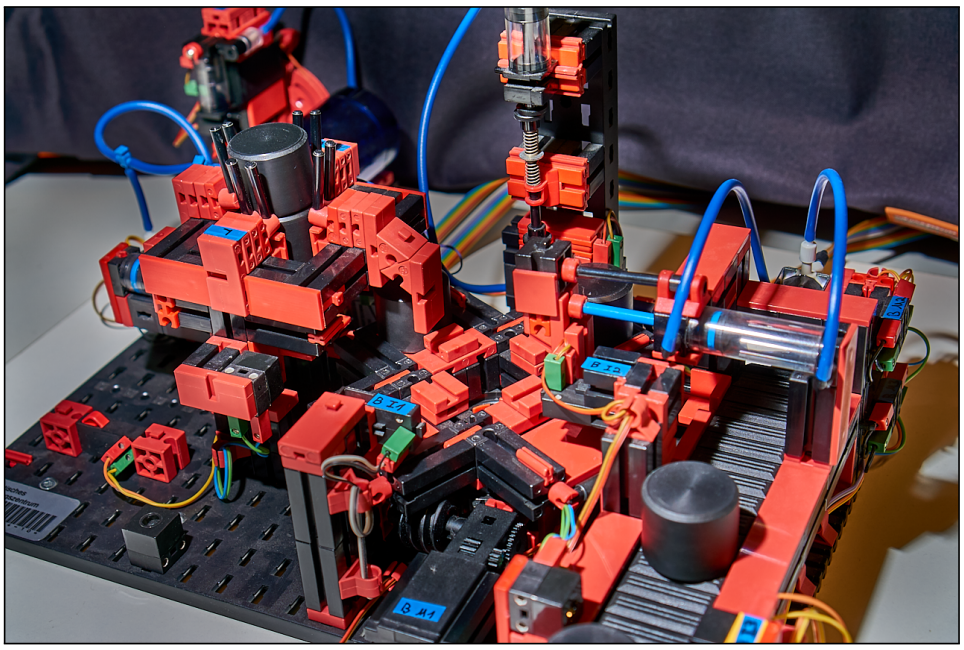


Figure 7.6: pneumatic processing center by fischertechnik®

> One of the tanks had reached its maximum level and the controller correctly closed its inlet valve. After that, the inlet was locked by one supervisor to avoid the tank to overflow. A little later, when the faulty controller tried to activate a pump upstream that valve and therefore open it again, the mentioned supervisor blocked the valve while another one disabled the activation of the pump.
>
> This also worked when the controller did not close the inlet in the first place. In that case, the first supervisor enforced the closing while the second enforced the pump to stop and locked it afterwards.

A table that summarizes all safety measures implemented by the synthesized supervisors can be found in [Aydin, 2017, p. 26].

7.3 Further Case Studies

Besides the M3P.AC, which has been controlled by a Siemens S7-400, further plants, PLCs and IDEs have been tested with SynTACS.

One of them is the fischertechnik® model plant *pneumatic processing center*, depicted in Figure 7.6 [41]. It takes discrete workpieces in the shape of cylinders from a magazine, one at a time, and pushes them onto a rotary table which has four cages and positions. The first is the entry position. At the second, a stamp is installed to punch the workpieces. The third position is equipped with a puller which draws the pieces onto a conveyor belt to deliver them to the next machine.

The PPC comprises several safety-critical aspects: The stamp and workpiece can be damaged if the rotary table moves while the piece is being punched. The pusher that fetches the workpiece from the magazine may only do so if the currently proximate cage of the table is free. The same holds for the puller, which may neither operate while the conveyor is moving nor before the previous workpiece has been carried away and the way is clear. Furthermore, there is a barrier above the fourth position of the rotary table, thus only empty cages may be moved in this position.

From these aspects, four supervisors were synthesized with SynTACS. Their generated code was tested on two different PLCs, a Siemens S7-300 and an ABB PC500[3]. As for the S7-400, STEP 7 TIA-Portal was used for the S7-300. The ABB system was programmed using ABB Automation Builder 1.2 which includes CoDeSys 2.3 for the controller implementation. Both systems do not support PLCopen XML imports, hence all generated POUs were imported manually. SynTACS offers a dedicated generation profile for the CoDeSys dialect of Structured Text as well as separate S7-300 and S7-400 SCL exports as these languages differ in terms of available data structures.

As for the M3P.AC, no disturbance of the regular workflow could be observed when a correct controller was used on either PLC. When bugs were introduced, e.g., the stamp moving down before the rotary table has stopped, the supervisors successfully obstructed critical actions such that the safety constraints described above were not violated.

It should be mentioned that applications from the domain of discrete manufacturing rarely involve hazy interdependencies of uncontrollable events. Hazards can almost always be averted by blocking the specific action which is directly related to the incident that shall be avoided, since machines and workpieces involve much less internal dynamics as, for instance, chemicals do.

Last, some small conceptual plants were simulated on CoDeSys 2.3 in order to evaluate some border cases. It could be observed that SynTACS does not check or guarantee *interleave insensitivity* yet, cf. Section 6.4 and [33]. The example of a washing machine [Aydin, 2017] shows that, even for interleave-sensitive supervisors, it would be desirable to allow user-defined event priorities. Besides, SynTACS is not able to check logic coherences between the triggers of two events. If a supervisor imposes two prohibitions which exclude each other logically, e.g., one prohibits an output being set to `true` while the other one forbids `false`, the result is not well-defined. Satisfiability checkers could help detecting these issues. Unfortunately, this would require the entire semantics of the target language being provided to the checker to cover all possible cases. Furthermore would it be necessary to exhaustively enumerate all reachable combinations of prohibitions amongst multiple supervisors, which, in the worst-case, requires their full composition.

The PLCopen XML generation was evaluated for earlier versions of SynTACS on CoDeSys 3.5 and the IDE TwinCat by Beckhoff Automation, although not with all currently available special functionalities yet (Timers, etc.).

[3]The particular CPU modules were an S7-313C (Siemens) and an PM554 (ABB)

7.4 Long-Term Usability

Besides the intuitive usability of SynTACS and the underlying modeling concepts for novices as investigated in Section 7.1, their long-term usability is essential. Therefore, the case studies sketched above were mainly performed by a computer science student who had no former knowledge about SCT. Her experiences can be read in her bachelor's thesis [Aydin, 2017].

Summarizing, she learned quickly how to apply the modeling concepts of SynTACS. Initially, she tended to model supervisors instead of specifications and plant models straightahead. This is problematic as some scenarios often stay unconsidered which can result in a less permissive or even unsafe supervisor. Also did she try to condense too many aspects in one plant model which often resulted in too complex and error-prone automata. However, she internalized the idea of modular modeling after only a few days. In general, she appreciated the concept of explicit prohibitions as she perceived it as more intuitive and the safety requirement was apparently visible in the specification:

> "Zudem ist in SynTACS die Sicherheitsanforderung anhand der Spezifikation sofort erkennbar. Somit ist die veränderte Modellierungsform intuitiver und im Aufwand reduziert." [Aydin, 2017][4]

Nonetheless, she observed that, although the distinct modeling paradigms for plants and specifications exist purposely in SynTACS to emphasize their different semantics, one sometimes forgets to add the necessary loops to the state of plant models. An automated function *add missing loops*, which adds loops to all states which must be deleted manually in order to specify that an event is *not* possible, could help.

The concept and realization of templates and conditional transitions turned out very useful and particularly reduced the effort on very large models with similar components. The action class *reset on denial* needs revision because it only supports hard-coded addresses whereas symbolic identifiers are not recognized [Aydin, 2017].

The missing undo/redo functionality for the automaton editor [Aydin, 2017] has been implemented in the meanwhile.

7.5 Benchmark of Incremental Synthesis

The strength of the incremental synthesis introduced in Section 5.8.3 is its strict limitation to the scope of the specification's needs. Consider the transfer line (TL) example introduced by Zhang and Wonham [88]. Figure 7.7 shows the topological structure of the plant schematically. It consists of two machines M1 and M2, and two buffers B1 and B2, which can each store up to two workpieces between and after these have been processed by the machines. The test unit TU checks the workpieces and either outputs them or sends them back to B1 for reprocessing.

[4] Translation by the author: *Besides, the safety requirement is instantaneously apparent in SynTACS. Therefore is the altered kind of modeling more intuitive and requires less effort.*

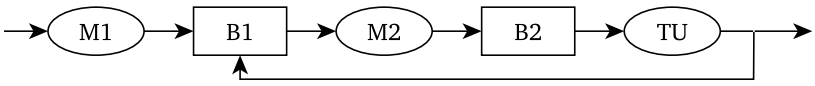


Figure 7.7: Transfer line (Figure redrawn after [117])

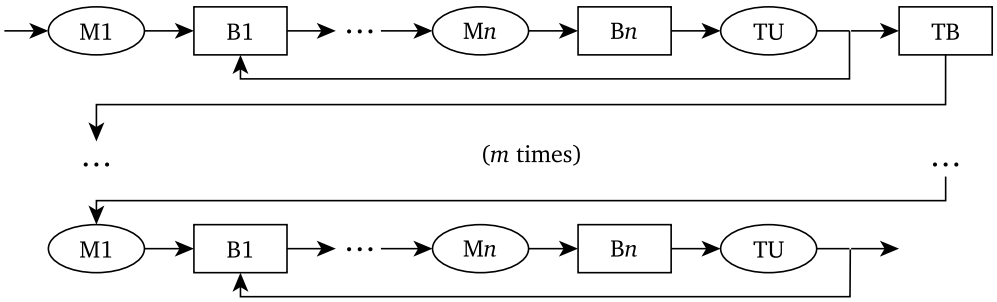


Figure 7.8: Several connected TLs with multiple machines each

The TL is characterized by the fact that just the machines and the test unit are modeled as plants while the behavior of the buffers is only limited by specifications. These prescribe that a buffer *may not* provide a workpiece when it is empty and that it *may not* take more workpieces than a specified capacity (3 and 1 in the original example; here we assume capacities of 2 pieces for each buffer). Although there appears to be no natural justification which leads to that kind of modeling, as obviously the buffers are *physically* limited to a certain capacity and particularly *cannot* hold a negative number of workpieces, this structure of alternating specifications and plants is beneficial for certain synthesis techniques. The fact that all plant models' alphabets are pairwise disjoint, as well all specifications, allows for *"decentralized supervision"* [117]. However, using conventional monolithic or even modular-specification methods, it is still up to the user to decide which automata should be composed and which ones should not. This becomes apparent when multiple TLs are connected as in the evaluation of [123] and/or the number of machines inside one TL between its TU and the return buffer B1 is increased. Figure 7.8 shows the setup of several connected TLs as used for this benchmark. An additional buffer TB has been introduced between the TU of one TL and its successor, resulting in $|\text{machines per TL}| \cdot |\text{TLs}| - 1$ buffers and thus specifications in total.

Table 7.1 lists several such modifications and subsequent concatenations of the TL along with the cumulated size of the resulting supervisors and their computation time on an Intel Core i5 750, 4 × 2.67 GHz, 4 GB RAM. The column MonSyn represents the results for a full composition of all plant and specification automata while ModSyn carries out a single synthesis for every specification yet still on a monolithically composed plant model. Apparently, both methods fail on more than 2 TLs or TLs with more than 2 machines due to the large state space. TO/MO denotes that the computation did not terminate within 10 minutes or the process went out of memory.

	MonSyn	ModSyn	iModSyn	IncSyn
2 TLs, 2 machines each (6 plants, 5 specifications)				
time	25 s	985 ms	96 ms	164 ms
# supervisors	1	5	5	5
Σ supervisor states	9000	5994	114	38
2 TLs, 4 machines each (14 plants, 13 specifications)				
time	TO/MO	TO/MO	165 ms	242 ms
# supervisors	1	13	13	13
Σ supervisor states			306	112
3 TLs, 2 machines each (9 plants, 8 specifications)				
time	TO/MO	60 s	109 ms	186 ms
# supervisors	1	8	8	8
Σ supervisor states		262440	180	60
4 TLs, 2 machines each (12 plants, 11 specifications)				
time	TO/MO	TO/MO	134 ms	194 ms
# supervisors	1	11	11	11
Σ supervisor states			246	82
5 TLs, 2 machines each (15 plants, 14 specifications)				
time	TO/MO	TO/MO	194 ms	238 ms
# supervisors	1	14	14	14
Σ supervisor states			312	104

Table 7.1: Experimental Results

The method iModSyn utilizes the *Dep* criterion by [3], adapted to the setting of SynTACS as explained in Section 5.8.3 (page 105). Due to the limited horizon of synchronization in the TL example, it performs very well and comparable to the incremental synthesis.

This changes dramatically, when events are introduced that are shared amongst several plants. Therefore, the abstract machines inside the TL were replaced by a more detailled modeling consisting of three automata each: a *process model* (6 states), a *drill* (5 states) and a *cooling unit* (2 states). The considered modified TL contains three of these "extended machines". As for the original TL these are topologically connected by buffers which are only modeled as specifications. Further, one global *sensor* (3 states) is introduced that shares an event with the *process model* of all three machines. Although the latter does not play a role for the considered specifications, it causes all plants to be in the *Dep* set of all specifications, making the problem intractable for iModSyn. Indeed, the algorithm stopped while generating the third supervisor due to an out-of-memory exception. Table 7.2 shows the results. The numbers in brackets refer to the computation of the first two supervisors.

	MonSyn	ModSyn	iModSyn	IncSyn
1 TL, 3 "extended machines" (15 plants, 4 specifications)				
time	TO/MO	TO/MO	TO/MO [*40 s*]	156 ms
# supervisors	1	4	[*2 of 4*]	4
Σ supervisor states			[*223608*]	48

Table 7.2: Experimental Results for modified TL with globally shared sensor events

Contrary, the incremental method only synchronizes with plant models that provide information about uncontrollable prohibitions, which the sensor does not. It hence needs significantly less supervisor states and terminates quickly although all plants transitively share events.

Chapter 8

Conclusion

The automatic derivation of software solutions from a description of the problem that shall be solved was and still is one of the most inspiring but also challenging ambitions in the field of computer science. Numerous approaches exist to achieve that goal in very different ways. One amongst these is the supervisory control theory by Ramadge and Wonham [87]. It provides a formal calculus to synthesize supervisors for discrete-event systems based on regular formal languages and, in later contributions, automata. Although it originates from the field of discrete control engineering and hence should be concerned with the typical goals of discrete control tasks as, for example, described by Lunze [70], it struggles with some of the objectives.

SCT appears rather unsuitable to find decent control sequences for chemical batch or manufacturing processes as it lacks liveness properties and, in particular, optimization mechanisms. Even if nonblockingness is considered, a solution is only guaranteed to be realiz*able* but is by no means computed as the CHOICE problem is not covered by the methods of SCT. Where it does perform well, however, is handling *side conditions* that are under all circumstances required to hold as the production goal is achieved. The most prominent example for that is safety requirements, which must never be violated by the plant or the controller. The maximally permissive nature of SCT-synthesized supervisors along with the fact that they are able to controllably guarantee compliance with the specified constraints makes them a perfect match for that task. The presented approach and its tool implementation SynTACS, which both have been developed in the scope of this dissertation project, address precisely this use case. Accordingly, the concerns of implementing a decent controller to realize the actual production goals on the one hand and establishing a supervision layer that definitely ensures compliance with side conditions on the other hand are separated. A violation of safety requirements can thus be excluded even if the controller fails.

SynTACS offers end-to-end tool support. It comprises all necessary algorithms as well as a graphical automaton editor and a code generator to map the results of synthesis to executable code. The latter, being conventionally implemented as an ordinary PLC program, is embedded in a runtime framework, the SyRF. It is generated from the results of supervisor synthesis accompanied by user-provided event definitions. Both framework and controller are programmed into the PLC.

During runtime, SyRF monitors the PLC's inputs as well as the output values of the controller and detects event occurrences based on changes of the signals. In order to make supervision available for a broader class of requirements, the user can choose between the combinations of multiple action and trigger classes for every event independently. It turned out that a flexible notion of what an event is, whether its occurrences should be detected or explicitly executed by the controller, and which effects the acceptance or rejection of such an occurrence has to the system is crucial for a successful application in practice. The same holds for the support of unstable states, i.e., states that cannot safely be kept arbitrarily long without violating or imperiling the specifications. Wonham proposes to utilize timed DES to that end [20, 117]. Unfortunately, these require more complicated models and can significantly affect the computational complexity of the problem. In this thesis, two alternative approaches were presented.

Preemption, the first one, solves the issue by imposing a *contract* between synthesis and the user. It claims that an enforceable action which has been modeled to leave the critical state can reliably preempt any uncontrollable event that is possible before but not after this action. In other words, if such an action can *prevent* an uncontrollable event, it is assumed to be executable fast and timely enough to also *preempt* it. Therefore, a third event class, the enforceable events, has been introduced to both the conceptual approach and the tool. They represent actions that the supervisor can actively trigger without the controller being involved. The synchronization of supervisors on enforceable events during runtime is not yet supported. This was discussed in detail in Section 6.4.2 and should be targeted in future contributions. In the presented case studies, preemption was able to realize all considered specifications with the same permissiveness that a manually implemented safety system could achieve.

The second option to handle unstable states is by defining cyclic events. Technically, these are ordinary, usually controllable, events which are detected recurringly in every PLC cycle as long as their trigger condition holds. That way, they implicitly yield a discretized notion of time similar to Brandin's and Wonham's approach, except that timing does not have to be considered in the models explicitly. When combined with the action class *action-on-denial*, a reaction to leave the critical state can be executed outside the scope of SCT. If the PLC is fast enough to be eligible for the considered control problem, the sampling rate that goes along with the resulting discretization of time is automatically high enough to cope with the situation. Furthermore, SynTACS allows the user to define timers for explicit time measurement. Although originally intended to give the controller additional time to react on a critical event before an emergency action would be enforced, they can be used for numerous other applications.

Despite the fact that SynTACS it is still a prototype, a smooth, easy-to-learn workflow was one of the main targets in order to evaluate how well the methods of SCT can be applied and integrated to the concepts used in factory automation. One of the main issues when providing plain DES models is the necessity of redundant modeling for multiple individual components of the same type that have equivalent behaviors but deal with physically different events. SynTACS allows the user to capture such behaviors once in terms of an automaton template which contains binding events and namespaces (roles). These can be instantiated several times inside the project. The instance maps the binding

events to concrete ones and the namespaces to existing modules in order to represent the individual physical components appropriately.

Moreover, SynTACS allows to restrict the scope of a transition or event prohibition to certain states of other automata by applying conditions to transitions or prohibitions. Since, in many cases, the advantages of modular modeling are consumed by the need of capturing aspects that depend on more than one component, these conditional transitions further reduce the manual modeling effort significantly. The user study further revealed that the participants made less modeling mistakes when they were allowed to define conditional transitions instead of providing equivalent unconditional models. In the current version, all conditions are resolved upon automaton composition.

Symbolic methods represent the state-of-the-art technique for efficiently coping with large state spaces. If realized in future versions of SynTACS, these could also help to handle conditions more effectively, potentially even without the need of explicit composition.

The idea of synchronizing (composing) subsystems not only by their events but also by their states, as suggested by conditional transitions, could be extended by means of alternative modeling languages, e.g., the one introduced by Förster et al. [38], in order to evaluate their suitability for supervisor synthesis.

An algorithm that either produces *interleave insensitive* supervisors right away or at least validates this property would be another desirable feature. Alternatively, a customized event priorization could be installed that allows the user to define the order in which events shall be applied to the supervisor that were detected concurrently, i.e., within the same PLC cycle. Event priorities would also help to select an enforceable event for preemption if more than one is available. Note that in case of multiple enforcements being necessary to avoid several forbidden uncontrollable events at once, the current version of SynTACS would already enforce all of them to achieve safety. In that case, priorities would only be relevant if the order of execution within the cycle influences the results. Note further that amongst multiple options of enforceable events, the algorithm would never pick one which could cause an uncontrollable event to happen that is forbidden within the same supervisor.

Synchronization between the supervisors during runtime would be required, particularly to track events in one supervisor that were enforced by another one. The tool would further benefit from an efficient method to detect prohibitions of enforceable events that were introduced during synthesis of another supervisor. The aspect whether this issue is better resolved during synthesis or dynamically during runtime needs further investigation.

Supervisory control theory, as well as its modification presented in this thesis, provenly yields correct results for correct inputs. The same holds for the maximal permissiveness which also is formally backed. Together with the fact that supervision does not add functionality to the controller, this basically characterizes the approach as a runtime verification method although the formalization is not yet grounded in the runtime verification calculus. The marriage of these two formalisms could be the subject of future research, similar to the existing approaches to unify SCT and reactive synthesis [31].

Compared to a straight controller synthesis, which is not available to a satisfying extent yet, the presented approach of runtime supervision allows a certain degree of redundant design for critical requirements. Although the current shape of the framework does not allow to operate SyRF entirely on technically independent hardware yet, which would be

required for redundancy in the narrower sense, these requirements are at least considered and realized by redundant pieces of software. Indeed that does neither enable fault-tolerant operation nor prevent violations due to hardware failure. The presented approach can, however, exclude the infringement of side conditions due to errors in the controller implementation. In a world of flexible manufacturing facilities, out-of-the-cloud controllers, batch size 1 productions and industry 4.0, controller routines will be dynamically exchanged and will undergo frequent modification in future process and manufacturing systems. SCT-based runtime supervision could contribute the necessary measures to watch over these controllers, keeping the systems safe.

Related Supervised Theses

[Aydin, 2017] Aydin, S. (2017). *Evaluation eines Werkzeugs zur Supervisorsynthese auf einer prozesstechnischen Anlage*. Bachelor's thesis, RWTH Aachen University.

[Gatto, 2016] Gatto, N. (2016). *Erweiterte Codegenerierung für ein Werkzeug zur Synthese von ereignisdiskreten Überwachern*. Bachelor's thesis, RWTH Aachen University.

[Ney, 2014] Ney, O. (2014). *Evaluation von Synthese-Werkzeugen für Steuerungssoftware im Bereich der Automatisierungstechnik Engineering*. Bachelor's thesis, RWTH Aachen University.

[Osetinski, 2017] Osetinski, M. (2017). *Remote Supervisor for Programmable Logic Controllers*. Bachelor's thesis, RWTH Aachen University.

[Timmermanns, 2015] Timmermanns, T. M. (2015). *Code-Generator und Framework für synthetisierte Sicherheitsmechanismen in SPS-Programmen*. Bachelor's thesis, RWTH Aachen University.

Comment on the Neutrality of the Evaluation

In the scope of Ms. Aydin's bachelor thesis [Aydin, 2017], the synthesis and supervision concept presented in Chapter 5 and the tool SynTACS have been evaluated. I suggested an extendable scope for the evaluation which she mainly followed: two case studies to determine the practical applicability of the approach and the tool, and a user study to collect information about its intuitive understanding and usability.

I encouraged Ms. Aydin to face my approaches and the tool with a neutral attitude, to see it as her object of investigation rather than as the embedding scope of her work and hence analyze it critically through the eyes of an external person. She was explcitly told that a bad rating of concepts and the tool would not have any consequences for her assessment or grade. As a consequence, she worked rather independently from me to avoid a significant bias. Particularly when gathering and resuming the user study's results and writing down her own experiences with the tool, I did not interfer with her work. My editing of the corresponding sections of her thesis were confined on grammatical and orthographical corrections but not on the presented findings.

Bibliography

[1] Knut Åkesson, Martin Fabian, and Hugo Flordal. Supremica in a Nutshell – Draft. Technical report, Chalmers, 2007.

[2] Knut Åkesson, Martin Fabian, Hugo Flordal, Arash Vahidi, Knut Akesson, Martin Fabian, Fabian Flordal, Arash Vahidi, Knut Åkesson, Martin Fabian, Hugo Flordal, and Arash Vahidi. Supremica – A Tool for Verification and Synthesis of Discrete Event Supervisors. *Proc. of the 11th Mediterranean Conference on Control and Automation*, 2003.

[3] Knut Åkesson, Hugo Flordal, Martin Fabian, Knut Åkesson, Hugo Flordal, and Martin Fabian. Exploiting modularity for synthesis and verification of supervisors. In *Proc. of the IFAC*, 2002.

[4] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M K Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2013.

[5] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[6] Francisco Sant Anna, Francisco Sant'Anna, Noemi Rodriguez, and Roberto Ierusalimschy. Advanced control reactivity for embedded systems. *Choice*, 9:1, 2013.

[7] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press Cambridge, 2008.

[8] Philippe Balbiani, Valentin Goranko, and Guido Sciavicco. Two-sorted point-interval temporal logics. *Electronic Notes in Theoretical Computer Science*, 278:31–45, 2011.

[9] Silvano Balemi. *Control of discrete event systems*. PhD thesis, ETH Zürich, 1992.

[10] Silvano Balemi, Gérard J. Hoffmann, Paul Gyugyi, H Wong Toi, and Gene F. Franklin. Supervisory control of a rapid thermal multiprocessor. *Automatic Control, IEEE Transactions on*, 1993.

[11] David Basin, Yves Deville, Pierre Flener, and Andreas Hamfelt. Synthesis of Programs in Computational Logic. *Program Development in Computational Logic*, pages 30–65, 2004.

[12] Hans Berger. *Automatisieren mit STEP 7 in AWL und SCL*. Publicis Publishing, Erlangen, 7th edition, 2011.

[13] Sebastian Biallas, Jörg Brauer, and Stefan Kowalewski. Counterexample-guided abstraction refinement for PLCs. *5th International Workshop on Systems Software Verification (SSV 2010), Vancouver, Canada*, page 2, 2010.

[14] Sebastian Biallas, Jörg Brauer, and Stefan Kowalewski. Arcade.PLC: A Verification Platform for Programmable Logic Controllers. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012.

[15] Alessandro Birolini. *Reliability Engineering*. 6th edition, 2010.

[16] Bitkom, VDMA, and ZVEI. Umsetzungsstrategie Industrie 4.0. - Ergebnisbericht der Plattform Industrie 4.0. Technical Report April, Plattform Industrie 4.0, 2015.

[17] Roderick Bloem. Reactive Synthesis, ExCAPE Summer School, Cambridge MA, 2015.

[18] Bertil A. Brandin, Robi Malik, and Petra Dietrich. Incremental system verification and synthesis of minimally restrictive behaviours. In *Proceedings of the 2000 American Control Conference*, pages 0–5, 2000.

[19] Bertil A. Brandin, Robi Malik, and Petra Malik. Incremental verification and synthesis of discrete-event systems guided by counter-examples. *IEEE Transactions on Control Systems Technology*, 12(3):387–401, 2004.

[20] Bertil A. Brandin and W. Murray Wonham. Supervisory Control of Timed Discrete-Event Systems. *Automatic Control, IEEE Transactions on*, 39(2):329–342, 1994.

[21] J. Richard Buchi and Lawrence H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

[22] Matteo Cantarelli and Jean-Marc Roussel. Reactive control system design using the supervisory control theory: evaluation of possibilities and limits. In *Proceedings of the 9th International Workshop on Discrete Event Systems*. IEEE, 2008.

[23] Christos Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Kuwer Academic Publishers, 2nd edition, 2009.

[24] Enke Chen and Stéphane Lafortune. On nonconflicting languages that arise in supervisory control of discrete event systems. *Systems and Control Letters*, 17(2):105–113, 1991.

[25] Alonzo Church. Application of Recursive Arithmetic to the Problem of Circuit Synthesis. *Journal of Symbolic Logic*, 28(4):289–290, 1963.

[26] Alessandro Cimatti and Alberto Griggio. Software Model Checking via IC3. In *Proceesdings of the International Conference on Computer Aided Verification*, 2012.

[27] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[28] Max Hering De Queiroz and José Eduardo Ribeiro Cury. Modular control of composed systems. In *Proceedings of the American Control Conference*, 2000.

[29] Robin Diekmann and Dirk Weidemann. Event Enforcement in the Context of the Supervisory Control Theory. In *Proceedings of the 18th International Conference on Methods and Models in Automation and Robotics*, 2013.

[30] Petra Dietrich, Robi Malik, W Murray Wonham, and Bertil A. Brandin. Implementation considerations in supervisory control. In *Synthesis and control of discrete event systems*, pages 185–201. Springer, 2002.

[31] Rüdiger Ehlers, Stéphane Lafortune, Stavros Tripakis, and Moshe Vardi. Reactive Synthesis vs . Supervisory Control Synthesis: Bridging the Gap. Technical report, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2013.

[32] Ralf Ermlich and Uwe Maier. Speicherprogrammierbare Steuerungen (SPS) und SPS-Systeme. In K. F. Früh and U. Maier, editors, *Handbuch der Prozessautomatisierung*, chapter 4.3, pages 229–249. Oldenbourg Industrieverlag, München, 3rd edition, 2004.

[33] M Fabian and A Hellgren. PLC-based Implementation of Supervisory Control for Discrete Event Systems. In *Proceedings of the 37th IEEE Conference on Decision and Control*, 1998.

[34] Herbert Fittler and Reiner Uhlig. Funktionen der Prozessleitebene: Rezeptfahrweise, Führung von Chargenprozessen. In K. F. Früh and U. Maier, editors, *Handbuch der Prozessautomatisierung*, chapter 3.2, pages 72–112. Oldenbourg Industrieverlag, München, 3rd edition, 2004.

[35] Hugo Flordal, Robi Malik, Martin Fabian, and Knut Åkesson. Compositional synthesis of maximally permissive supervisors using supervision equivalence. *Discrete Event Dynamic Systems*, 2007.

[36] Stefan T. J. Forschelen. *Supervisory control of theme park vehicles*. Master's thesis, Eindhoven University of Technology, 2010.

[37] Stefan T. J. Forschelen, Joanna M. van de Mortel-Fronczak, Rong Su, and Jacobus E. Rooda. Application of supervisory control theory to theme park vehicles. *Discrete Event Dynamic Systems: Theory and Applications*, 22(4):511–540, 2012.

[38] Marc Förster, Marko Auerswald, Phillip Keldenich, and Stefan Kowalewski. Semantic interfaces for automotive software components: exemplary development & validation of a practical specification language. Technical Report ESL-TR-2014-AK-02, RWTH Aachen University, Aachen, 2014.

[39] Georg Frey. Regeln und Steuern. In Hans-Jürgen Gevatter and Ulrich Grünhaupt, editors, *Handbuch der Mess- und Automatisierungstechnik in der Produktion*, chapter 4, pages 39–52. Springer Berlin Heidelberg, Berlin/Heidelberg, 2nd edition, 2006.

[40] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. 2002.

[41] Florian Göbe, Selin Aydin, and Stefan Kowalewski. Applicability of Supervisory Control Theory for the Supervision of PLC Programs. In *Proceedings of the 22nd IEEE International Conference on Emerging Technology & Factory Automation*, 2017.

[42] Florian Göbe, Oliver Ney, and Stefan Kowalewski. Reusability and Modularity of Safety Specifications for Supervisory Control. In *Proceedings of the 21st IEEE International Conference on Emerging Technology and Factory Automation*, 2016.

[43] Florian Göbe, Thomas Timmermanns, Oliver Ney, and Stefan Kowalewski. Synthesis Tool for Automation Controller Supervision. In *Proceedings of the 13th International Workshop on Discrete Event Systems*, pages 424–431, 2016.

[44] C.H. Golaszewski and P.J. Ramadge. Control of Discrete Event Processes with Forced Events. In *Proceedings of the 26th Conference on Decision and Control*, 1987.

[45] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *ACM SIGPLAN Notices*, 2011.

[46] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *ACM SIGPLAN Notices*, 2008.

[47] Hans-Michael Hanisch and Stefan Kowalewski. Algebraic synthesis and verification of discrete supervisory controllers for forbidden path specifications. In *Proceedings of the Fourth International Conference on Computer Integrated Manufacturing and Automation Technology*, 1994.

[48] Anders Hellgren, Bengt Lennartson, and Martin Fabian. Modelling and PLC-based implementation of modular supervisory control. *Sixth International Workshop on Discrete Event Systems, 2002. Proceedings.*, (FEBRUARY 2002):371–376, 2002.

[49] R.C. Hill and D.M. Tilbury. Incremental hierarchical construction of modular supervisors for discrete-event systems. *International Journal of Control*, 2008.

[50] Gérard J. Hoffmann and H. Wong-Toi. Symbolic Synthesis of Supervisory Controllers. In *Proceedings of the American Control Conference*, number L, pages 2789–2793, 1992.

[51] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Automata theory, languages, and computation. *International Edition*, 24, 2006.

[52] Jing Huang and Ratnesh Kumar. An optimal directed control framework for discrete event systems. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 2007.

[53] Jing Huang and Ratnesh Kumar. Optimal nonblocking directed control of discrete event systems. *Automatic Control, IEEE Transactions on*, 2008.

[54] International Electrotechnical Commission. International Standard IEC 61131-1:2003: Programmable Controllers – Part 1: General Information.

[55] International Electrotechnical Commission. International Standard IEC 61131-3:2013: Programmable Controllers – Part 3: Programming Languages.

[56] International Electrotechnical Commission. International Standard IEC 61508:1998: Functional Safety of Electrical, Electronic and Programmable Electronic Safety-Related Systems.

[57] International Electrotechnical Commission. International Standard IEC 61511:2005 Functional safety - Safety instrumented systems for the process industry sector.

[58] International Electrotechnical Commission. International Standard IEC 61512-2:2002: Batch Control - Part 2: Data structures and guidelines for languages.

[59] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. The MIT Press, Cambridge, MA, USA, revised edition, 2012.

[60] Karl-Heinz John and Michael Tiegelkamp. *SPS-Programmierung mit IEC 61131-3*. Springer-Verlag, Berlin Heidelberg, 4th edition, 2009.

[61] Stefan Kowalewski and H-M Hanisch. Permissive control of boolean condition/event systems: synthesis and limits. In *Proceedings of the International Symposium on Intelligent Control*, pages 118–123, Columbus, OH, USA, 1994. IEEE.

[62] Tina Krausser, Lars Evertz, and Ulrich Epple. A Hands-On Laboratory on Industrial Hardware, Process Control and Advanced Automation. *IFAC Proceedings Volumes*, 2012.

[63] Bruce H. Krogh and Stefan Kowalewski. State Feedback of Condition/Event Control Systems. *Mathematical and Computer Modelling*, 23(11):161–173, 1996.

[64] Ratnesh Kumar and Vijay K. Garg. Optimal supervisory control of discrete event dynamical systems. *SIAM Journal on Control and Optimization*, 33(2):419–439, 1995.

[65] Ryan James Leduc. *PLC implementation of a DES supervisor for a manufacturing testbed: An implementation perspective*. PhD thesis, University of Toronto, 1996.

[66] Nancy G. Leveson. Safeware: System Safety and Computers. *Medical Physics*, 23(10):1821, 1995.

[67] R. W. Lewis. *Programming industrial control systems using {IEC} 1131-3 – Revised Edition*. The Institution of Electrical Engineers, London, UK, 1998.

[68] F. Lin, A.F. Vaz, and W.M. Wonham. Supervisor specification and synthesis for discrete event systems. *International Journal of Control*, 48(1):321–332, 1988.

[69] Jørn Lind-Nielsen, Henrik Reif Andersen, Henrik Hulgaard, Gerd Behrmann, Kåre Kristoffersen, and Kim G Larsen. Verification of large state/event systems using compositionality and dependency analysis. *Formal Methods in System Design*, 18(1):5–23, 2001.

[70] Jan Lunze. *Automatisierungstechnik*. Oldenbourg Wissenschaftsverlag GmbH, München, 2003.

[71] Jan Lunze. *Regelungstechnik 1 - Systemtheorietische Grundlagen, Analyse und Entwurf einschleifiger Regelungen*. Springer, 9 edition, 2012.

[72] Chuan Ma and W. Murray Wonham. *Nonblocking Supervisory Control of State Tree Structures*. Lecture Notes in Control and Information Sciences, Springer Berlin Heidelberg New York, Heidelberg, Germany, 2005.

[73] Chuan Ma and W. Murray Wonham. Nonblocking supervisory control of state tree structures. *Automatic Control, IEEE Transactions on*, 2006.

[74] Uwe Maier. Situation der Prozessautomatisierung: Marktsituation und Markttrends. In K. F. Früh and U. Maier, editors, *Handbuch der Prozessautomatisierung*, chapter 1.1, pages 2–6. Oldenbourg Industrieverlag, München, 3rd edition, 2004.

[75] Robi Malik and Hugo Flordal. Yet another approach to compositional synthesis of discrete event systems. *Proceedings - 9th International Workshop on Discrete Event Systems, WODES' 08*, pages 16–21, 2008.

[76] Zohar Manna, Richard Waldinger, Zohar Manna, and Richard Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, 1992.

[77] Jasen Markovski, Koen G.M. Jacobs, Dirk. A. Van Beek, Lou J.A.M. Somers, and Jacobus E. Rooda. Coordination of resources using generalized state-based requirements. In *Proceedings of WODES*, volume 2010, pages 300–305, 2010.

[78] S Miremadi, B Lennartson, and K Akesson. A BDD-Based Approach for Modeling Plant and Supervisor by Extended Finite Automata. *Control Systems Technology, IEEE Transactions on*, 2012.

[79] Jürgen Mittelstraß. *Enzyklopädie Philosophie und Wissenschaftstheorie, Bd. 5*. J.B. Metzler, Springer-Verlag, 2013.

[80] Sergio Montenegro. Sichere und fehlertolerante Steuerungen. *Entwicklung sicherheitsrelevanter Systeme. Hanser. M{ü}nchen, Wien*, 1999.

[81] Thomas Moor, Klaus Schmidt, and Sebastian Perk. libFAUDES: An open source C++ library for discrete event systems. In *Proceedings of the 9th International Workshop on Discrete Event Systems*, 2008.

[82] Thomas Moor, Klaus Schmidt, and Sebastian Perk. Applied supervisory control for a flexible manufacturing system. *IFAC Proceedings Volumes*, 43(12):253–258, 2010.

[83] Bernd Opitz. *Methods of supervisory control: A software implementation*. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2005.

[84] Lucas P. Pinheiro, Yuri K. Lopes, André B. Leal, and Roberto S.U. Rosso Junior. Nadzoru: A Software Tool for Supervisory Control of Discrete Event Systems. *IFAC-PapersOnLine*, 48(7):182–187, 2015.

[85] PLCopen Technical Committee 5 – Safety Software. Technical Specification – Concepts and Function Blocks, Version 1.0, 2006.

[86] T. J. Prati, J. M. Farines, and M. H. De Queiroz. Automatic test of safety specifications for PLC programs in the oil and gas industry. *IFAC-PapersOnLine*, 48(6):27–32, 2015.

[87] Peter J. G. Ramadge and W. Murray Wonham. Supervisory Control of a Class of Discrete Event Processes. *SIAM Journal on Control and Optimization*, 63(1):206–230, 1987.

[88] Peter J G Ramadge and W Murray Wonham. The Control of Discrete Event Systems. In *Proceedings of the IEEE*, 1989.

[89] Peter J.G. Ramadge and W. Murray Wonham. Supervisory control of a class of discrete event processes. Technical Report 1, Systems Control Group, Dept. of Electrical Engineering, Univ. of Toronto, Toronto, Ont., Canada, 1984.

[90] Christianne Reiser, Antonio E C Da Cunha, and José Eduardo Ribeiro Cury. The environment grail for supervisory control of discrete event systems. In *Discrete Event Systems, 2006 8th International Workshop on*, pages 390–391, 2006.

[91] L. Ricker, S. Lafortune, and S. Gene. Desuma: A tool integrating giddes and umdes. In *Software tools, 8th international workshop on discrete-event systems*, 2006.

[92] B. Riera, A. Philippot, R. Coupat, F. Gellot, and D. Annebicque. A non-intrusive method to make safe existing PLC program. *IFAC-PapersOnLine*, 28(21):320–325, 2015.

[93] Francisco Sant'Anna, Roberto Ierusalimschy, Noemi Rodriguez, Silvana Rossetto, and Adriano Branco. The Design and Implementation of the Synchronous Language Céu. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(4):98:1–98:25, 2017.

[94] Klaus Schmidt. *Hierarchical Control of Decentralized Discrete Event Systems*. Dissertation, Universität Erlangen-Nürnberg, 2005.

[95] Klaus Schmidt, Thomas Moor, and Sebastian Perk. Nonblocking hierarchical control of decentralized discrete event systems. *Automatic Control, IEEE Transactions on*, 2008.

[96] Ann-Kathrin Schmuck, Sven Schneider, Jörg Raisch, and Uwe Nestmann. Extending supervisory controller synthesis to deterministic pushdown automata—enforcing controllability least restrictively. In *Proceedings of the 12th International Workshop on Discrete Event Systems*, Cachan, France, 2014.

[97] M. Schuh, M. Zgorzelski, and J. Lunze. Experimental evaluation of an active fault–tolerant control method. *Control Engineering Practice*, 43:1–11, 2015.

[98] Melanie Schuh and Jan Lunze. Feedback control of nondeterministic Input/Output automata. In *Decision and Control (CDC), 2014 IEEE 53rd Annual Conference on*, pages 6737–6743. IEEE, 2014.

[99] Melanie Schuh and Jan Lunze. Fault-tolerant control of deterministic I/O automata with ambiguous diagnostic result. In *13th International Workshop on Discrete Event Systems*. IEEE, 2016.

[100] Raja Sengupta and Stéphane Lafortune. an Optimal Control Theory for Discrete Event Systems. *SIAM Journal on Control and Optimization*, 36(2):488–541, 1998.

[101] Mohammad Reza Shoaei, Laura Kovács, and Bengt Lennartson. Supervisory Control of Discrete-Event Systems via IC3. In *Hardware and Software: Verification and Testing*, pages 252–266. Springer, 2014.

[102] David Smith and Kenneth Simpson. *Functional Safety: A Straightforward Guide to Applying IEC 61508 and Related Standards*. Routledge, 2nd edition, 2004.

[103] Fabio Somenzi and Aaron R. Bradley. IC3: Where monolithic and incremental meet. *Formal Methods in Computer-Aided Design*, pages 3–8, 2011.

[104] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001.

[105] R. S. Sreenivas and B. H. Krogh. On condition/event systems with discrete state realizations. *Discrete Event Dynamic Systems: Theory and Applications*, 1(2):209–236, 1991.

[106] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In *ACM Sigplan Notices*, 2009.

[107] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *ACM Sigplan Notices*, 2010.

[108] Stefan Stattelmann, Sebastian Biallas, Bastian Schlich, and Stefan Kowalewski. Applying Static Code Analysis on Industrial Controller Code. In *19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2014.

[109] Rong Su, Jan H. Van Schuppen, and Jacobus E. Rooda. Synthesize nonblocking distributed supervisors with coordinators. In *Proceedings of the 17th Mediterranean Conference on Control and Automation*, 2009.

[110] Rong Su, Jan H. Van Schuppen, and Jacobus E. Rooda. Aggregative synthesis of distributed supervisors based on automaton abstraction. *IEEE Transactions on Automatic Control*, 55(7):1627–1640, 2010.

[111] Rolf J M Theunissen, M Petreczky, Ramon R H Schiffelers, Dirk A van Beek, and Jacobus E Rooda. Application of Supervisory Control Synthesis to a Patient Support Table of a Magnetic Resonance Imaging Scanner. *IEEE Trans. Automation Science and Engineering*, 11(1):20–32, 2014.

[112] Wolfgang Thomas. Church 's Problem and a Tour through Automata Theory. In *Pillars of computer science*, pages 635–655. Springer-Verlag, Heidelberg, Germany, 2008.

[113] Arash Vahidi, Martin Fabian, and Bengt Lennartson. Efficient supervisory synthesis of large systems. *Control Engineering Practice*, 2006.

[114] VDI/VDE-GMA. Industrie 4.0 Service Architecture. Technical Report November, 2016.

[115] Agnelo D. Vieira, José E.R. Cury, and Max H. de Queiroz. A Model for PLC Implementation of Supervisory Control of Discrete Event Systems. In *Proceedings of IEEE Conference on Emerging Technologies and Foctory Automation*, 2006.

[116] Alexey Voronov and Knut Åkesson. Supervisory control using satisfiability solvers. *Proceedings - 9th International Workshop on Discrete Event Systems, WODES' 08*, pages 81–86, 2008.

[117] W. M. Wonham. *Supervisory Control of Discrete-Event Systems*. Edward S. Rogers Sr. Dept. of Electrical & Computer Engineering, University of Toronto, 2014.

[118] W. M. Wonham and P. J. G. Ramadge. On the supremal controllable sublanguage of a given language. In *Proceedings of the 23rd Conference on Decision and Control*, 1984.

[119] W. M. Wonham and P. J. G. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal on Control and Optimization*, 25(3):637–659, 1987.

[120] W. M. Wonham and P. J. G. Ramadge. Modular supervisory control of discrete-event systems. *Mathematics of Control, Signals and Systems*, 1(1):13–30, 1988.

[121] Ming-Li Yeh and Chuei-Tin Chang. An automata-based approach to synthesize untimed operating procedures in batch chemical processes. *Korean Journal of Chemical Engineering*, 29(5):583–594, 2012.

[122] Ming-Li Yeh and Chuei-Tin Chang. An automata based method for online synthesis of emergency response procedures in batch processes. *Computers & Chemical Engineering*, 38:151–170, 2012.

[123] Zhonghua Zhang and W. Murray Wonham. STCT: An Efficient Algorithm for Supervisory Control Design. In *Synthesis and Control of Discrete Event Systems*. Springer US, 2002.

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

http://aib.informatik.rwth-aachen.de/

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: biblio@informatik.rwth-aachen.de

2016-01 * Fachgruppe Informatik: Annual Report 2016

2016-02 Ibtissem Ben Makhlouf: Comparative Evaluation and Improvement of Computational Approaches to Reachability Analysis of Linear Hybrid Systems

2016-03 Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl: Lower Runtime Bounds for Integer Programs

2016-04 Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder: Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution

2016-05 Mathias Pelka, Grigori Goronzy, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 2nd KuVS Expert Talk on Localization

2016-06 Martin Henze, René Hummen, Roman Matzutt, Klaus Wehrle: The SensorCloud Protocol: Securely Outsourcing Sensor Data to the Cloud

2016-07 Sebastian Biallas : Verification of Programmable Logic Controller Code using Model Checking and Static Analysis

2016-08 Klaus Leppkes, Johannes Lotz, and Uwe Naumann: Derivative Code by Overloading in C++ (dco/c++): Introduction and Summary of Features

2016-09 Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann: Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic

2016-10 Stefan Wüller, Ulrike Meyer, and Susanne Wetzel: Towards Privacy-Preserving Multi-Party Bartering

2017-01 * Fachgruppe Informatik: Annual Report 2017

2017-02 Florian Frohn and Jürgen Giesl: Analyzing Runtime Complexity via Innermost Runtime Complexity

2017-04 Florian Frohn and Jürgen Giesl: Complexity Analysis for Java with AProVE

2017-05 Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl: Complexity Analysis for Term Rewriting by Integer Transition Systems

2017-06 Oliver Kautz, Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe: CD2Alloy: A Translation of Class Diagrams to Alloy

2017-07 Klaus Leppkes, Johannes Lotz, Uwe Naumann, and Jacques du Toit: Meta Adjoint Programming in C++

2017-08 Thomas Gerlitz: Incremental Integration and Static Analysis of Model-Based Automotive Software Artifacts

2017-09 Muhammad Hamad Alizai, Jan Beutel, Jó Ágila Bitsch, Olaf Landsiedel, Luca Mottola, Przemyslaw Pawelczak, Klaus Wehrle, and Kasim Sinan Yildirim: Proc. IDEA League Doctoral School on Transiently Powered Computing

2018-01 * Fachgruppe Informatik: Annual Report 2018

2018-02 Jens Deussen, Viktor Mosenkis, and Uwe Naumann: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen

2018-03 Igor Kalkov: A Real-time Capable, Open-Source-based Platform for Off-the-Shelf Embedded Devices

2018-04 Andreas Ganser: Operation-Based Model Recommenders

2018-05 Matthias Terber: Real-World Deployment and Evaluation of Synchronous Programming in Reactive Embedded Systems

2018-06 Christian Hensel: The Probabilistic Model Checker Storm - Symbolic Methods for Probabilistic Model Checking

2019-01 * Fachgruppe Informatik: Annual Report 2019

2019-02 Tim Felix Lange: IC3 Software Model Checking

2019-03 Sebastian Patrick Grobosch: Formale Methoden für die Entwicklung von eingebetteter Software in kleinen und mittleren Unternehmen

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.