Otto–von–Guericke–Universität Magdeburg

Studienarbeit

Supervisory Control of a Manufacturing Cell: Modeling and Implementation

von

Steffi Klinge
(***** in Burg)

14. September 2007

Eingereicht an die: Otto–von–Guericke–Universität Magdeburg
Fakultät für Elektrotechnik und Informationstechnik
Prüfungsamt
Universitätsplatz 2
39016 Magdeburg,
Deutschland

Prüfer: Prof. Dietrich Flockerzi

Max Planck Institut
Dynamik komplexer technischer Systeme
Sandtorstraße 1
39106 Magdeburg
Deutschland

Betreuer: Prof. José E. Ribeiro Cury
Prof. Max Hering de Queiroz

Departamento de Automação e Sistemas
Centro Tecnológico
Universidade Federal de Santa Catarina
Florianópolis - SC
CEP 88040-900
Brasil

Table of contents

List of figures **vi**

List of tables **vii**

Outlines and objectives **ix**

Preface **xi**
Acknowledgments xi
Declaration of originality xii

Introduction **1**

1 Theory and tools **3**
1.1 Introduction 3
1.2 Discrete Event Systems 4
1.2.1 Languages 4
1.2.2 Automata 5
1.3 Supervisory Control Theory 9
1.3.1 Construction of an optimal, nonblocking supervisor . . . 9
1.3.2 Reduction of supervisors 10
1.4 Tools 11
1.4.1 TCT 12
1.4.2 GRAIL 12
1.4.3 IDES 12

2 Models and specifications **15**
2.1 Introduction 15
2.2 Testbed 16
2.2.1 Verbal description 16
2.2.2 Sensors and actuators 17

2.2.3 Desired proceeding 18
2.3 Similarities 19
2.3.1 Modeling the plant 19
2.3.2 Modeling the specifications 20
2.4 Model 1 21
2.4.1 Modeling the robot 21
2.4.2 Modeling the specifications without rework 21
2.4.3 Modeling the specifications with rework 1 24
2.4.4 Modeling the specifications with rework 2 28
2.5 Model 2 31
2.5.1 Modeling the robot 31
2.5.2 Modeling the specifications without rework 32
2.5.3 Modeling the specifications with rework 1 33
2.5.4 Modeling the specifications with rework 2 36
2.6 Model 3 and 4 37
2.6.1 Modeling the robot 37
2.6.2 Modeling the specifications without rework 38
2.7 Comparison 39

3 Supervisors 43
3.1 Introduction 43
3.2 Monolithic supervisors 44
3.3 Modular supervisors 44
3.3.1 Results 45
3.3.2 Conflicts 52
3.4 Comparison 57

4 Implementation 59
4.1 Introduction 59
4.2 General information 59
4.2.1 Hierarchical structure 60
4.2.2 Initializing the system 61
4.2.3 Implementation of transitions 61
4.2.4 Programming languages 62
4.3 Manufacturing cell 64
4.3.1 Initialization and reinitialization 64
4.3.2 Siemens PLC 64
4.3.3 Altus PLC 66

Conclusions 69

A IDES2ST 71
A.1 Introduction 71
A.2 IDES2ST.java 71
A.3 ControlSystem.java 74
A.4 Automaton.java 78
A.5 State.java 87
A.6 Event.java 89
A.7 Transition.java 91

Bibliography 95

List of figures

1.1 A simple automaton 6
1.2 A not accessible and not coaccessible automaton 7
1.3 A simple workstation 8

2.1 Photo of the testbet 16
2.2 Scheme of the testbet 17
2.3 Photo of pieces of work 17
2.4 Model: drilling and welding station 19
2.5 Model: table 19
2.6 Model: test 19
2.7 Model: port 1 and port 2 20
2.8 Specification: mutual exclusion, E_{11}, E_{12} and E_{13} 20
2.9 Specification: avoid moving empty table, E_2 21
2.10 Model 1: robot 22
2.11 Specification model 1 without rework: mutual exclusion, E_{14} . . . 22
2.12 Specification model 1 without rework: operating sequence, E_{31} . . 22
2.13 Specification model 1 without rework: operating sequence, E_{32} . . 23
2.14 Specification model 1 without rework: operating sequence, E_{33} . . 23
2.15 Specification model 1 without rework: operating sequence, E_{34} . . 23
2.16 Specification model 1 without rework: corresponding outcome, E_4 24
2.17 Specification model 1 without rework: get new pieces, E_5 24
2.18 Specification model 1 rework 1: alternative routine, E_6 25
2.19 Specification model 1 rework 1: mutual ecxlusion, E_{14} 25
2.20 Specification model 1 rework 1: operating sequence, E_{31} 26
2.21 Specification model 1 rework 1: operating sequence, E_{32} 26
2.22 Specification model 1 rework 1: operating sequence, E_{33} 27
2.23 Specification model 1 rework 1: operating sequence, E_{34} 27
2.24 Specification model 1 rework 1: corresponding outcome, E_4 28
2.25 Specification model 1 rework 1: get new pieces, E_5 28

2.26 Specification model 1 rework 2: operating sequence, E_{33} 29
2.27 Specification model 1 rework 2: operating sequence, E_{34} 30
2.28 Specification model 1 rework 2: corresponding outcome, E_{41} 30
2.29 Specification model 1 rework 2: only one piece, E_6 31
2.30 Model 2: robot 32
2.31 Specification model 2 without rework: mutual exclusion, E_{14} . . . 32
2.32 Specification model 2 without rework: operating sequence, E_{31} . . 33
2.33 Specification model 2 without rework: operating sequence, E_{34} . . 33
2.34 Specification model 2 without rework: corresponding outcome, E_4 34
2.35 Specification model 2 without rework: get new pieces, E_5 34
2.36 Specification model 2 rework 1: alternative routine, E_6 34
2.37 Specification model 2 rework 1: mutual ecxlusion, E_{14} 35
2.38 Specification model 2 rework 1: operating sequence, E_{31} 35
2.39 Specification model 2 rework 1: operating sequence, E_{34} 36
2.40 Specification model 2 rework 1: corresponding outcome, E_4 36
2.41 Model 3: robot 37
2.42 Model 4: arm 37
2.43 Model 4: grabber 38
2.44 Specification model 3 without rework: Robot control, E_6 38
2.45 Specification model 3 without rework: Mutual exclusion, E_{14} . . . 39

3.1 Reduced supervisor model 1 without rework: mutual exclusion, R_{11} 45
3.2 Reduced supervisor model 1 without rework: avoid empty table, R_2 46
3.3 Reduced supervisor model 1 without rework: operating sequ., R_{31} 46
3.4 Reduced supervisor model 1 without rework: corres. outcome, R_4 46
3.5 Reduced supervisor model 1 without rework: get new pieces, R_5 . 46
3.6 Reduced supervisor model 1 rework 1: alternative routine, R_6 . . . 47
3.7 Reduced supervisor model 1 rework 1: operating sequence, R_{31} . . 48
3.8 Reduced supervisor model 1 rework 1: get new pieces, R_5 48
3.9 Reduced supervisor model 1 rework 2: operating sequence, R_{33} . . 49
3.10 Reduced supervisor model 1 rework 2: corres. outcome, R_{41} 50
3.11 Reduced supervisor model 1 rework 2: only one piece, R_6 50
3.12 Reduced supervisor model 2 without rework: mutual exclusion, R_{14} 51
3.13 Reduced supervisor model 2 without rework: corres. outcome, R_4 51
3.14 Supervisor model 3: robot control, S_6 53
3.15 Reduced supervisor model 3: robot control, R_6 53
3.16 Scheme of supervisors and parts of the plant 54
3.17 Reduced coordinator for model 2 without rework: $C_{red,1}$ 56
3.18 Reduced coordinator for model 2 without rework: $C_{red,2}$ 56

4.1 Hierarchical structure 60
4.2 Ladder diagram: an example 63

List of tables

2.1 Inputs 18
2.2 Outputs 18
2.3 Physical possibilities of the models 39
2.4 Size of models without rework 40
2.5 Size of models with rework 40
2.6 Size of specifications without rework 41
2.7 Size of specifications with rework 42

3.1 Size of monolithic supervisors 44
3.2 Local and reduced supervisors for model 1 without rework 45
3.3 Local and reduced supervisors for model 1 with rework 1 47
3.4 Local and reduced supervisors for model 1 with rework 2 49
3.5 Local and reduced supervisors for model 2 without rework 50
3.6 Local and reduced supervisors for model 2 with rework 1 52
3.7 Local and reduced supervisors for model 2 with rework 2 52
3.8 Local and reduced supervisors for model 3 without rework 53
3.9 Coordinators for model 2 and 3 56
3.10 Reduced supervisors for model 1, 2 and 3 without rework 57
3.11 Reduced supervisors for model 1 and 2 with rework 58

Outlines and objectives

The goal of this thesis is the comparison of different approaches to supervisor design in conjunction with a concrete practical application to a small manufacturing cell.

In the course of this thesis, the student should become familiar with the basics of Automata and Supervisory Control Theory since the understanding of the possibilities and the limits of automata and supervisors is fundamental for a successful realisation of the project.

The practical part of the work deals with a mechanical model of a manufacturing cell. The student should be able to create and compare different models of the plant – restrictive small models as well as flexible large models.

In the next step, different types of supervisors — for instance of monolithic, modular or hierarchical nature — should be implemented using software tools such as TCT, GRAIL or IDES. The complexity of the model together with its potential blocking problems will influence the choice of the software tool used.

The third part of the work should show tests of the implemented supervisors on the plant. It should be possible to control the plant with different models and supervisors for different requirements and specifications.

Finally the student is asked to compare the various models and supervisors, to indicate advantages and disadvantages of the different approaches and to highlight their capabilities and limits.

Preface

Acknowledgments

First of all I would like to thank Prof. Jörg Raisch, Technische Universität, Berlin, Germany, for making the connection to Brazil.

Furthermore I would like to express my deep gratitude to Prof. José E. R. Cury and Prof. Max H. de Queiroz, Universidade Federal de Santa Catarina, Florianópolis, Brazil, for their support during my stay, their motivation and their suggestions to improve my work.

Especially I would like to thank Florian Knorn for his ceaseless constructive critic, his motivation and his endless patent while answering my questions.

Thanks to Lenko Grigorov, Queen's University, Kingston, Canada, for his help, his suggestions and the realization of a lot of changes in *IDES*.

Thanks to Harald Bauer, Friedrich-Alexander-Universität Erlangen-Nürnberg, for his patience while explaining Java–routines and his help while writing *IDES2ST*, Luis Gustavo Marquez and Guilherme Siviero Lise, Universidade Federal de Santa Catarina, Florianópolis, Brazil, for programming the subroutines, Christiano Casanova, Universidade Federal de Santa Catarina, Florianópolis, Brazil, for his suggestions and the *Friedrich–Naumann–Stiftung* for the founding.

Declaration of originality

I hereby declare that this thesis and the work reported herein was composed and originated entirely by myself. Information derived from the published and unpublished work of others is acknowledged in the text and a list of references is given in the bibliography.

____

Steffi Klinge

Introduction

Whenever the choice to use a discrete approach to solve a problem, especially a technical problem, there are two basic options to model and control a system. A very common and also often used way to model systems are Petri Nets, first developed by C.A. Petri in 1962, [20]. This approach is an efficient possibility to handle most of the problems occurring in technical issues. In many cases nevertheless Petri Nets are not necessary. Often real problems allow to use the most basic class of DES models (Discrete Event System), which are automata. They are easy to understand and to analyse. Although there exist situations where automata would grow huge or even infinitely, many problems can be solved with automata in an efficient way.

In order to control automata P.J. Ramadge and W.M. Wonham developed the Supervisory Control Theory in the 1980's,[21]. The main idea is to formulate rules which should not be violated by the plant. These specifications are modeled as one or more automata. After building the synchronous product of the plant and the specification a supervisor can be created automaticly.

Finally a supervisor is an automaton as well, which is tracking the actions of the plant and disabling undesired controllable events in order to avoid violation of specifications or blocking. Blocking would mean that one or more sequences of events lead to an situation from which it is not possible to complete a desired task, which whould mean to reach a marked state, without violating a specification.

Even though this sounds easy, modeling the real system and compose the specifications in order to get an optimal solution is not trivial. As real world often handles with continuous variables as e.g. time, velocity or distance it depends completely on the developer how to define an event in the model. To define quite a lot of different events describing in a very specific way every physical event of the plant with the objective of receive a flexible but sometimes huge plant is also possible as combine some of these events in order to get a smaller but more restrictive model.

After obtaining the model of the free plant formulating the specifications is the next task. Security reasons can call for specifications as well as required work flows. A good knowledge both of the real technical system and the abstract model is absolutely needed to express specifications as automata. Yet to bear all undesired but possible actions of the plant is normally very complex. Also it is almost always possible to find more than one solution to assure the compliance of requirements. So obtained supervisors can vary in size and restrictions.

For the purpose of achieving only one supervisor, that makes sure that every specification respected, building the synchronous product of all automata

representing different demands would be the next step. This procedure can cause a huge, inscrutable supervisor, which cannot be analyzed or debugged.

This problem can be a reason to implement a modular solution. To do so from every automaton representing a specification a local supervisor is computed. On the one hand changes in the specifications can be realized easily and the resulting local supervisors are smaller and easier to understand and to debug, on the other hand a local approach can lead to a conflicting problem. As every modular supervisor only attend to one specification the interaction between them can result a situation where the completion of a task is not possible or even all events are disabled by the local supervisores.

To solve this problem it may help to reduce the number of local supervisors by building the synchronous product of specifications causing a blocking situation. Also it could be serviceable to add a further specification which avoids the blocking situation. Both of these ways assume that the developer already has an idea where exactly the problem occurs. Yet to detect the reason for the blocking is not trivial and requires an excellent knowledge of the plant and the specifications.

CHAPTER 1

Theory and tools

This chapter gives a short overview about what Discrete Event Systems are, how to model them by automata, what are supervisors and how to compute them, as well as an introduction to the tools used in this project to model and compute supervisors.

1.1 Introduction

In industry often high numbers of different processes have to be coordinated. In some applications only the start and the end of procedures or moves, the quantity of items or sensors with a defined, bounded and small number of output signals are important. In these cases it is always good to think about modeling the system by an automaton.

As long as systems only work with one piece at a time a simple sequence control may be sufficient to control the plant. Once the handling of more pieces is desired or a more robust solution is required the construction of a supervisor may help to solve the problem.

This chapter will try to explain the most basic definitions used in Discrete Event Theory, specially to handle with automata and Supervisory Control Theory. As a lot of very good literature about these theories is available, we will only try to give a short overview. All definitions and theorems are taken from [8], [28], [23], [26], [25], [18] and [24].

There are quite a lot of divers software tools available. They differ in terms of visualization, appliance, the used algorithms, the capability of handling with hugh automata and their velocity. In this project three tools have been used: TCT, developed by the Systems Control Group of the University of Toronto, Grail for Supervisory Control, developed by the Department of Automation and Systems (DAS) of the Federal University of Santa Catarina (UFSC), and IDES, developed by the Department of Electrical and Computer Engineering of the Queen's University.

1.2 Discrete Event Systems

1.2.1 Languages

Before we start talking about Discrete Event Systems it is recommendable to clarify basic definitions. In literature a lot of definitions for the terms *system* and *event* can be found.

We will understand a *system* as *a combination of components that act together to perform a function not possible with any of the individual parts* as mentioned in [19].

Also we want an event to occur instantaneously, causing transitions from one state value to another, like it is expressed in [8]. The definitions in this section are based on [8] and [28].

All events of a system build the discrete *event set* or *alphabet* Σ.

Definition 1.2.1 (Language)**.** A *language* L defined over an alphabet Σ is a set of finite-legth strings formed from events in Σ.

The countable infinite set of all finite strings of elements of Σ and the empty string ϵ is called the Kleene – closure Σ^*.

If there is a string $s = tuv$, so

- t is called a *prefix* of s,
- u is called a *substring* of s and
- v is called a *suffix* of s.

Example 1.2.2. For $\Sigma = \{a, b, c\}$ some possible languages would be

- $L_1 = \{\epsilon, a, bb, abb\}$,
- $L_2 = \{\epsilon, c, cc\}$ or
- $L_3 = \Sigma^* = \{\epsilon, a, b, c, aa, ab, ac, ba, bb, \dots\}$.

It is easy to see, that every language over Σ is a subset of Σ^*.

Definition 1.2.3 (Concatenation)**.** Let L_1 and L_2 be two languages with $L_1, L_2 \subseteq \Sigma^*$, then L_1L_2 is the concatenation of them with

$$L_1L_2 := \{s \in \Sigma^* : (s = s_1s_2) \ , (s_1 \in L_1) \text{ and } (s_2 \in L_2)\}.$$

Example 1.2.4. Let L_4 be the concatenation of L_1 and L_2 of Example 1.2.2.

$$L_4 = L_1L_2 = \{\epsilon, a, bb, abb, c, ac, bbc, bbcc, abbc, cc, acc, abbcc\}$$

Definition 1.2.5 (Prefix – Closure)**.** Let $L \subseteq \Sigma^*$, then

$$\bar{L} := \{s \in \Sigma^* : \exists t \in \Sigma^*(st \in L)\}$$

A language is called *prefix – closed*, if it contains every prefix of every string of the language, $L = \bar{L}$.

Example 1.2.6. Looking at the languages defined in Example 1.2.2:

- $\bar{L}_1 = \{\epsilon, a, b, ab, bb, abb\}$
- L_2 is prefix – closed.

Definition 1.2.7 (Kleene – Closure)**.** Let $L \subseteq \Sigma^*$, then

$$L^* := \{\epsilon\} \cup L \cup LL \cup LLL \cup \dots$$

Example 1.2.8. The Kleene – closure of L_1 of Example 1.2.2 is

$$L_1^* = \{\epsilon, a, bb, abb, aa, bba, abba, bbbb, abbbb, aabb, bbabb, abbabb, aaa, \dots\}$$

Discrete Event Systems can be described by languages. If a physical system generates a set of events Σ, the behaviour of the plant can be described by L,$L_m \subseteq \Sigma^*$. The language L is formed by all strings representing the possible physical events of the plant. $L_m \subseteq L$ includes all strings of L, which correspond to completed tasks.

To express languages in a formal way *regular expressions* are a good choice. It is a defined as follows:

- $\emptyset$ is a regular expression representing the empty language.
- ϵ is a regular expression meaning the language $\{\epsilon\}$.
- δ is a regular expression denoting the language $\{\delta\}$ $\forall \delta \in \Sigma$.
- If s and r are regular expressions thus rs, r^*, s^* and $(r + s)$ (what means r or s) are as well.
- Every regular expression can be obtained by applying the rules above a finite number of times.

1.2.2 Automata

Under an automaton we will understand a form to describe two languages representing the behavior of a system and thus to present a Discrete Event System. It can be visualized by a directed graph, which consists of nodes representing states of the system and directed arcs between them representing transitions.

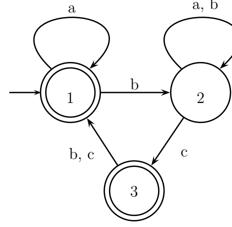
Definition 1.2.9 (Deterministic Automaton)**.** A deterministic Automaton G is a five – tuple

$$G = (X, \Sigma, f, x_0, X_m)$$

where:

- X is the set of *states*,
- Σ is the set of events, which define the alphabet of the generated language,
- $f : X \times \Sigma \to X$ is the, possibly partial, transition function,
- x_0 is the initial state and
- $X_m \subseteq X$ is the set of *marked states*.

Figure 1.1: A simple automaton



Every language, which can described using a regular expression, can also be expressed as an finite automaton, where the set of states is finite.

Example 1.2.10. The automaton in Figure 1.1 describes a Discrete Event System with the event set $\Sigma = \{a, b, c\}$ and three states, $X = \{1,2,3\}$. The marked states are $X_m = \{1,3\} \subseteq X$, visualized by a double circle, and the initial state is 1, marked by an arrow. The transition function f is partial in this case.

$$\begin{array}{lll} f(1,a) = 1, & f(1,b) = 2, & \\ f(2,a) = & f(2,b) = 2, & f(2,c) = 3, \\ & f(3,b) = & f(3,c) = 1 \end{array}$$

The transitions $f(1,c)$ and $f(3,a)$ are not defined.

The transition function f can be extended to $\hat{f}$, which can operate sequences of transitions in the following way:

$$\begin{array}{lll} \hat{f}(x,\epsilon) & := x & \\ \hat{f}(x,\delta) & := f(x,\delta), & \delta \in \Sigma \\ \hat{f}(x, s\delta) & := f(\hat{f}(x,s),\delta), & s \in \Sigma^*, \delta \in \Sigma \end{array}$$

Example 1.2.11. So the extended transition function of the automaton of Figure 1.1 would be

$$\hat{f}(1, abc) = f(\hat{f}(1,ab),c) = f(f(f(1,a),b),c) = f(f(1,b),c) = f(2,c) = 3$$

As we said an automaton G is presenting two languages. The generated language $L(G)$ and the marked language $L_m(G)$, which are defined as follows.

$$L(G) := \{s \in \Sigma : \hat{f}(x_0,s) \text{ is defined}\}$$

$$L_m(G) := \{s \in L(G) : \hat{f}(x_0,s) \in X_m\}$$

Example 1.2.12. The languages corresponding to the automaton in Figure 1.1 are

$$L(G) = (a^*b(a+b)^*c(b+c))^* \, (\epsilon + a^* + a^*b(a+b)^*(\epsilon + c + c(b+c)))$$

$$L_m(G) = (a^*b(a+b)^*c(b+c))^* \, (a^* + a^*b(a+b)^*(c + c(b+c)))$$

Definition 1.2.13 (Equivialent Automaton)**.** Two automaton G_1 and G_2 are said to be *equivalent* if

$$L(G_1) = L(G_2) \qquad \text{and} \qquad L_m(G_1) = L_m(G_2)$$

Definition 1.2.14 (Accessible Part)**.** A state x of G is said to be *accessible* if in the language generated by G, $L(G)$, exists at least one sequence of events $s \in L(G)$, which satisfies

$$x = \hat{f}(x_0, s)$$

All accessible states of an automaton build the *accessible part* $Ac(G)$. A automaton is said to be *accessible* if $Ac(G) = G$.

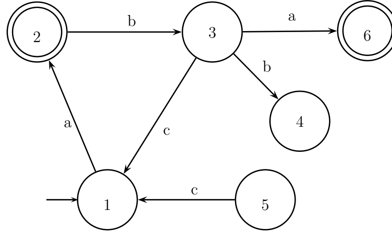
Definition 1.2.15 (Coaccessible Part)**.** A state x of G is said to be *coaccessible* if in the marked language generated by G, $L_m(G)$, there exists at least one sequence of events $s \in Lm(G)$, which goes through x. All coaccessible states of an automaton build the *coaccessible part* $CoAc(G)$. An automaton is said to be *coaccessible* or *nonblocking* if $CoAc(G) = G$ or

$$\overline{L_m(G)} = L(G)$$

.

Definition 1.2.16 (Trim Part)**.** An automaton or a part of an automaton, which is accessible and coaccessible, is said to be *trim* or the *trim part*.

Figure 1.2: A not accessible and not coaccessible automaton



Example 1.2.17. The automaton shown in Figure 1.2 is neither accessible nor coaccessible. The accessible part contains the states $X_{Ac} = \{1,2,3,4,6\}$. The states $X_{CoAc} = \{1,2,3,5,6\}$ are coaccessible and the states $\{1,2,3,6\}$ build the trim part of the automaton. Although in state 6 no more transition is possible, it is coaccessible, because it is marked.

Definition 1.2.18 (Blocking)**.** An automaton G is said to be *blocking*, if

$$\overline{L_m(G)} \subset L(G).$$

Example 1.2.19. The automaton in Figure 1.2 is blocking because from state 4 it is not possible to reach a marked state.

$$abb \in L(G) \quad \text{but} \quad abb \notin \overline{L_m(G)}$$

Often it is useful to model small parts of a bigger system and afterwards add them to the complete model. Therefore the *parallel composition* is used.

Definition 1.2.20 (Parallel Composition)**.** The *parallel composition* of the automata G_1 and G_2 is the automaton

$$G_1||G_2 := Ac(X_1 \times X_2, \Sigma_1 \cup \Sigma_2, f_{1||2}, (x_{0_1}, x_{0_2}), X_{m_1} \times X_{m_2})$$

with

$$f_{1||2} : (X_1 \times X_2) \times (\Sigma_1 \cup \Sigma_2) \to (X_1 \times X_2)$$

$$f_{1||2}((x_1,x_2),\delta) := \begin{cases} (f_1(x_1,\delta), f_2(x_2,\delta)) & \text{if } \delta \in \Sigma_1 \cap \Sigma_2 \text{ and } \delta \in \Sigma_1(x_1) \cup \Sigma_2(x_2) \\ (f_1(x_1,\delta), x_2) & \text{if } \delta \in \Sigma_1, \delta \notin \Sigma_2 \text{ and } \delta \in \Sigma_1(x_1) \\ (x_1, f_2(x_2,\delta)) & \text{if } \delta \in \Sigma_2, \delta \notin \Sigma_1 \text{ and } \delta \in \Sigma_2(x_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note, that $\Sigma(x)$ is the set of all events, which can occur in a state x.
The parallel composition is commutative and associative.

$$G_1||G_2 = G_2||G_1 \qquad G_1||(G_2||G_3) = (G_1||G_2)||G_3$$

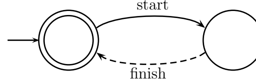
Since now we did not make any difference between events assuming their behavior would be equal. In fact two properties of events need to be explained before talking about Supervisory Control Theory.

An event is said *controllable* if it can be prevented from happening, or disabled, by supervisors. An *uncontrollable* event cannot be prevented from happening. The disjoint subsets of controllable events Σ_c and the uncontrollable events Σ_{uc} build the event set of a system.

$$\Sigma = \Sigma_c \dot{\cup} \Sigma_{uc}$$

In the following controllable events will be illustrated as a solid arrow and uncontrollable ones as a dashed arrow, as shown in Figure 1.3.

Figure 1.3: A simple workstation



Example 1.2.21. A simple system could be a single machine as in Figure 1.3 processing a piece of work. The event *start* to start the process is controllable while the event *finish* only depends on the machine and cannot be prevented to happen.

Another important property of events is observability. An event, which can be detected by the supervisor is called *observable*, one, that cannot be seen by the supervisor is said to be *unobservable.* The disjoint subsets of observable events Σ_o and unobservable events Σ_{uo} build the event set of a system.

$$\Sigma = \Sigma_o \dot{\cup} \Sigma_{uo}$$

As all events in the future example of this work will be observable, the problem of unobservability won't be discussed here any further. To learn more about dealing with unobservability chapter 3.7 of *Introduction to Discrete Event Systems* by Cassandras and Lafortune, [8], is recommended.

1.3 Supervisory Control Theory

1.3.1 Construction of an optimal, nonblocking supervisor

After discussing basic facts about Discrete Event Systems, now we want to introduce Supervisory Control Theory.

One of the most important elements of continuous control is the idea of feedback. One or more output signal from the system is used to calculate the best input signals in order to influence the behavior of the system and its outputs.

This main idea fits also to supervisors in Discrete Event Systems. Observable events can be detected by sensors in order to disable controllable events according to these informations to influence the systems behavior.

A supervisor is an automaton S, with the same event set as the plant G, $\Sigma_G = \Sigma_S$. The occurrence of an event in the plant will also cause transitions in the supervisor. In each state of the supervisor one or more controllable event can be disabled in G in order to satisfy given specifications. Certainly the language generated by the controlled system can be smaller or equal than $L(G)$. If the System is controllable, the controlled System S/G can be described by the parallel composition $S||G$.

$$K := L(S/G) = L(S||G) \subseteq L(G) \qquad L_m(S/G) = L_m(S||G) \subseteq L_m(G)$$

K is the language generated by the controlled system.

To control a DES using a supervisor the following steps need to be implemented.

1. Model the plant, which should be controlled.
2. Create automata that represent the specifications, which should be respected.
3. Generate a nonblocking and optimal supervisor.

Often the system can be divided in smaller parts. After modeling all parts separately the plant G is obtained by building the parallel composition of all parts.

Normally the controlled system should satisfy more than one specification E_i. The conjunction of all specifications is build by the parallel composition E of all automata representing individual requirements.

How to model a plant and its specification will be discussed further in Chapter 2.

The plant G and the conjunction of requirements E now are taken to build the supervisor S in the following steps.

1. Build the parallel composition of the plant G and the conjunction of all specifications E: $R = G||E$
2. Delete all non accessible and non coaccessible states of R.
3. Find all forbidden states. If there aren't any , $R = S$.

4. Delete all forbidden states and go back to step 2.

A state $q = (x,\cdot)$ of R is said to be forbidden, if an uncontrollable event $\delta \in \Sigma_{uc}$ could occur in the plant, $\delta \in \Sigma_G(x)$, but not in R, $\delta \notin \Sigma_R(q)$. Or, in other words, the supervisor would have to prevent the occurrence of an uncontrollable event, that cannot be disabled.

The result of this algorithm is the supervisor $S = (\Omega,\Sigma,f_s,\omega_0,\Omega_m)$. To every state in this automaton corresponds a list of controllable events, which could occur in the plant in the certain state, but not in the supervisor and thus must be disabled.

It is easy to see, that the algorithm to compute a supervisor above will always converge. The worst case would be an empty supervisor when all states were deleted.

Definition 1.3.1 (Controllability)**.** Let K and $M = \bar{M}$ be languages over event set Σ. Let Σ_{uc} be a designated subset of Σ. K is said to be *controllable* with respect to M and Σ_{uc} if

$$K\overline{\Sigma}_{uc} \cap M \subseteq \overline{K}.$$

1.3.2 Reduction of supervisors

To create a supervisor often the parallel composition of parts of the plant and of divers automata representing specifications need to be built. That can cause an exponential growth in the worst case. So even for small systems supervisors can reach huge sizes.

The main problem with very big supervisors is not only the needed memory space for implementation but also the loss off understandability.

In order to solve this problem, A.F. Vaz and W.M. Wonham developed in [25] a first idea how to reduce supervisors, which was taken up in [18], [26] and [24]. We only want to give a short introduction to this algorithm, based on [25].

First we have to define a supervisor S as a combination of an automaton $\{Q,\Sigma,\xi,q_0,Q_m\}$, and a control law $\psi : Q \times \Sigma \rightarrow \{0,1,dc\}$. (Note, that S now is a six – tuple.) ψ is defined as follows.

Definition 1.3.2 (Control Law)**.**

$$\psi(q,\delta) = \begin{cases} 0 & \text{if } \delta \text{ is disabled }, (\exists s \in K : \xi(q_0,s) = q, s\delta \in L, s\delta \notin K) \\ 1 & \text{if } \delta \text{ is enabled }, (\exists s \in K : \xi(q_0,s) = q, s\delta \in K) \\ dc & \text{if } \delta \text{ don't care }, (\exists s \in K : \xi(q_0,s) = q, s\delta \notin L) \end{cases}$$

The main idea is to merge states in *covers*.

Definition 1.3.3 (Cover of S)**.** A *cover* of a supervisor $S = \{Q,\Sigma,\xi,q_0,Q_m,\psi\}$ is a family $C = \{Q_i, i \in I\}$ of subsets of Q with the following properties

$$(\forall i)Q_i \neq \emptyset;$$

for a subset $I_m \in I$,

$$Q_m = \cup\{Q_i | i \in I_m\}, \quad Q - Q_m = \cup\{Q_i | i \in I - I_m\};$$

$(\forall i,\delta) : (\exists y \in Q_i)\xi(y,\delta)$ is defined

$\Rightarrow (\forall q \in Q_i)(\exists j) \cdot \xi(q,\delta)$ is defined $\Rightarrow \xi(q,\delta) \in Q_j$

$(\forall i, \delta)(\forall x,y \in Q_i)\psi(x,\delta) \neq dc \neq \psi(y,\delta) \Rightarrow \psi(x,\delta) = \psi(y,\delta)$

That means basicly, that marked and non marked states are merged in covers. The cover elements behave in the same way under the transition function and they exhibit uniform control action at those states where control matters.

We define a reduced supervisor $\bar{S}$ as follows

Definition 1.3.4 (Reduced Supervisor)**.**

$\bar{S} = \{I,\Sigma,\bar{\xi},i_0,I_m,\bar{\psi}\}$

select $i_0 \in I$ such that $q_0 \in Q_{i0}$;

define $\bar{\xi} : I \times \Sigma \rightarrow I$ as follows:

for $\delta \in \Sigma,i \in I$ select $j \in I$ such that $\bar{\xi}(i,\delta) := j$;

define $\bar{\psi} : I \times \Sigma \rightarrow \{0,1,dc\}$ as follows:

for $\delta \in \Sigma, i \in I,$ if there exist $q \in Q_i$ such that$\psi(q,\delta) \neq dc$

then let $\bar{\psi}(i,\delta) := \psi(q,\delta)$;

otherwise let $\bar{\psi}(i,\delta) := dc$.

Normally there are more than one possibility to build covers. So there will be more than one possible reduced supervisor, depending on the selection of covers. But clearly every reduced supervisor has to generate the same language in combination with the plant.

$L(S/G) = L(\bar{S}_1/G) = L(\bar{S}_2/G)$

1.4 Tools

To handle bigger systems it is recommendable to use some software tools as *GRAIL*, *IDES* or *TCT*. The implemented algorithms can be used to model, analyze and edit automata as well as compute supervisors and, at least in *GRAIL* and *TCT*, reduce supervisors. *IDES* can even be used to generate PLC–code as Instruction List.

There exist of course more programs or toolboxes, that treat discrete event systems. Since we just used *GRAIL*, *IDES* and *TCT*, we will only introduce these three tools.

1.4.1 TCT

TCT was developed by the *Systems Control Group* of the *University of Toronto*. It is a clearly arranged program for the *"synthesis of supervisory controls for untimed discrete-event systems."*,[28]. Automata are saved as a a five-tuple

$$[Size, Init, Mark, Voc, Tran]$$

Size is the number of states. The states will be numbered $\{0, 1, 2, \ldots Size - 1\}$. The initial state, Init, is allways 0. A list of all marked states is saved as Mark. Vocal states, Voc, are represented as a pair $[I,V]$, representing the positive integer output I at state V. Transitions are saved as triples [exit state, event, entrance state] in the list Tran.

Events can be named by positive integers between 0 and 999. Odd numbers represent controllable events, even numbers uncontrollable ones.

Among other things procedures to create, edit and show automata, add selfloops, build the trim part of an automaton, build the parallel composition of two automata, reduce a supervisor and test for conflicts and isomorphism are available.

For Windows and Linux the tool can be downloaded from Wonham's webpage,[7].

1.4.2 GRAIL

In 1994 *GRAIL* was introduced by Raymond and Wood, [22]. *Grail is a symbolic computation environment for finite-state machines, regular expressions, and other formal language theory objects.*, [3].

GRAIL is written in C++ and can handle regular expressions, automata and finite languages as well. The newest version (*GRAIL 3.0 / GRAIL+*) is available at Raymond's webpage [3].

The group of José E. R. Cury of the *Departamento de Automação e Sistemas* of the *Universidade Federal de Santa Catarina* also implemented a Multitasking toolbox, a Hierarchical toolbox and Condition/Event toolbox, which are available at Cury's webpage, [5].

Use of *GRAIL* is less restrictive than *TCT*. States are also named with natural numbers and the initial state has not to be 0. Event names can contain numbers and letters as well as underlines. It is also possible to create batch files to combine different functions.

To learn more about *GRAIL* a small guide is available on Raymond's, [3], and on Cury's webpage, [5]. We also can recommend C. Reiser's masterthesis, [23].

1.4.3 IDES

If a more user friendly program is desired, *IDES* may be a good choice. This java based program was developed by the *DES Lab* at *Queen's University*.

It allows a visual approach as automata are created and edited via drag and drop with the mouse. All common routines to process automata are implemented. Unfortunately an algorithm to reduce supervisors has not been implemented yet. The *IDES.jar* file, which runs under Windows, Linux and MacOSX, can be downloaded at the webpage [4]. A short users guide is available on the page as well.

In this project an experimental version of *IDES* with some additional features was used. Some often used automata are available as templates ore can be added to the templates library. After creating a template the user can easily combine different parts of the plant and specifications via arcs. It is also possibly to generate PLC–code (Instruction List), import and export files from and to *GRAIL* and *TCT*. More information will be available at [16].

Chapter 2

Models and specifications

Here we want to introduce the little mechanical model of a fabric cell, which will be controlled later via modular and monolithic supervisors. First a description of the physical system is presented, followed by the description of four possible ways to model the plant and their specifications for three of the four models. Finally these models will be compared.

2.1 Introduction

In real life most systems are based on continuous processes and can be described with continuous variables as velocity, time or high quite well. In some situations modeling a system with a continuous approach is possible but not recommended, because that would cause a huge systems with a lot of information, which is not needed to control the system. To model such a system as a Discrete Event System, it can help to reduce the model and focus only on the details needed to control the system.

To decide how actions of the real system are represented by events is a not trivial task. According to requirements different ways to model the system are possible. A very detailed model, which works with many specific events is a good way to describe the system close to the real problem. Very precise information can be a big advantage on the one hand but will lead to a larger model, which can be difficult to handle.

Corresponding to targeted control requirements, building a more abstract model can help to reduce the size of the model. To decrease the number of events some events can be merged. This reduction in complexity will lead to a smaller and easier to treat model. But the loss of details also involve a loss of information which may reduce the possible control also.

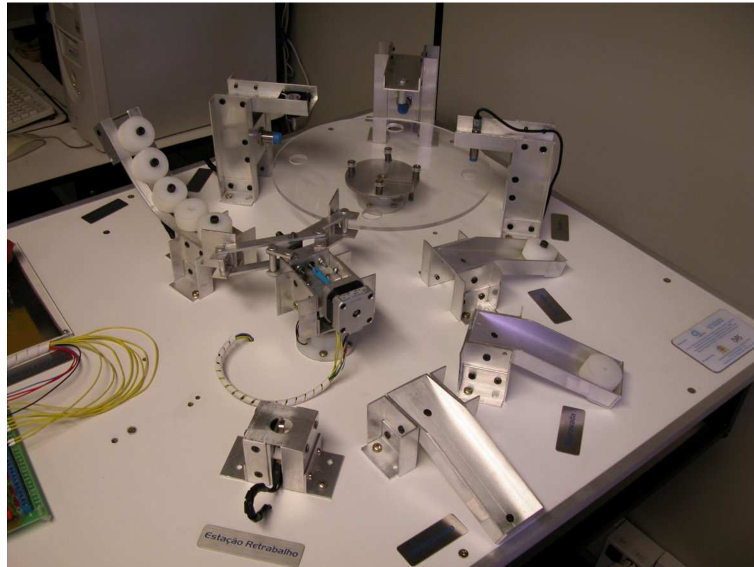
To model a real system as a Discrete Event System depends on the desired control tasks as well as the need of detailed information about the system state. After introducing the model of a fabric cell, in Section 2.2, four different ways to model it as a set of automata, in Section 2.4, Section 2.5 and Section 2.6, and the comparison of them, in Section 2.7, will follow in this chapter.

Some of the presented ways to model the specifications in the following chapter are based on [17].

2.2 Testbed

The manufacturing cell, shown in Figure 2.1 was constructed by students of *Curso Superior de Tecnologia em Automação Industrial – CSTAI* of the *Centro Federal de Educação Tecnológica de Santa Catarina – CEFETSC* in Florianópolis, Brazil, [2].

Figure 2.1: Photo of the testbet



2.2.1 Verbal description

In Figure 2.2 a plan of the cell is presented. The central element is the robot, which can rotate 360° in both directions. The locations around the robot are numbered from *0* to *6*. It is not possible to rotate more than 360°. If the robot arm passes the region between *6* and *0* it will be detected by sensor. The drive mechanism is realized by a stepper motor with two inputs. The first one for the direction of rotation and the second for the steps.

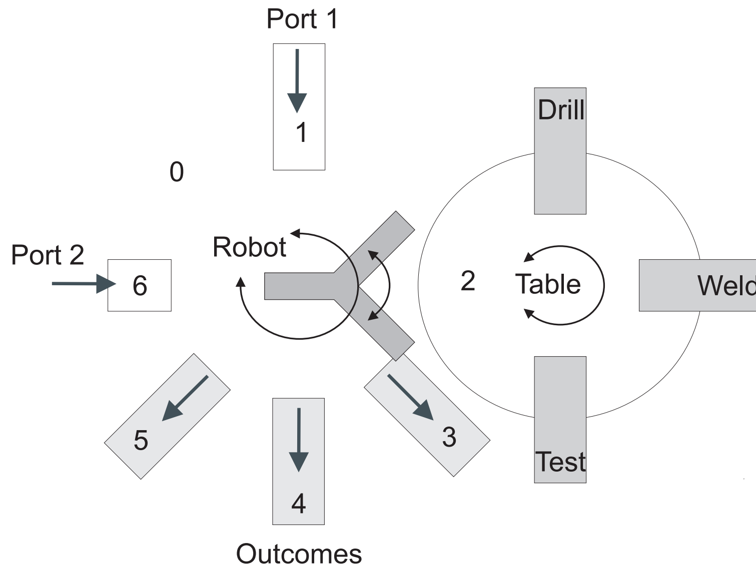
The grabber can close and open in order to grab or release a piece of work. It is also driven by a stepper motor with two inputs for the direction of rotation and the steps. The maximal opening position is detected by a mechanical sensor.

The rotation of the table in both directions is also driven by a stepper motor with two inputs for the direction of rotation and the steps. A sensor can be used to detect the table position at a certain place.

Pieces can arrive at two ports: port 1 at place *1* (standard input) and port 2 at place *6* (rework input). Incoming pieces has to be transported to the table at place *2*. If new pieces arrive at a port, they will be detected by the corresponding mechanical sensor at the port.

Around the table three workstations for drilling, welding and testing the pieces are modeled. A motor at the drilling station simulates the drilling pro-

Figure 2.2: Scheme of the testbet



cess. A capacitive sensor can detect if a piece of work is placed in front of the drilling station. The weld is simulated by a LED and a capacitive sensor to detect pieces. The test station is represented by an inductive sensor that can distinguish between an *approved* and a *disapproved* piece of work.

Pieces of work are plastic cylinders with a diameter of $4cm$ and a height of $2cm$ with or without a metallic screw on top, shown in Figure 2.3. Pieces with a screw should be *approved*, pieces without one should be *disapproved*.

Figure 2.3: Photo of pieces of work

The grabber can release pieces of work at three outcomes situated at position *3* (approved pieces), *4* (disapproved pieces) and *5* (rework) between the table at position *2* and the second port at position *6*.

2.2.2 Sensors and actuators

A table with the inputs and outputs is shown in Table 2.1 and Table 2.2.

Table 2.1: Inputs

In	Sensor type	Location	Function
E1	mechanical	port 1	detects arriving pieces at port 1
E2	mechanical	grabber	detects if the grabber is opened
E3	mechanical	robot arm	detects arm in reference position
E4	mechanical	table	detects a certain table position
E5	capacitive	drill	detects pieces in front of drilling station
E6	capacitive	weld	detects pieces in front of welding station
E7	inductive	test	detects if pieces are with or without screw
E8	mechanical	port 2	detects arriving pieces at port 2

Table 2.2: Outputs

Out	Actuator type	Location	Function
S1	stepper motor	grabber	information about direction of rotation
S2	stepper motor	grabber	number of steps
S3	stepper motor	robot arm	information about direction of rotation
S4	stepper motor	robot arm	number of steps
S5	stepper motor	table	information about direction of rotation
S6	stepper motor	table	number of steps
S7	DC motor	drill	simulate drilling
S8	LED	weld	simulate welding

2.2.3 Desired proceeding

Pieces coming from the standard income (port 1) should be transported by the robot to the table and pass the drilling and the welding station. Afterwards pieces need to be tested. If the test is positive, which means that the piece of work holds a screw, the pieces should be deposited at outcome *3*. *Disapproved* pieces should go to outcome *5*. They will be reworked in a separated, not modeled section.

After being reworked, pieces will enter in the system again via the second port at position *6*. These pieces must not be drilled or welded again. They only need to be tested. Approved pieces should be deposited at outcome *3*. If a piece is disapproved the second time, it will not be reworked again and is deposited at outcome *4*.

That can be realized in different ways. It is possible to put pieces from port 2 at the table, rotate the table 90° counterclockwise, test it, rotate the table 90° clockwise and deposit the piece at the corresponding outcome.

Also it may be possible to always rotate the table clockwise. So if a piece stems from port 2, it should be ignored at the drilling and the welding station, tested and put away properly.

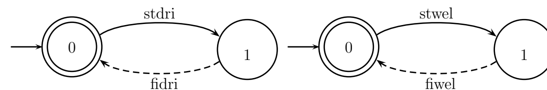
2.3 Similarities

2.3.1 Modeling the plant

Four different ways to model the plant described in Subsection 2.2.1 will be presented below. Although every model has its special characters some parts are the same in every approach. The automata modeling the table, the drill, the weld, the test and the two ports are equal in these four models.

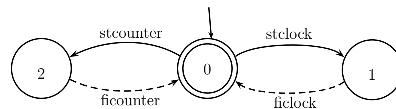
The drill and the weld are modeled in a similar way. The automata, presented in Figure 2.4, consist of two states and two transitions. If the event *stdri* or *stwel* occurs the plant will move from the initial state *0* (idle) to *1* (working). These events are controllable and start the drilling or the welding procedure. When finishing the routine, the uncontrollable event *fidri* or *fiwel* will cause the transition back to the state *0*. The initial state is marked because a completed task would mean, that the drilling or the welding was finished.

Figure 2.4: Model: drilling and welding station



The automaton representing the table, shown in Figure 2.5 is almost like modeling the drill and the weld. But the table can move in both directions, so three events are needed: *stclock* to start the table rotating 90 ° clockwise, *stcounter* to start the table rotating 90 ° counterclockwise and *fitab* to represent the finish of a rotation. The three states represent the status of the table: *0* if it is idle, *1* if it is rotating clockwise and *2* if it is rotating counterclockwise.

Figure 2.5: Model: table

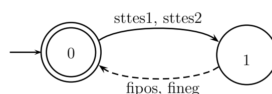


In some cases it will not be necessary to let the table rotate counterclockwise. So only states *0* and *1* and the events *stclock* and *fitab* are needed.

The test is started by the controllable event *sttes*. It can finish in two ways. If the piece got a screw on its top, it should be detected by the sensor and the event *fipos* will occur, otherwise the event *fineg* will occur. So the automaton would consist of two states and three transitions.

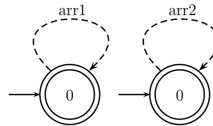
In some situations it can be helpful to have two events to start the test, *sttes1* and *sttes2*, as we see later. The automaton with two states and four transitions is shown in Figure 2.6.

Figure 2.6: Model: test



The arrival of pieces at port 1 and 2 will be modeled by the events *arr1* and *arr2*. As they do not depend on other events they are represented by two simple automaton with one state and a selfloop, illustrated in Figure 2.7.

Figure 2.7: Model: port 1 and port 2



If a piece of work is located at the drilling or the welding station, it will be detected by one of the two capacitive sensors. The signals are modeled by two automata with a single state and a selfloop, representing the uncontrollable events *sigdri* or *sigwel*, respectively.

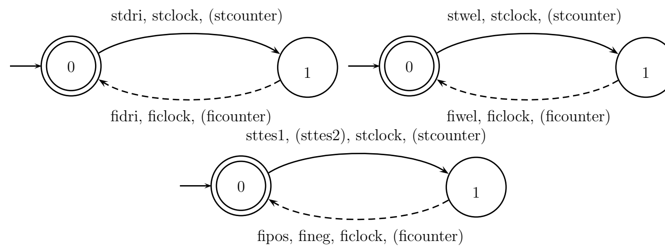
The most important part of the plant is the robot with the rotating arm and its grabber. It was modeled in four ways, mentioned in the following sections (Section 2.4, Section 2.5 and Section 2.6).

2.3.2 Modeling the specifications

Mutual exclusion, $E_{11} - E_{13}$

In any case it is important to make sure, that the table is not moving while a workstation, like the drill, the weld or the test, is working. This can be avoided by specifications of the kind *mutual exclusion*. It make sure, that only one process can run at a time. There are shown in Figure 2.8.

Figure 2.8: Specification: mutual exclusion, E_{11}, E_{12} and E_{13}



Avoid moving empty table, E_2

One of the most important elements in the manufacturing cell is the table, which moves pieces of work from one workstation to another. Although it is important to make sure that the table is able to move in most situations, it should not rotate while it is empty.

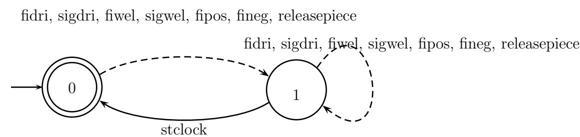
To detect, if a piece of work is located on the table, several events are used. A piece at the drilling station can be detected by *sigdri* or *fidri*, if you make sure, that the drill only runs with a piece.

The same can be said for the events *sigwel* and *fiwel*, corresponding to the welding station. A piece located at the test station is detected by the events corresponding to the test result, *fipos* or *fineg*.

If the robot put a piece on the table, it cannot be detected by a sensor. Nevertheless the table should rotate 90° clockwise. The event corresponding to the action *release a piece at the table* differs according to the model of the robot. It will be called *releasepiece* here and will be substituted later by the corresponding event of every model.

At least one of these events is needed to move from state *0* to *1* from which the event *stclock* to rotate the table clockwise is allowed.

Figure 2.9: Specification: avoid moving empty table, E_2



2.4 Model 1

2.4.1 Modeling the robot

It is easy to see, that the position *2* is the most important around the robot. So the situation *robot is situated next to the table with an open grabber* will be the initial state.

A piece can be taken from port 1 at position *1*. The event *stgra1* means that the grabber starts to move towards position *1*, closes the grabber in order to get a piece from port 1, moves back to position *2* and opens the grabber to release the piece. The end of this procedure is detected by the uncontrollable event *figra1*.

The events *stgra6* and *figra6* describe the action of getting a piece from the second port at position *6*. It is only needed if the model includes the rework.

After processing a piece at the workstations it should be taken from the table, moved to the corresponding outcome, the grabber should open to release the piece and move back to the table. The events *stput3*, *stput4* and *fiput* are needed in any case, *stput5* only if the rework is modeled as well. Because all procedures to put away a piece end with the same event, *fiput*, only 4 states to model this simple robot are needed. The automaton is shown in Figure 2.10.

2.4.2 Modeling the specifications without rework

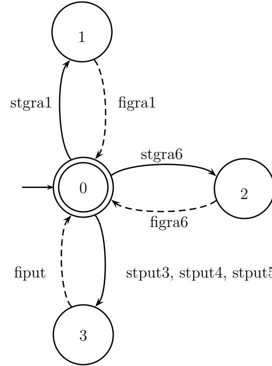
To start to model all specifications for the first model, we will begin without the rework. So the table only need to rotate clockwise (*stclock*, *ficlock*), only one test routine (*sttes1*) is needed. The model of the robot in composed of three states and five transitions (*stgra1*, *figra1*, *stput3*, *stput4*, *fiput*).

Approved pieces will be put to outcome *3* (*stput3*), disapproved ones to *4* (*stput4*). Now some more specifications than in Subsection 2.3.2 on page 20 are needed.

Mutual exclusion, E_{14}

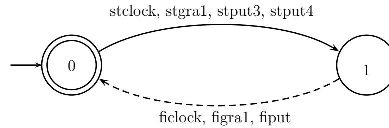
It is not desired that the table is moving while the robot is acting next to it. Because the model of the robot is very simple, from the moment of the

Figure 2.10: Model 1: robot



occurrence of *stgra1*, *stput3* or *stput4* till *figra1* or *fiput*, respectively, no information about the actual position of the robot is available. So in order to avoid collisions between the table and the robot it has to be avoided, that they work at the same time. This specification is modeled by a simple two states automaton, demonstrated in Figure 2.11.

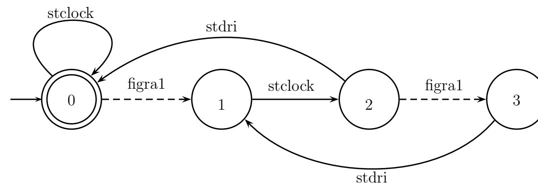
Figure 2.11: Specification model 1 without rework: mutual exclusion, E_{14}



Operating sequence, E_3's

If a piece of work was put on the table, it should be impossible to put a new one without rotating the table before. Furthermore every piece coming from port 1 should be drilled once. The automaton representing this specification is shown in Figure 2.12.

Figure 2.12: Specification model 1 without rework: operating sequence, E_{31}



If a piece was drilled, one want to make sure, that it won't be drilled twice. Also it is desired to make sure that a piece is welded once after being drilled. To guarantee this operating sequence the specification E_{32} is used, displayed in Figure 2.13.

In the same way E_{33} is used to make sure, that a welded piece is tested afterwards, shown in Figure 2.14.

Figure 2.13: Specification model 1 without rework: operating sequence, E_{32}

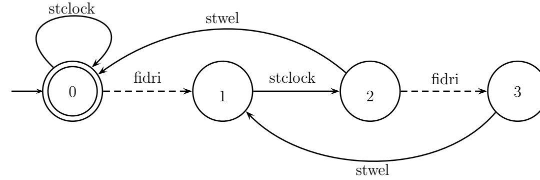
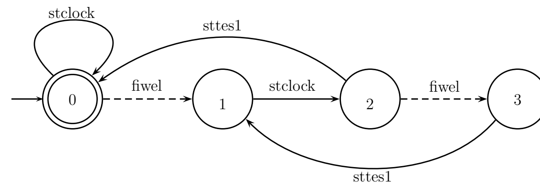
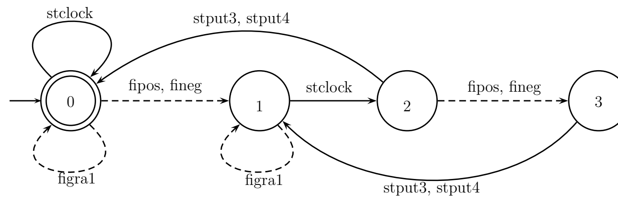


Figure 2.14: Specification model 1 without rework: operating sequence, E_{33}



Also after testing the piece once, the table should transport it to the position, where the robot can grab it and put it away, either to outcome *3* or *4*. As long as the place at the table next to the robot is empty (states *0* and *1*), it is allowed to release new pieces there. The event *figra1* can occur. The automaton is illustrated in Figure 2.15.

Figure 2.15: Specification model 1 without rework: operating sequence, E_{34}



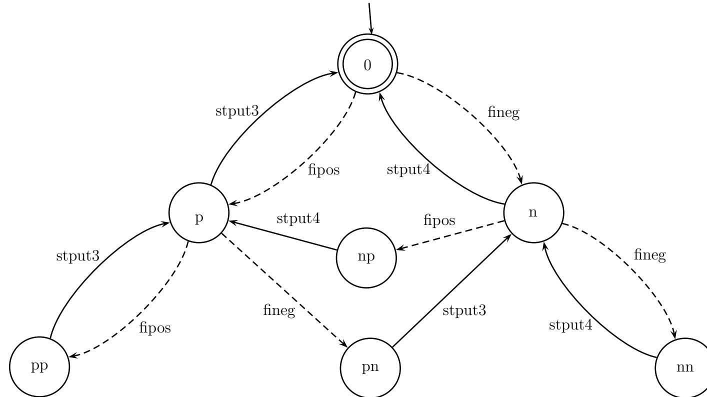
Chose the corresponding outcome, E_4

It is fundamental to assure to put every piece to the corresponding outcome. Approved ones should be put at *3*, disapproved ones to *4*. The automaton in Figure 2.16 consists of seven states. If no piece has been tested, it is in state *0*, if the test result is positive (*fipos*) it moves to state *p*, in case of a negative result (*fineg*) to *n*. After rotating the table it is possible that the next piece was tested before the first one was taken away from the table. So as to memorize the test result information the states *pp*, *pn*, *np* and *nn* are needed.

Get new pieces, E_5

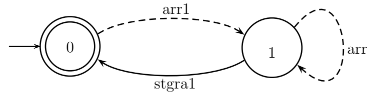
The robot only should move to position *1* to get a new piece (*stgra1*) , if one arrived at port 1 (*arr1*). If one piece arrived, it is mechanically not possible, that a second piece arrives before taking away the first. As the arrival was modeled as simple automaton with an uncontrollable selfloop (Figure 2.7), it

Figure 2.16: Specification model 1 without rework: corresponding outcome, E_4



is necessary to add a selfloop at state *1* of the specification in Figure 2.17, in order to get a non empty supervisor.

Figure 2.17: Specification model 1 without rework: get new pieces, E_5



2.4.3 Modeling the specifications with rework 1

As managed before, pieces coming from port 2 at position *6* only need to be tested. One idea to realize it, would be the following: Get a piece from port 2, put it onto the table, rotate the table 90 ° counterclockwise, test the piece and put it to outcome *3* or *4*, respectively.

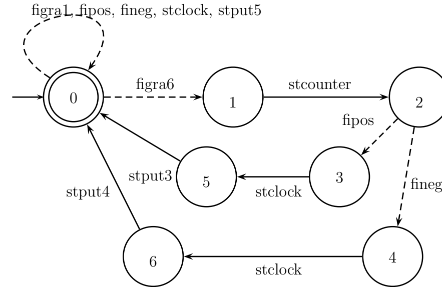
Alternative routine, E_6

We could call the treatment of pieces coming from the first port as standard. When getting a reworked piece the system should handle it in the way explained above and till putting it to the corresponding outcome interrupt the standard routine. The operational procedure for the alternative routine is organized by the specification E_6, presented below in Figure 2.18.

The alternative routine starts if a piece was taken from the second port (*figra6*). After rotating the table counterclockwise, testing the piece and rotating the table clockwise to the original position, it is put to outcome *3* or *4* depending on the test result.

As long as the system is handling pieces in the normal way, it should be possible to rotate the table only clockwise, get new pieces from port 1, test them or put them to *3* and *5*, so the corresponding events are added as a selfloop on state *0*. All events not mentioned in E_6 won't be disabled in any state.

Figure 2.18: Specification model 1 rework 1: alternative routine, E_6

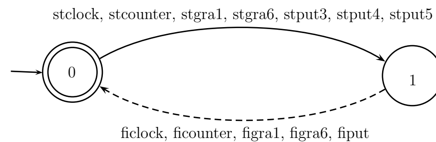


Actually the occurrence of all of them (for example *stdri* or *stwel*) will not be possible during the alternative routine. Later on we will see how other specifications will be modified to assure the desired behavior.

Mutual exclusion, E_1's

Now, as we also need to rotate the table counterclockwise, to take pieces from the second port at position *6* and put pieces to the outcome *5*, from where from it goes to the not modeled rework part, the events *stcounter*, *ficounter*, *stgra6*, *figra6* and *stput5* has to be added to E_{14}. The extended automaton is shown in Figure 2.19

Figure 2.19: Specification model 1 rework 1: mutual ecxlusion, E_{14}



stcounter and *ficounter* also need to be extended to the specifications E_{11}, E_{12} and E_{13}.

Operating sequence, E_3's

While handling standard pieces the operating sequences should be passed as mentioned in Subsection 2.3.2 and Subsection 2.4.2. But when entering at the alternative they should pause and after passing the routine continue at the same state.

For every state where you can enter to the alternative routine an extra state is added, corresponding the pausing standard routine. In this pausing state all events off the normal operating sequence, which are needed while the alternative routine, are added as a selfloop to these states (e.g. *stclock*).

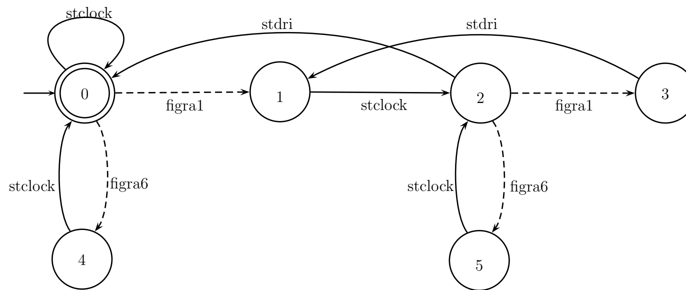
With which events the transition from a standard procedure state to a alternative procedure state and the transition back are made, depend on the specification.

The extended specification E_{31} is demonstrated in Figure 2.20. Only in the states *0* and *2* it is possible to enter the alternative routine with the event

figra6. States *1* and *3* correspond to situations, where a piece from port 1 already occupies the place on the table next to the robot.

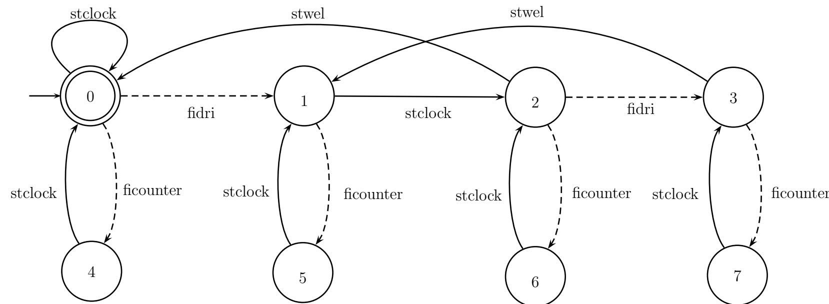
As the events *stput3, 4 or 5* are not part of E_{31} and before completing the alternative procedure it is not possible to get pieces from port 1 (*stgra1* and *figra1*), the event *stclock* can be used as the transition back to the standard routine.

Figure 2.20: Specification model 1 rework 1: operating sequence, E_{31}



In the specifications E_{32} and E_{33} a transition from a standard situation to a state corresponding to the alternative routine is always possible. The transition from the standard to alternative routine is made by the clockwise and counterclockwise rotation of the table.

Figure 2.21: Specification model 1 rework 1: operating sequence, E_{32}



Because testing a piece during the alternative procedure should be possible as well, a selfloop with *sttes1* has to be added to the states *4* to *7* in Figure 2.22. Otherwise *sttes1* would be disabled in these states.

New pieces should only be grabbed if the place on the table next to the robot is not occupied by a piece coming from the test station. So entering the alternative procedure as well as getting a new piece from port 1 is only possible at state *0* and *1* in Figure 2.23. Because the event *stput3* is also part of the alternativ routine, *stput3* and *stput4* are used in stead of *stclock* to mark the transition back to the standard routine. If *stclock* would have been used to mark the transition back to the standard routine, *stput3* and *stput4* would have to be added on state *0* and *1* in order not to disable them. That would mean, that *stput3* is not disabled at state *0*. So the robot could execute the corresponding subroutine without a piece located on the table. The occurrence of *stput4* would be disabled by the specification E_6.

Figure 2.22: Specification model 1 rework 1: operating sequence, E_{33}

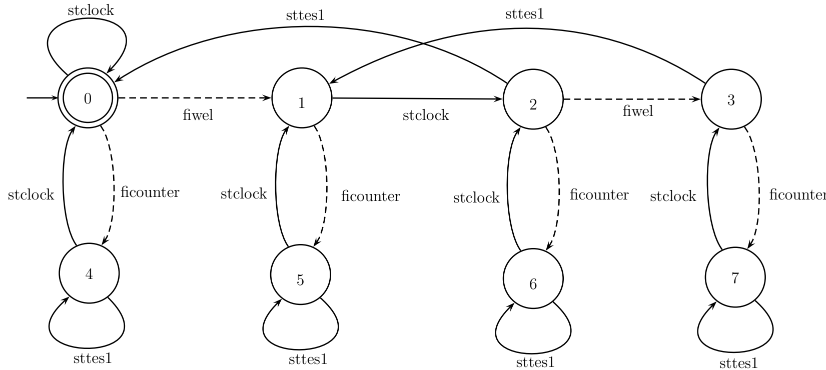
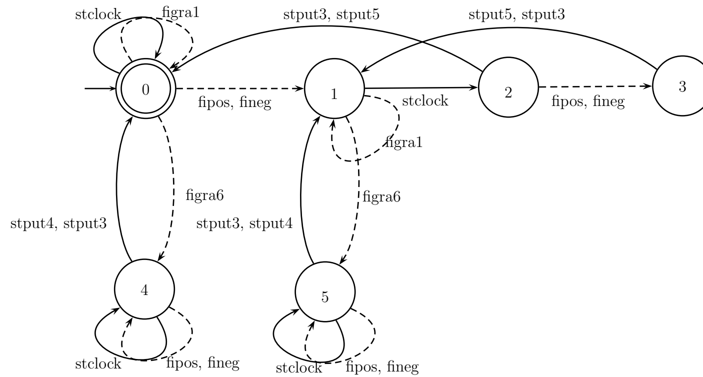


Figure 2.23: Specification model 1 rework 1: operating sequence, E_{34}



Choose the corresponding outcome, E_4

The automaton E_4 organizes where to release pieces during the standard routine. Also the extended automaton in Figure 2.24 care only about putting the pieces from port 1 to the corresponding outcome. Where to put reworked pieces after testing them is organized by E_6, Figure 2.18.

If two pieces were tested (states *pp*, *pn*, *np* and *nn*) without putting them away, the place on the table next to the robot is occupied. So get a new piece from port 2 and enter the alternative routine should not be possible and thus no extra state was added.

Get new pieces, E_5

Now it is possible to get pieces of work from port 1 and port 2, if a piece arrives at one of them. If at ports a new piece comes in, it would be possible to get them corresponding to the order of their arrival. Here we prefer to get a reworked piece, if pieces arrive at both ports. The automaton modeling this specification can be seen in Figure 2.25.

Figure 2.24: Specification model 1 rework 1: corresponding outcome, E_4

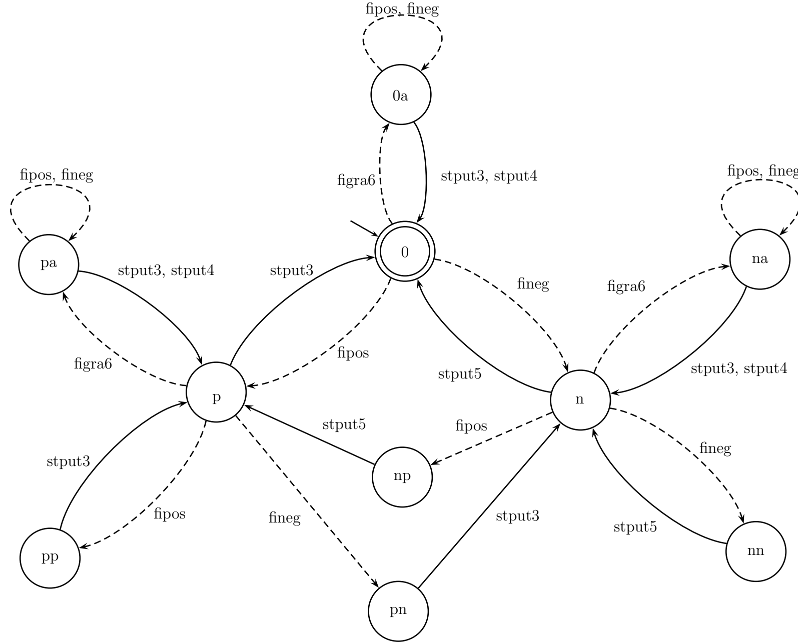
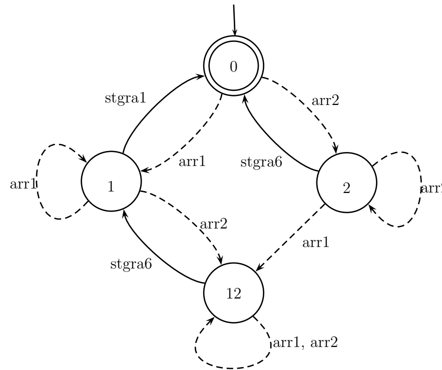


Figure 2.25: Specification model 1 rework 1: get new pieces, E_5



2.4.4 Modeling the specifications with rework 2

Another possibility to realize the treatment of the reworked pieces would be to rotate the table always clockwise. Thus reworked pieces should be ignored at the drilling and the welding station.

The automata for the specifications E_{11} to E_{14} (mutual exclusion) will be the nearly same as in Figure 2.8 and Figure 2.19. Of course we don't need the events *stcounter* and *ficounter* because the table does not require to rotate counterclockwise.

The events *sigdri* and *sigwel* in E_2, Figure 2.9, will assure, that the table can rotate even if there is only a reworked piece on the table, which is not treated at the drilling and the welding station.

The specification E_5, *Get new pieces*, will stay the same as in the first rework part in Figure 2.25.

Operating sequence, E_3's

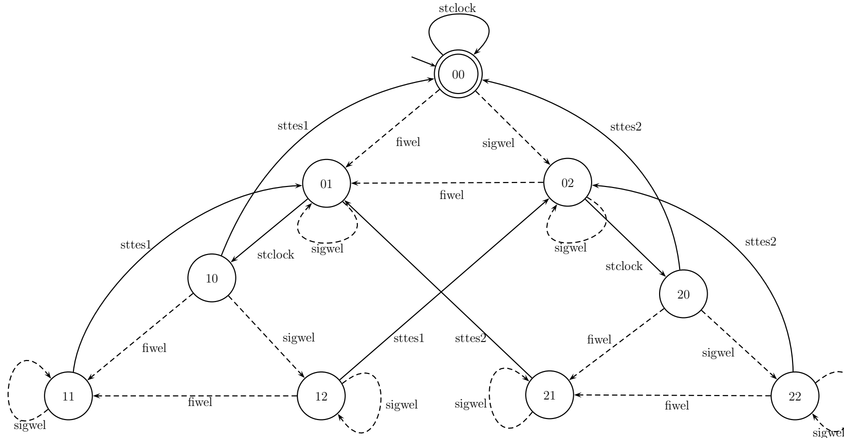
The operating sequences shown in Figure 2.20 and Figure 2.13 don't have to be changed. Only if a piece was taken from port 1 it will be drilled and only a drilled piece will be welded. So reworked pieces of worked will be ignored both at the drilling and the welding station.

However, the automaton in Figure 2.14 cannot be used anymore. It would only allow to test pieces from port 1, because they were drilled before.

We need to know, if a piece has been at the drilling station before rotating the table. The signal given by the sensor at the welding station (event *sigwel*) will show if a piece of work is situated at this workstation. If there was a piece, but it was not drilled it is reworked one. Pieces from port 1 will be drilled and detected by the sensor.

Now we will use two different events to represent the test of a piece. All states in Figure 2.26 are named by two digits. The first one corresponds to the piece at the testing station (*0* = no piece, *1* = piece from port 1, *2* = piece from port 2), the second to the piece at the welding station.

Figure 2.26: Specification model 1 rework 2: operating sequence, E_{33}



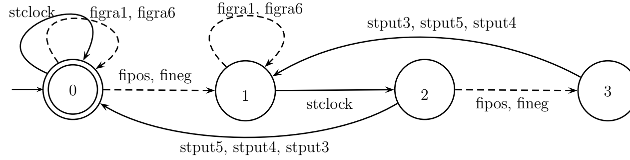
Reworked pieces will be tested by the routine 2 (*sttes2*), pieces from port 1 by 1 (*sttes1*). Physically both test routines are the same. Later on we will see, why it will help to make a difference between them.

Also the last specification to organize the operating sequence of Figure 2.15 need to be changed. As the events *stput5* and *figra6* now are needed as well, they have to be added. The result is presented in Figure 2.27.

Chose the corresponding outcome, E_4's

To choose the corresponding outcome now depends not only on the test result but also on the origin of the piece of work. It would be possible to create an automaton, which saves all information about the origin and the test result of a piece in order to distinguish between the three different outcomes.

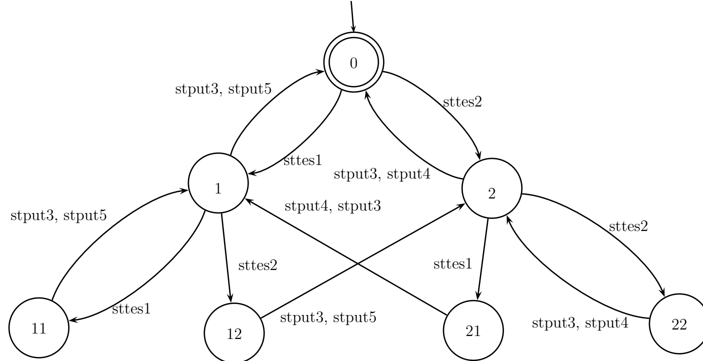
Figure 2.27: Specification model 1 rework 2: operating sequence, E_{34}



The automaton would have to save at least the information about three places on the table, drilling and test station and next to the robot and would probably grow because of that. By inventing a second test event, we can save the information about the origin of the piece by the corresponding test routine, realized by the automaton E_{33}, shown in Figure 2.26.

Pieces, which were treated the first time, will be tested by using the event *sttes1* and can be put to outcomes 3 or 5. Reworked pieces will be tested by using *sttes2* and should be put to 3 or 4, because they won't be reworked again. So the state *12* in **??** means, that first *sttes1* and then *sttes2* occurred. So the first piece was a standard one and should be put to outcome 3 or 5 and the second piece is already reworked and corresponds to the outcome 3 or 4.

Figure 2.28: Specification model 1 rework 2: corresponding outcome, E_{41}



The test result is also important to distinguish between the three outcomes. The specification E_{42} is almost the same as in Figure 2.16 on page 24. The only difference will be, that disapproved pieces can be put to outcome 4 or 5. So from the sates *n*, *nn*, *pn* not only the event *stput4* but also *stput5* is possible.

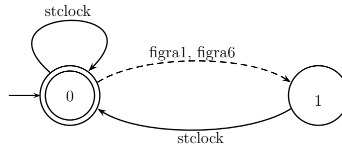
Sure, if a test result is positive the piece should go to outcome 3 and the origin of the piece is irrelevant. It would be possible to create a specification which combines both E_{41} and E_{42}.

The solution chosen here follows more the idea of modular control, where it is preferred to build more and smaller specifications. So the variation of small details should not cause the changing of a lot of specifications.

Only one piece at a time, E_6

Till now it would be possible to get two pieces from different ports or both from port 2 without rotating the table. To avoid this a simple automaton with two states (*0* if there is no piece on the table next to the robot, *1* if the place is already occupied), presented in Figure 2.29.

Figure 2.29: Specification model 1 rework 2: only one piece, E_6



2.5 Model 2

2.5.1 Modeling the robot

The Automaton to model the robot we created in the section above, Section 2.4, is a small but quite restrictive one. If the robot released a piece at one of the outcomes and want to get a piece from the second port afterward, it has to move back to position *2* next to the table before going back the same way to port 2 at position *6*. (See Figure 2.2.)

Another problem in the first model is that between the occurrence of the events *stgra1* and *figra1* no information about the actual position of the robot is available. So even if the grabber is empty or not next to the table it is not possible to detect. That is why we have to avoid the movement of the robot and the rotation of the table at the same time.

Thus, now we want to create another possible automaton to model the robot. The second model of the robot is less restrictive and features more possibilities to influence the movements of the robot.

Now not only the move from and towards position *2* (next to the table) is possible. Also the motion from position *3*, *4* and *5* towards port 2 at position *6* will be modeled now. This will be necessary to get a reworked piece after releasing one at an outcome.

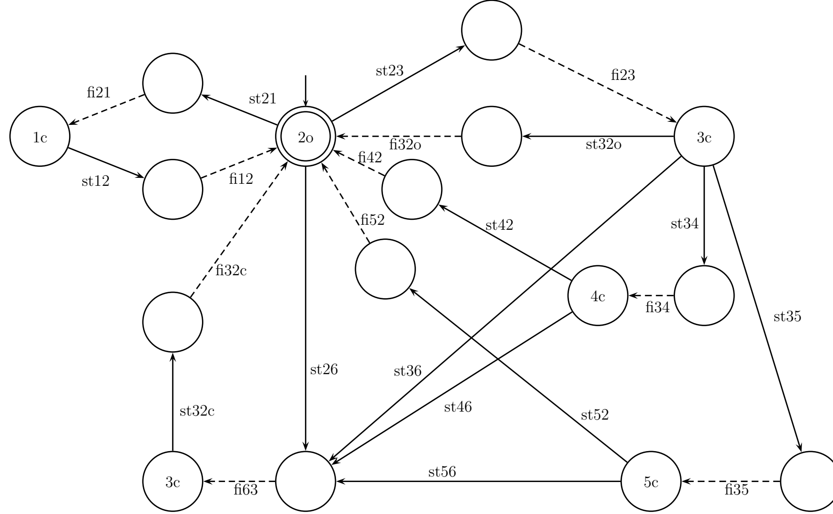
To detect more details about the actual position of the robot we will establish a security range around the table from position *1* to *3*. In this area we will avoid to move the closed grabber of the robot and the table at the same time.

As you can see in Figure 2.30, this extensions lead to a greater model with more transitions and states. The names of the most important states are generated by the position and the actual condition of the grabber (open or closed).

The events *st21* and *fi21* model the movement from position *2* to *1* and the closure of the grabber. The move back to the table and the opening of the grabber is presented by *st12* and *fi12*.

After closing the grabber at the table and moving to position *3* (*st23* and *fi23*), you can as well move to *4* or *5* by *st34* or *st35*, respectively. The occurrence of the event *fi23* will tell us, that the robot has left the security area around the table and it is possible to rotate the table.

Figure 2.30: Model 2: robot



At every outcome it is possible to open the grabber and go to position *2* or *6* by *st32o*, *st42*, *st52*, *st26*, *st36*, *st46* or *st56*, respectively. (The *o* at *st32o* means, that the grabber will be open while moving back towards the table.)

After moving to position *6* the grabber will close automatically and move back to position *3* (finish of this process is detected by *fi63*), before moving with the closed grabber to position *2* and opening the grabber at the table by *st32c* and *fi32c*.

2.5.2 Modeling the specifications without rework

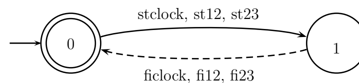
Again we want to start to model all specifications without respecting the rework unit. Thus the states and events corresponding with position *5* or *6* are not needed, (*st35*, *fi35*, *st52*, *fi52*, *st·6*, *fi63*, *st32c*, *fi32c*).

The specifications E_{11} to E_{13}, Figure 2.8, will remain like in model 1. The *releasepiece*–event in E_2, Figure 2.9, will be replaced by *fi12*.

Mutual exclusion, E_{14}

One reason to model the robot in the second way was to obtain more detailed information about the actual position of the robot. This information now is used in specification E_{14}, Figure 2.31.

Figure 2.31: Specification model 2 without rework: mutual exclusion, E_{14}



Only if the grabber is closed and the robot is acting in the security area around the table the table is not allowed to rotate. As well entering or leaving

the area around the table with a closed grabber while the table is rotating won't be admitted.

Operating sequence, E_3's

The specifications concerning the operating sequences will almost remain as for model 1. Thus the robot is not a part of the second and the third specification, they can be used without any change, Figure 2.13 and Figure 2.14.

In E_{31} and E_{34} the events *figra1*, *stput3* and *stput4* will be replaced by *fi12* and *st23*.

Figure 2.32: Specification model 2 without rework: operating sequence, E_{31}

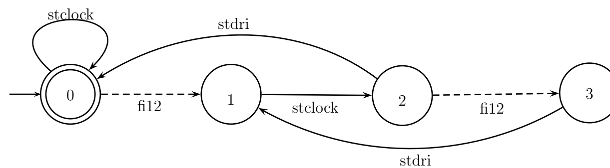
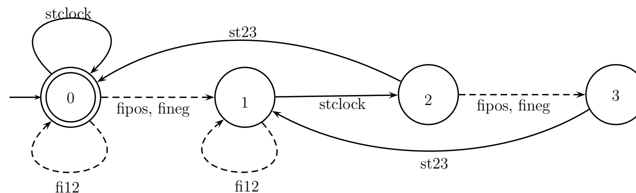


Figure 2.33: Specification model 2 without rework: operating sequence, E_{34}



Chose the corresponding outcome, E_4

As seen before approved pieces should be put to outcome *3*, disapproved ones to outcome *4*. The automaton in Figure 2.34 consists of 15 states. Since we let the table rotate while putting away a piece of work and after leaving the safety zone a third piece can be tested. Thus we need to remember the testresult of at most three pieces.

Get new pieces, E_5

Also the specification to make sure, that the robot only move towards port 1, if a piece arrived, remains almost the same as in Figure 2.17. The event *stgra1* has to be replaced by *st21*.

2.5.3 Modeling the specifications with rework 1

As managed in Subsection 2.5.3, pieces coming from port 2 at position *6* only need to be tested. So after getting a piece from port 2, put it onto the table, rotate the table 90° counterclockwise, test the piece and put it to outcome *3* or *4*, respectively.

Figure 2.34: Specification model 2 without rework: corresponding outcome, E_4

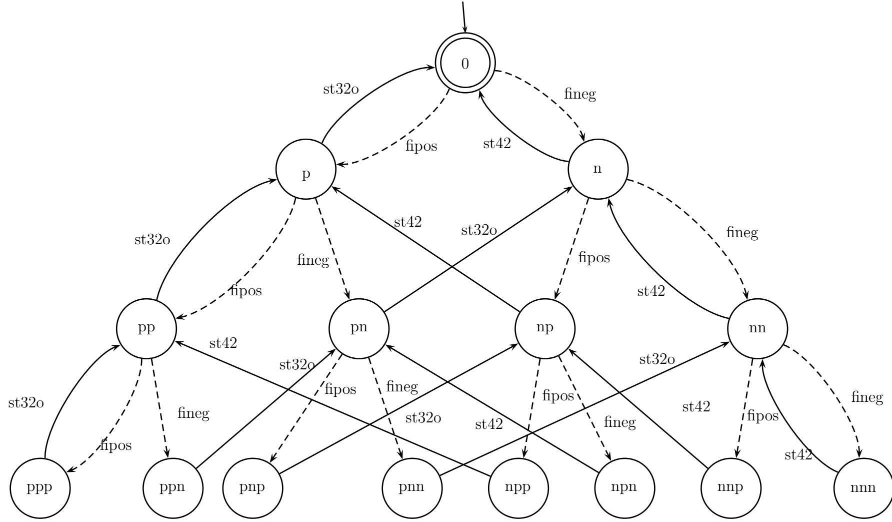
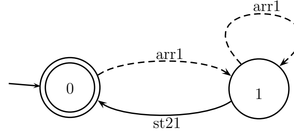


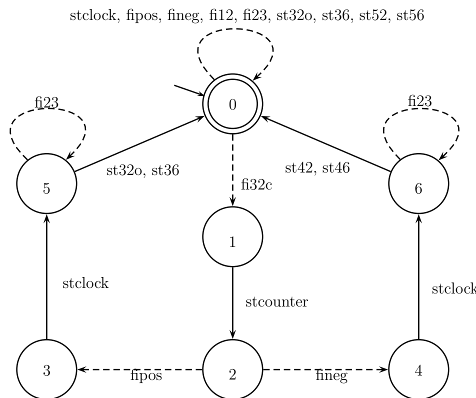
Figure 2.35: Specification model 2 without rework: get new pieces, E_5



Alternative routine, E_6

Again we will call the treatment of pieces coming from the first port as standard and the one for pieces from port 2 alternative routine. The operational procedure for the alternative routine is organized by the specification E_6, presented below in Figure 2.36.

Figure 2.36: Specification model 2 rework 1: alternative routine, E_6



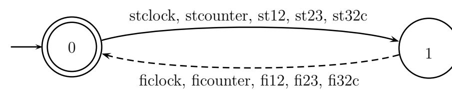
Now the alternative routine starts if a piece was taken from the second port and released at the table (*fi32c*). After rotating the table counterclockwise, testing the piece and rotating the table clockwise to the original position, it is put to outcome *3* or *4* depending on the test result.

To make sure, that the grabber closes in order to get the piece of work after rotating clockwise or within the standard routine, the event *fi23* is added as a selfloop to the states *0*, *5* and *6*. The supervisor will disable the event *st23* at all other states.

Mutual exclusion, E_1's

Again we have to add the counterclockwise rotation of the table to all specifications. The specification E_{14} has to be changed in order to adjust it to the robot.

Figure 2.37: Specification model 2 rework 1: mutual ecxlusion, E_{14}



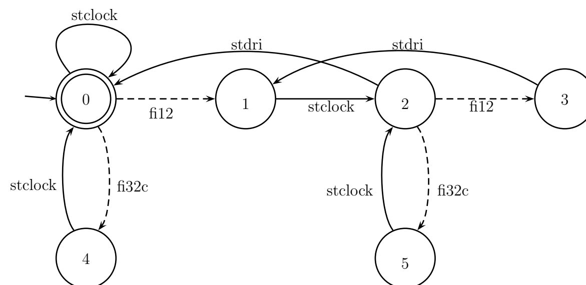
While the robot is putting a piece on the table or moving a piece away the table should not rotate.

Operating sequence, E_3's

The automaton shown in Figure 2.21 and Figure 2.22 can be adopted directly to the second model, because they do not include any event of the robot.

On E_{31}, shown in Figure 2.38, and E_{34}, illustrated in Figure 2.39, some little changes has to be realized. In E_{31} we will replace the events *figra1* and *figra6* by *fi21* and *fi32c*.

Figure 2.38: Specification model 2 rework 1: operating sequence, E_{31}



In E_{34}*stput3* and *stput4* have to be replaced by *st23* also.

Chose the corresponding outcome, E_4

As in Figure 2.34 we need 15 states to remember the testresults of three pieces. Only if the place on the table next to the robot is empty, the entrance into the alternative routine by *fi32c* is possible.

Figure 2.39: Specification model 2 rework 1: operating sequence, E_{34}

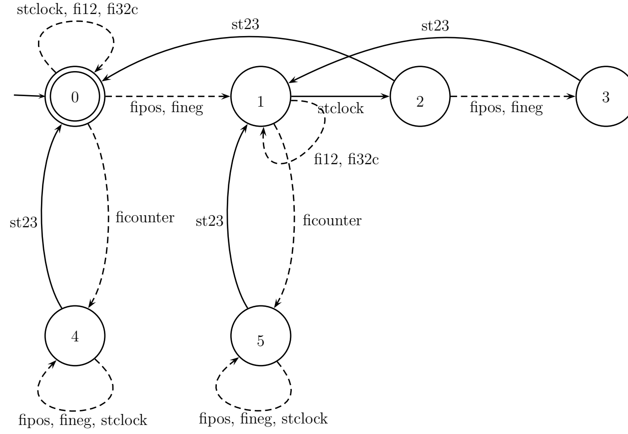
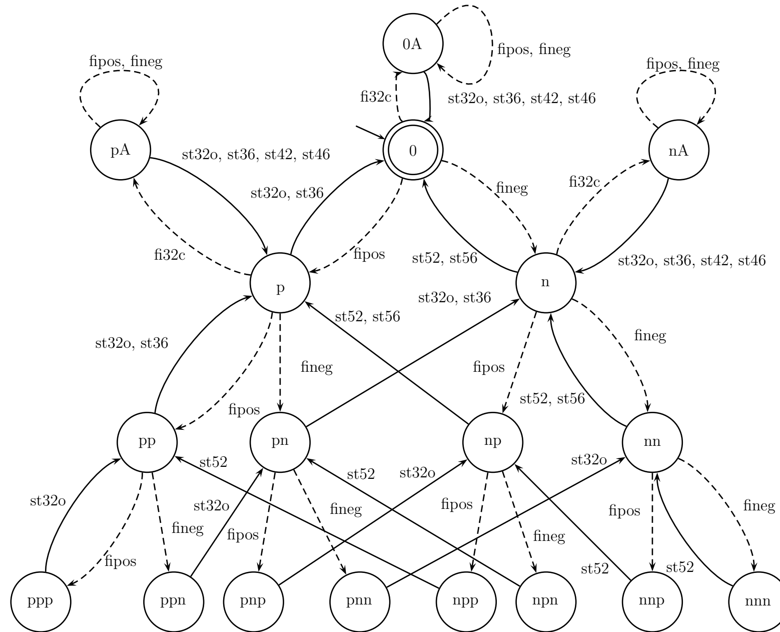


Figure 2.40: Specification model 2 rework 1: corresponding outcome, E_4



Get new pieces, E_5

E_5 need to be modified as well, but will stay in the same structure. Only the events *stgra1* and *stgra6* have to be replaced by *st21*, *st26*, *st36*, *st46* and *st56*.

2.5.4 Modeling the specifications with rework 2

The idea to realize the rework in the second way is the same as in Subsection 2.4.4.

Because the replacements of events will be similar as mentioned before, we will not show them in this section but just give an short overview.

The E1's are the same as in Subsection 2.5.3 without the events *stcounter*, *ficounter*. E_{34} will stay as in Subsection 2.5.2, E_{33} is equal to Figure 2.26.

To differentiate between the corresponding outcome, we will again use the information about the chosen testroutine and the testresult. The only difference will be, that we need to remember three testroutines and testresoults. So both automata have 15 states.

An automaton to make sure not to put two pieces on the table at the same place is needed as well.

2.6 Model 3 and 4

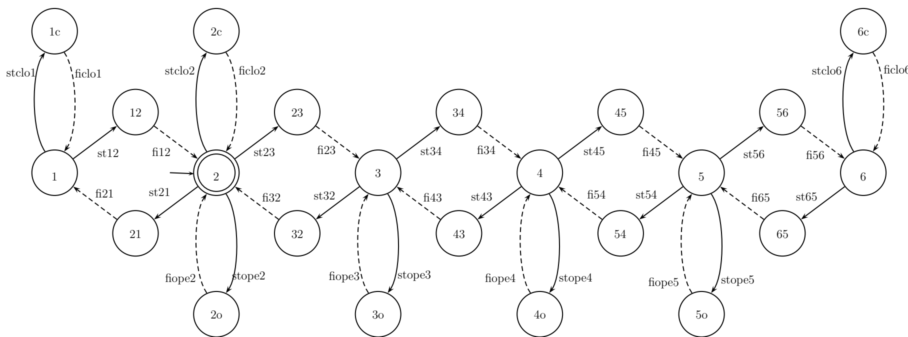
2.6.1 Modeling the robot

It would be possible to model the robot in an even less restrictive way. Every rotation of the robot from one position to another could be modeled separately.

The events representing the opening and closing of the grabber are only modeled at the positions where needed.

The model 3, shown in Figure 2.41 has states 23 and 34 transitions.

Figure 2.41: Model 3: robot



It is easy to see, that on the one hand this model gives the robot more possibilities to move. On the other hand a specification is needed to assure, that the events to close and open the grabber occur alternately. The information about the actual condition of the grabber is not saved in the states of the model.

It would be possible to model the rotating arm and the grabber of the robot separately as well. The two automata are presented in Figure 2.42 and Figure 2.43

Figure 2.42: Model 4: arm

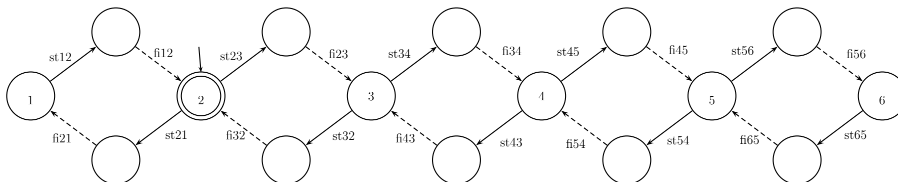
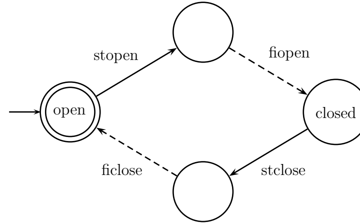


Figure 2.43: Model 4: grabber



With this model it would be possible to rotate the robot and open or close the grabber at the same time. Sure, that would give us a lot of possibilities to control the robot, but in this application it is not needed. So using this model would lead us to a complexity of the plant, which cannot be handled by the used tools.

Only the model of the robot, which would be the synchronous product of the Figure 2.42 and Figure 2.43 would consist of 64 states and 144 transitions.

We can see, that modeling every move of the robot separately causes very complex systems. So we will only try to model the system with the third model of the robot without rework. Therefor we can cut the states and transitions corresponding to the fifth and sixth position.

2.6.2 Modeling the specifications without rework

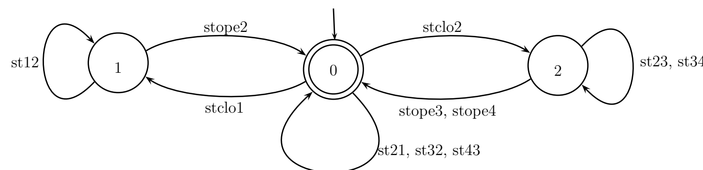
Robot control, E_6

We do not want to open the robot the grabber when it is already open or try to close the close grabber again. Closing and opening the grabber on position *2* one after another does not make a lot sense as well and should be avoided. Therefor a three states automaton, Figure 2.44, is created.

It forces the robot, once it moved from position *2* to *1*, to close the grabber, move back to position *2* and open the grabber again.

Then, once closed the grabber at position *2*, it is only possible to move to position *3* or *4*, open the grabber to release a piece and moving back with the open grabber.

Figure 2.44: Specification model 3 without rework: Robot control, E_6

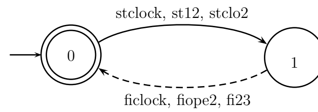


Mutual exclusion, E_{14}

Thus the robot is controlled and influenced by E_6 in Figure 2.44, we can used this information about the robots behavior to modify the automaton E_{14}.

While the the robot is moving with a new piece from port 1 to the table and opening the grabber we do not want the table to move. The same counts if the grabber is closing in order to put a treated piece away.

Figure 2.45: Specification model 3 without rework: Mutual exclusion, E_{14}



Operating sequence, E_3's

The two automata in Figure 2.13 and Figure 2.14 remain as mentioned before.

In Figure 2.12 and Figure 2.15 the events *figra1*, *stput3* and *stput4* have to be replaced by *fiope2* and *stclo2*.

Corresponding outcome, E_4

With 15 states the automaton record again the testresult of three pieces. Approved pieces should be put to outcome *3*, *stope3*, disapproved ones to *4*, *stope4*.

Get new pieces, E_5

The robot can only move from *2* to *1* if a piece arrived at the first port.

2.7 Comparison

In this section four different ways to model the robot have been presented. Depending on the application, every model can be more or less useful. It is necessary to find the best tradeoff size of the model on the one hand and the possibilities of the plant on the other hand. In this section we want to give an overview and compare the models in terms of their size, their features and their potentials.

First of all, it is useful to keep in mind, what is physical possible in every model. A short overview is given in Table 2.3.

Table 2.3: Physical possibilities of the models

Model	1	2	3	4
Move to port 2 without moving to the table first	n	y	y	y
Detect if the robot has left the area around the table	n	y	y	y
Move the robot step by step to every position	n	n	y	y
Move the robot and open the grabber at the same time	n	n	n	y

In Table 2.4 the number of states of all parts of the four models without the rework part and of the synchronous product of all parts of the plant (G) are shown.

Here is it easy to see, that even in the simple approach, without the possibility to rework pieces, models can grow very fast. Model 1 is 13 times smaller than model 4.

Table 2.4: Size of models without rework

Model	1	2	3	4
Table	2	2	2	2
Drill	2	2	2	2
Weld	2	2	2	2
Test	2	2	2	2
Port 1	1	1	1	1
Signal drill	1	1	1	1
Signal weld	1	1	1	1
Robot	3	10	15	10 4
Total without rework (G)	48	160	240	640

Table 2.5: Size of models with rework

Model	1	2	3	4
Table	3 / 2	3 / 2	3 / 2	3 / 2
Drill	2	2	2	2
Weld	2	2	2	2
Test	2	2	2	2
Port 1	1	1	1	1
Port 2	1	1	1	1
Signal drill	1	1	1	1
Signal weld	1	1	1	1
Robot	4	16	23	16 4
Total rework 1 (G)	96	384	552	1536
Total rework 2 (G)	64	256	368	1024

In Table 2.5 you can see, that not only the decision to realize the rework section in the model or not but also the way how it is realized influence the size of the uncontrolled plant.

Looking on the possibilities of the cell and keeping the desired behavior of it in mind, it gets clear, that model 3 and 4 are able to realize exactly the same desired behavior of the plant as the second model but are 1.5 or 4 times bigger than model 2.

The main problem of the first model is, that the robot and the table cannot move at the same time and the robot always has to go back to the table before moving to the second port. But the model is the smallest one and if the table and the robot are not the velocity determinant elements, this model could be an excellent choice.

If a monolithic supervisor is required, the synchronous product of all specifications is needed. According to the desired features of the plant and the wanted behavior of the controlled plant the result can grow fairly large.

So it is important to have a look at the size of the modeled specifications as well. A summary of all created specifications is given below in Table 2.6 and Table 2.7.

Table 2.6: Size of specifications without rework

Model	1	2	3
Mutual exclusion (table, drill), E_{11}	2	2	2
Mutual exclusion (table, weld), E_{12}	2	2	2
Mutual exclusion (table, test), E_{13}	2	2	2
Mutual exclusion (table, robot), E_{14}	2	2	2
Avoid rotating empty table, E_2	2	2	2
Operating sequence (robot, drill), E_{31}	4	4	4
Operating sequence (drill, weld), E_{32}	4	4	4
Operating sequence (weld, test), E_{33}	4	4	4
Operating sequence (test, robot), E_{34}	4	4	4
Corresponding outcome (robot, test), E_4	7	15	15
Get new pieces (robot, port 1), E_5	2	2	2
Robot control (robot), E_6	-	-	3
Total, $E = \|\|E_i$	3464	30180	12412
$E\|\|G$	1049	3614	6330

When building a synchronous product of two automata with a number of states a and b the result could have $c = ab$ states in the worst case. Looking at the size of the product of all specifications, E, make clear, that even with small and concise specifications the product of them can lead to hardly manageable automata.

Comparing the size of the specifications at Table 2.6 it is interesting that the specifications for model 2 have the same number of states as those for model 3. But the usage of different events and the additional appearance of the specification E_6 with three states, which only influences the behavior of the robot, leads to an automaton, that is smaller than the one for the second model.

If the rework is not implemented, it is not necessary that the robot moves to port 2. So the only disadvantage of system 1 compared to system 2 is, that the table and the robot cannot work on the same time. But, in return the generated automaton is almost nine times smaller.

In Table 2.7 it is interesting to see, that the uncontrolled plant is greater if the rework is realized in the first way, but the specifications in this case are smaller than in the second approach. With this information it is not possible yet to know which controlled system will be smaller in the end.

As you can see, it was not possible to build the synchronous product of the second model and its specification for rework 2. So it is not possible either to build the monolithic supervisor with this method and the used tools.

Table 2.7: Size of specifications with rework

Model, rework	1, rw 1	1, rw 2	2, rw 1	2, rw 2
Mutual exclusion, E_{11}	2	2	2	2
Mutual exclusion, E_{12}	2	2	2	2
Mutual exclusion, E_{13}	2	2	2	2
Mutual exclusion, E_{14}	2	2	2	2
Avoid rotating empty table, E_2	2	2	2	2
Operating sequence, E_{31}	6	4	6	4
Operating sequence, E_{32}	8	4	8	4
Operating sequence, E_{33}	8	9	8	9
Operating sequence, E_{34}	6	4	6	4
Corresponding outcome, E_4	10	7 7	18	18 15
Get new pieces, E_5	4	4	4	4
Alternative rout/one piece, E_6	7	2	7	2
Total, $E = \|\|E_i$	36192	57920	483000	2530816
$E\|\|G$	4388	6120	13773	-

Chapter 3

Supervisors

Having modeled the plant and the specifications as automata, they will be used in this chapter to calculate supervisors to control the system. After the introduction, not only the monolithic approach but also how to create modular supervisors and the problems and chances resulting from this approach will be discussed.

3.1 Introduction

Building the automata to model the uncontrolled plant and the specifications characterizing the desired behavior of the plant is the first and most difficult step to realize a supervisor.

How to create an optimal, nonblocking supervisor has been discussed in Subsection 1.3.1 on page 9. This routine is well defined and can be done by several tools. Some of them (*TCT*, *GRAIL* and *IDES*) have been presented shortly in Section 1.4 on page 11.

It is not always possible to calculate a supervisor. If the plant is not able to realize the wanted behavior without the possibility of the occurrence of an undesired uncontrollable event, the supervisor will be empty. In this case the plant is not controllable in the wanted way.

Even if it is possible to create an optimal, nonblocking supervisor, this does not automaticly mean that this is the best way to implement the system. Often supervisors obtain a huge number of states and events, which would have had to be implemented. Depending on the used PLC or micro controller the implemented system may need to much space on the disc.

Because the supervisor contains information about both the plant and the control sequences, it can be reduced. The result would be a reduced supervisor, which can be a lot smaller than the original supervisor and need to be implemented with the plant.

Although a reduced supervisor will exist in most cases, in some situations they cannot be calculated, because the used tools cannot handle the complexity of the system.

Choosing a modular approach would be another possibility to control these systems by Supervisory Control Theory. Calculating a number of small, local

supervisors instead of a monolithic one, may help to reduce the complexity of the system. A less complex system is not only easier to implement but also easier to understand, to realize small changes and to debug. The main disadvantage of a modular approach is the risk to get into a conclict situation.

In this chapter these two ways to obtain one or more supervisors to control the manufacturing cell will be presented. After presenting the resulting monolithic supervisors for the first three models in Section 3.2, the modular approach with the resulting supervisors and the handling of conflicting situations is discussed in Section 3.3. Before talking about how to implement the results on a PLC in Chapter 4 on 59, the different supervisors will be compared in Section 3.4.

3.2 Monolithic supervisors

The first step to generate a monolithic, optimal and nonblocking supervisor is to build the synchronous product (parallel composition) of the plant G and the specification E. The result is besides the model of the plant and of all specifications the largest automaton, which can occur during the calculation. As most programs only use a limited space on the hard disk, they cannot handle large and very complex systems. So, however, it may be possible to create a monolithic supervisor in theory, it does not have to be possible in practice by using standard tools.

The result of the routine described in Subsection 1.3.1 on page 9 is always an optimal, nonblocking supervisor, which generates the maximal controllable language of the system or an empty supervisor, if the system is not controllable with respect to the desired specifications.

As you can see in Table 3.1 not in all cases the calculation of the monolithic supervisor was feasible. The synchronous product of the plant G and the specification E is too large. While computing the supervisor TCT went out of memory. Later on, in Subsection 3.3.1, we will see, how the monolithic supervisor can be calculated in another way.

Table 3.1: Size of monolithic supervisors

Model	1	2	3
without rework	1326	3264	5760
rework 1	4132	12156	-
rework 2	6084	-	-

Although most monolithic supervisors could be computed, the reduction of them did not succeed. As the monolithic supervisors are too large to be implemented and impractical to debug, another approach to control the system is needed.

3.3 Modular supervisors

Every specification controls a certain part of the plant. So, to realize this specification and to calculate an adequate local supervisor only the local plant is re-

quired. The routine to compute remains how it was described in Subsection 1.3.1 on page 9.

3.3.1 Results

Model 1

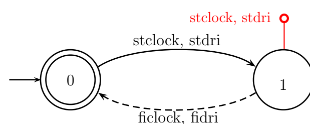
For all specifications for the first model without the possibility to rework pieces local supervisors were computed and reduced by *TCT*. The size of the local plants G, specifications E, local supervisors S and reduced supervisors R are resumed in Table 3.2.

Table 3.2: Local and reduced supervisors for model 1 without rework

	G_i	E_i	S_i	R_i
Mutual exclusion, 11	4	2	3	2
Mutual exclusion, 12	4	2	3	2
Mutual exclusion, 13	4	2	3	2
Mutual exclusion, 14	6	2	4	2
Avoid rot. empty table, 2	48	2	96	2
Operating sequence, 31	12	4	40	4
Operating sequence, 32	8	4	24	4
Operating sequence, 33	8	4	24	4
Operating sequence, 34	12	4	30	5
Corresponding outcome, 4	6	7	30	7
Get new pieces, 5	3	2	6	2

The first four reduced supervisors are fairly similar to their specifications. R_{11} is shown in Figure 3.1.

Figure 3.1: Reduced supervisor model 1 without rework: mutual exclusion, R_{11}



In state *1* the events *stclock* and *stdri* have to be disabled. In R_{12}, R_{13} and R_{14} instead of *stdri* the events *stwel*, *sttes1*, *stgra1*, *stput3* or *stput4* have to be disabled, respectively.

To avoid the rotation of the empty table, the specification E_2 was created. The supervisor with 96s can be reduced to a reduced supervisor with only two states, presented in Figure 3.2.

As the four supervisors organizing the operational sequence are similar, only the first resulting reduced supervisor is shown in Figure 3.3.

The resulting reduced supervisor to make sure, that every piece of work is put to the correct port according to the testresult R_4 is shown in Figure 3.4. In the states *pp*, *pn*, *np* and *nn* the event *sttes1* is disabled as well. Actually

Figure 3.2: Reduced supervisor model 1 without rework: avoid empty table, R_2

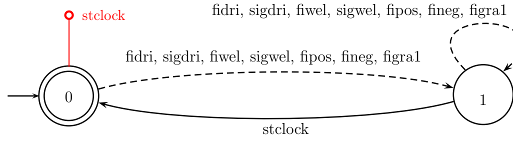
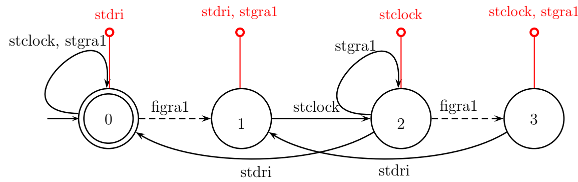
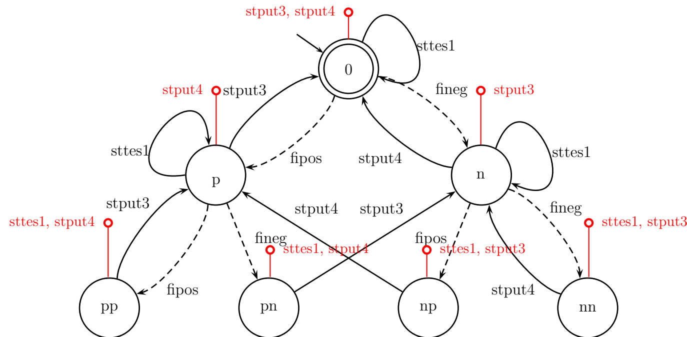


Figure 3.3: Reduced supervisor model 1 without rework: operating sequ., R_{31}



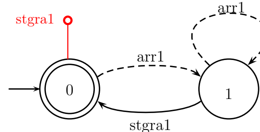
this is not needed because if two pieces have been tested without putting them away, the supervisor R_{34} will disable the event *sttes1*. If a deactivation of *sttes1* is not wanted in R_4, it would be possible to add selfloops with *fipos* and *fineg* at the corresponding state.

Figure 3.4: Reduced supervisor model 1 without rework: corres. outcome, R_4



The simple two state supervisor R_5 is shown in Figure 3.5. If port 1 is empty (no peace arrived), the event *stgra1* has to be disabled.

Figure 3.5: Reduced supervisor model 1 without rework: get new pieces, R_5



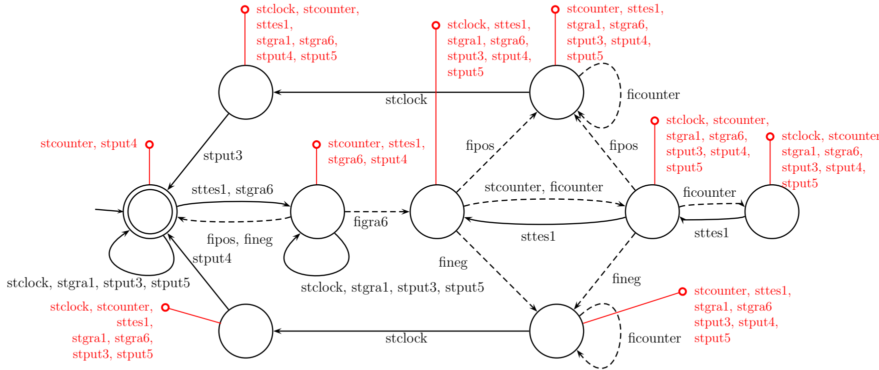
For the realization of the rework part in the first way, the specifications are already modeled. The size of the local plants G, specifications E, local supervisors S and reduced supervisors R are presented in Table 3.3

Table 3.3: Local and reduced supervisors for model 1 with rework 1

	G_i	E_i	S_i	R_i
Mutual exclusion, 11	6	2	4	2
Mutual exclusion, 12	6	2	4	2
Mutual exclusion, 13	6	2	4	2
Mutual exclusion, 14	12	2	6	2
Avoid rot. empty table, 2	96	2	192	2
Operating sequence, 31	24	6	84	5
Operating sequence, 32	12	8	40	11
Operating sequence, 33	12	8	40	10
Operating sequence, 34	24	6	66	8
Corresponding outcome, 4	8	10	46	14
Get new pieces, 5	4	4	16	4
Alternative routine, 6	24	7	28	9

First we want to have a look at the R_6, in Figure 3.6, which organize the alternative procedure.

Figure 3.6: Reduced supervisor model 1 rework 1: alternative routine, R_6



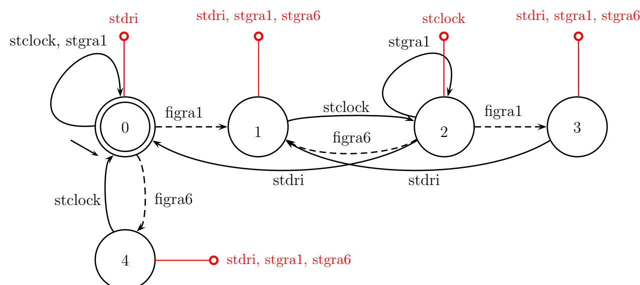
The resulting supervisors R_{11}, R_{12}, R_{13} and R_{14} are almost the same as in the version without rework. The events *stcounter* and *ficounter* have to be added and in state *1 stcounter* need to be disabled as well.

Although the supervisor S_2 is not the same as in Table 3.2, the reduced supervisor is the equal to Figure 3.2.

The supervisors to organize the operation sequences were changed in order to realize the rework of pieces by adding states. Only the first reduced supervisor R_{31} is shown in Figure 3.7.

The resulting reduced supervisor R_4 to chose the corresponding outcome is quite large and not very concise. Also the specification E_4, Figure 2.24 on page 28, is symmetric, R_4 is not. This result show, that not only one way to reduce a supervisor is possible. It is possible to find an reduced supervisor, which is symmetric. To test, if an automaton R is a potential reduced supervisor for a plant G, we have to test, if the language and the marked language generated $R||G$ are the same as the ones generated by the supervisor S. So it would be

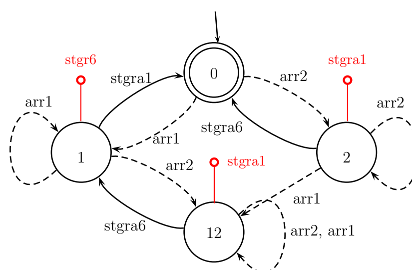
Figure 3.7: Reduced supervisor model 1 rework 1: operating sequence, R_{31}



possible to create a symmetric automaton and test, if it is a reduced supervisor for R_4. To learn more about symmetry, [12] and [11] are recommended.

R_5, the reduced local supervisor to organize how to get new pieces is presented in Figure 3.8.

Figure 3.8: Reduced supervisor model 1 rework 1: get new pieces, R_5



The specifications to realize the rework section in the second way were modeled in Subsection 2.4.4 on page 28. The resulting local supervisors and the reduced supervisors are presented below.

The resulting supervisors S_{11}, S_{12} and S_{32} as well as the corresponding reduced supervisors are equal to the automaton presented in the part without reworking the pieces.

Since to the specifications corresponding to the system without rework some events were added to E_{13} and E_{14}, the resulting supervisors are not exactly the same as shown above. However, they wont be presented here, because the reduced supervisors equal their specifications. The events *stclock*, *sttes1* and *sttes2* or *stclock*, *stgra1*, *stgra6*, *stput3*, *stput4* and *stput5*, respectively, have to be disabled in state *1*.

Although the local supervisor is different to the versions before due to a different local plant, the reduced supervisor R2 has the same structure as seen before and disables *stclock* in state *0*.

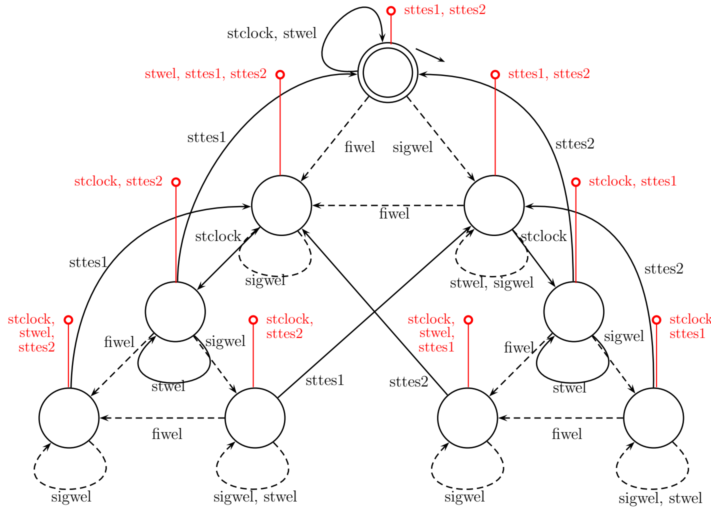
R31 is equal to Figure 3.3 even though the non reduced supervisor S_{31} cannot be the same as in the part without rework. Now let us have a look at R_{33} in Figure 3.9. Every piece of work will only be tested once. Which routine is used (*sttes1* or *sttes2*) corresponds to the origin of the piece. Standard pieces

Table 3.4: Local and reduced supervisors for model 1 with rework 2

	G_i	E_i	S_i	R_i
Mutual exclusion, 11	4	2	3	2
Mutual exclusion, 12	4	2	3	2
Mutual exclusion, 13	4	2	3	2
Mutual exclusion, 14	8	2	5	2s
Avoid rot. empty table, 2	64	2	128	2
Operating sequence, 31	16	4	56	4
Operating sequence, 32	8	4	24	4
Operating sequence, 33	8	9	60	9
Operating sequence, 34	16	4	36	5
Corresponding outcome, 41	8	7	56	7
Corresponding outcome, 42	8	7	40	7
Get new pieces, 5	4	4	16	4
Only one piece at a time, 6	8	2	12	2

have been welded before and will pass the first test routine; reworked ones will be tested by the second routine.

Figure 3.9: Reduced supervisor model 1 rework 2: operating sequence, R_{33}



The information about the used test routine then is used to chose the corresponding outcome in Figure 3.10. Standard pieces, which have been tested by using the first test routine should be put to outcome *3* or *5*. Reworked pieces will be put to outcome *3* or *4* corresponding to the test result of *sttes2*. Both R_{41} and R_{42}, which won't be presented here because it is like the corresponding specification, are symmetric.

The reduced supervisor R5 remains the same as mentioned above, realizing the rework in the first way. To make sure, that only one piece can be taken to the table before rotating the table at least once, R_6, Figure 3.11 was computed.

Figure 3.10: Reduced supervisor model 1 rework 2: corres. outcome, R_{41}

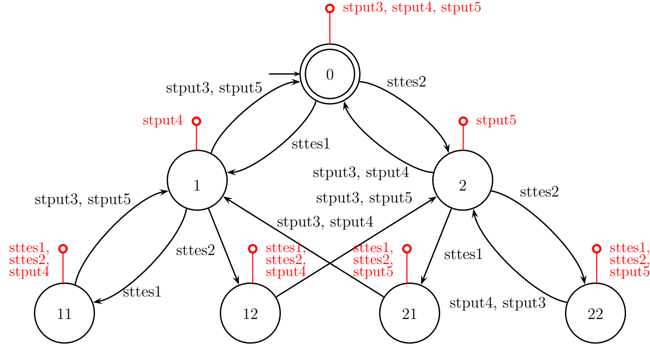
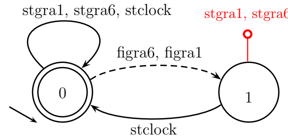


Figure 3.11: Reduced supervisor model 1 rework 2: only one piece, R_6



Model 2

All local supervisors for the second model without rework have been computed and reduced by *TCT*. The size of the local plants G, specifications E, local supervisors S and reduced supervisors R for the parts, which differ from model 1 are shown in Table 3.5.

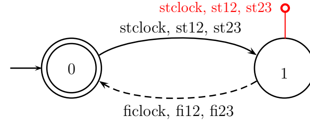
Table 3.5: Local and reduced supervisors for model 2 without rework

	G_i	E_i	S_i	R_i
Mutual exclusion, 11	4	2	3	2
Mutual exclusion, 12	4	2	3	2
Mutual exclusion, 13	4	2	3	2
Mutual exclusion, 14	20	2	18	2
Avoid rot. empty table, 2	160	2	320	2
Operating sequence, 31	40	4	152	4
Operating sequence, 32	8	4	24	4
Operating sequence, 33	8	4	24	4
Operating sequence, 34	40	4	102	5
Corresponding outcome, 4	20	15	196	15
Get new pieces, 5	10	2	20	2

As it can be seen in Figure 3.12, while the table is rotating or the robot is putting pieces to or from the table no one of these events may occur.

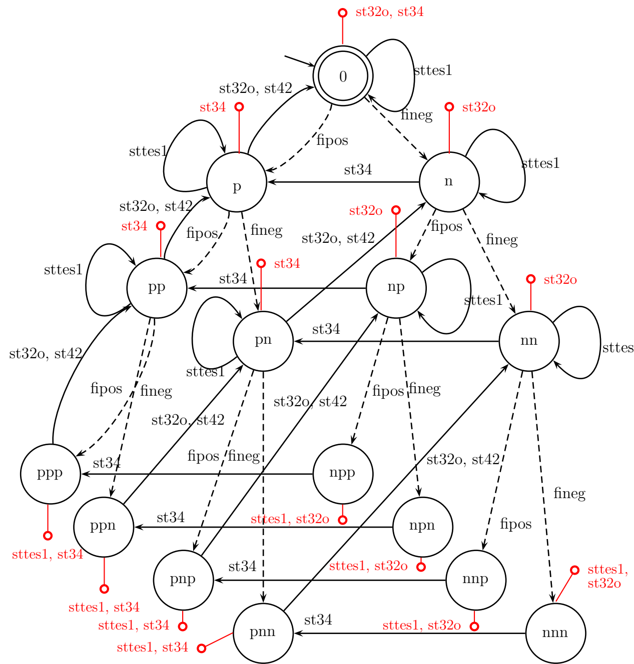
The reduced supervisors R_2, R_{31}, R_{34} and R_5 are not the same as for model 1, but they differ only in the events of the robot, corresponding to the same action of the plant. So they are not presented here.

Figure 3.12: Reduced supervisor model 2 without rework: mutual exclusion, R_{14}



Although it is fairly large, the reduced supervisor R_4, which takes care of putting the pieces to the corresponding outcome, is presented in Figure 3.13.

Figure 3.13: Reduced supervisor model 2 without rework: corres. outcome, R_4



Now let us look at the results, if we take the second model of the robot and the first way to realize the rework in Table 3.6.

As S_{11}, S_{12}, S_{13}, S_{32} and S_{33} do not depend on the robot, they are the same as mentioned above for model 1 with rework 1. The remaining supervisors are not equal but as seen before only differ in the events of the robot corresponding to the same physical action of the robot.

Before we will look at the third model, the size of all local plants G, specifications E, local supervisors S and reduced supervisors R are shown in Table 3.7. We will not present them in detail, because they are quite similar or equal to their specifications or to supervisors presented before.

Model 3

Finally we want to have a look at the resulting supervisors and reduced supervisors for the third model.

Table 3.6: Local and reduced supervisors for model 2 with rework 1

	G_i	E_i	S_i	R_i
Mutual exclusion, 11	6	2	4	2
Mutual exclusion, 12	6	2	4	2
Mutual exclusion, 13	6	2	4	2
Mutual exclusion, 14	54	2	42	2
Avoid rot. empty table, 2	96	2	768	2
Operating sequence, 31	108	6	444	5
Operating sequence, 32	12	8	40	11
Operating sequence, 33	12	8	40	10
Operating sequence, 34	108	6	266	8
Corresponding outcome, 4	36	18	278	26
Get new pieces, 5	72	4	64	4
Alternative routine, 6	24	7	80	11

Table 3.7: Local and reduced supervisors for model 2 with rework 2

	G_i	E_i	S_i	R_i
Mutual exclusion, 11	4	2	3	2
Mutual exclusion, 12	4	2	3	2
Mutual exclusion, 13	4	2	3	2
Mutual exclusion, 14	32	2	29	2
Avoid rot. empty table, 2	256	2	64	2
Operating sequence, 31	64	4	248	4
Operating sequence, 32	16	4	24	4
Operating sequence, 33	8	9	60	9
Operating sequence, 34	64	4	156	5
Corresponding outcome, 41	32	15	424	17
Corresponding outcome, 42	32	15	304	15
Get new pieces, 5	16	4	64	4
Only one piece at a time, 6	32	2	60	2

As the last specification only influences the robot, we replace the original automaton for the robot, Figure 2.41 on page 37, by the resulting supervisor S_6, Figure 3.14. To compute the missing modular supervisors S_6 will taken as the automaton for the robot.

To implement the system later on, we can use the original automaton for the robot and the corresponding reduced supervisor, shown in Figure 3.15, or only the supervisor S_6 as a part of the plant.

A overview over all local plants G, specifications E, modular supervisors S and reduced local supervisors R is presented in Table 3.8.

3.3.2 Conflicts

If we build the synchronous product of all local supervisor for the first model, we can see, that the product is equal to the monolithic supervisor which was computed before. Thus to control the plant by a monolithic or a set of local

Figure 3.14: Supervisor model 3: robot control, S_6

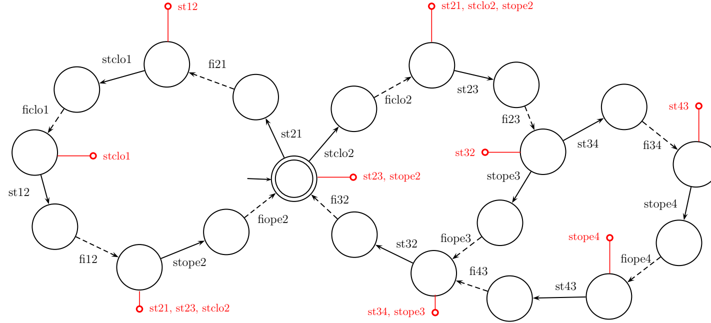


Figure 3.15: Reduced supervisor model 3: robot control, R_6

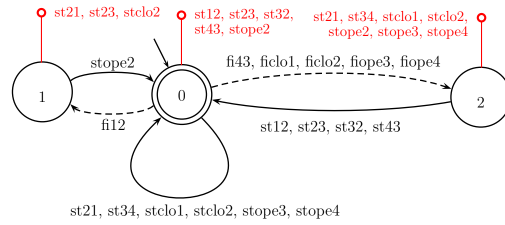


Table 3.8: Local and reduced supervisors for model 3 without rework

	G_i	E_i	S_i	R_i
Mutual exclusion, 11	4	2	3	2
Mutual exclusion, 12	4	2	3	2
Mutual exclusion, 13	4s	2	3	2
Mutual exclusion, 14	40	2	34	2
Avoid rot. empty table, 2	320	2	640	2
Operating sequence, 31	80	4	312	4
Operating sequence, 32	8	4	24	4
Operating sequence, 33	8	4	24	4
Operating sequence, 34	80	4	198	5
Corresponding outcome, 4	40	15	416	15
Get new pieces, 5	20	2	40	2
Robot control, 6	15	3	20	3

supervisors, won't influence the behavior of the system. So as local, reduced supervisors are smaller and easier to implement as a monolithic supervisor, the set of modular supervisors can be used.

Creating the synchronous product S of all local supervisors S_i for the second and third model, we will see, that the result is not equal to the monolithic supervisor S_{mon}. S is not trim; its trim part will be equal to S_{mon}. It generates a language L_S, which is greater than the language generated by S_{mon}. The marked languages are equal. This means, that the set of modular supervisors let the system run into a state, from which it is not possible to reach a marked state again, a conflict.

Detect conflicts

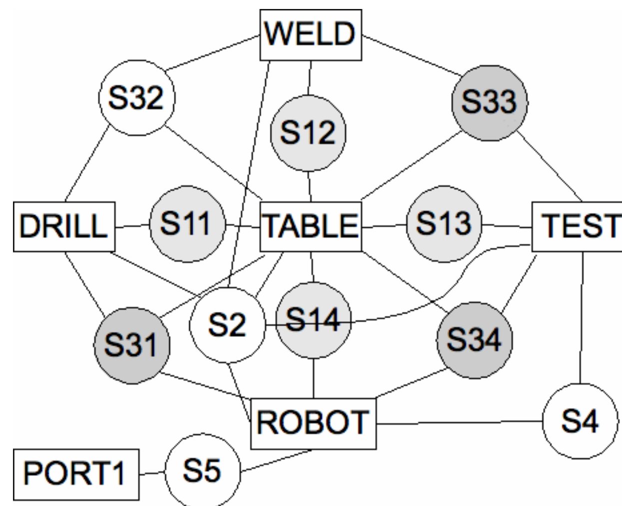
The system is conflicting, if $S_{mon} \subset ||_{i \in J} S_i$. To find out which strings lead into a conflict can be very complex. Often several subsets of two local supervisors can conflict. But on the one hand the total system can be non conflicting and on the other hand solving the conflict problems of subsets of local supervisors will not always lead to a non conflicting system.

If only a subset of local supervisors cause the overall conflicting problem, it is possible to detect them by the following procedure:

- Build the synchronous product of a subset of supervisors $S_{conf} = ||_{i \in I \subset J} S_i$.
- Build the corresponding $SupC(S_{conf})$.
- Build the synchronous product of the resulting automaton and the remaining supervisors, $S = SupC(S_{conf})||(||_{i \in J, i \notin I} S_i)$.
- If $S = S_{mon}$, the conflict can be solved by building a supervisor for S_{conf}.

The conflict in the presented example is caused by the local supervisors S_{31}, S_{33} and S_{34}. As long as the place next to the table is not occupied, the robot can rotate and get a new piece from port 1 (or port 2, if the rework section is modeled). While the robot is working but has not yet entered the security area around the table, the table can rotate. If a piece of work was situated on the test station before, the place on the table is not empty anymore and the robot cannot release the piece on the table or take the proceeded piece away, because it is not possible to release the new piece at another place. A scheme, which supervisor influences the behavior of which part of the plant for the second plant without rework, is shown in Figure 3.16. Although the conflict is only caused by S_{31}, S_{33} and S_{34}, it may help to use S_{11}, S_{12}, S_{13} and S_{14} as well, because the resulting supervisor will be smaller and easier to reduce if we build the synchronous product with S_{11}, S_{12}, S_{13} and S_{14}.

Figure 3.16: Scheme of supervisors and parts of the plant



Creating a coordinator

Some methods used in this work to find coordinators, which handle overall conflicts, are taken from [9], [14] and [27].

If a system is conflicting, it may be possible to create a coordinator, which will avoid conflicts. To understand how this supervisor is created, it is useful to have a look at the procedure to create an optimal, nonblocking supervisor in Subsection 1.3.1 on page 9 again. The coordinator will not only assure, that the controlled plant will not generate undesired strings, but also build a nonblocking supervisor.

So if we consider the synchronous product of all local supervisors S as a plant, we could create a coordinator, which avoid entering non coaccessible parts of the plant.

The trim part of S, the monolithic supervisor S_{mon}, S or an one state automaton, with a selfloop of all events appearing in the system, can be used as a specification to compute the coordinator. As the specification does not influence the language generated by the plant, the result should always be the monolithic supervisor S_{mon}.

As we have seen before, the monolithic supervisor can be very large and so it might be very helpful to reduce the supervisor before implementing it. In the presented example of the manufacturing cell this was not possible using *IDES*, *GRAIL* or *TCT*.

Heuristic methods

If it is not possible to reduce the monolithic supervisor in order to get coordinator, there are some more possibilities to obtain a non conflicting system.

It is possible to combine the specifications, whose supervisors cause the conflict and obtain a new set of less but probably larger local supervisors. By building the synchronous product of all local supervisors and testing if the result is trim, one can verify if the problem was solved.

In the presented example the resulting local supervisor for E_{31}, E_{33} and E_{34} could not been reduced and was too large to be implemented.

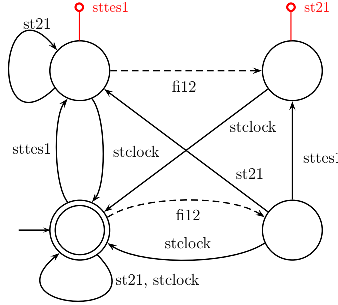
Instead of taking the synchronous product of all local supervisors as the plant to create a coordinator, it may also be possible to take only the conflicting subset. As the reduction of the coordinator was not possible using the presented tools, we built a coordinator taking only a subset of the local supervisors causing the conflicting situation as the plant.

The synchronous product of S_{31} and S_{34} would be sufficient to solve the conflict between them. As the resulting automaton is smaller by adding the supervisors S_{11}, S_{13} and S_{14}, the plant was created building the synchronous product of these five automata (S_{conf1}).

The resulting coordinator C has been reduced using *TCT*. The result for the second model without rework is presented in Figure 3.17.

As the combination of C_1 and S_{33} still can lead us into a non coaccessible state, we have to create another coordinator. But reducing the supervisor, which has been created by using the synchronous product of them as the plant, was not possible.

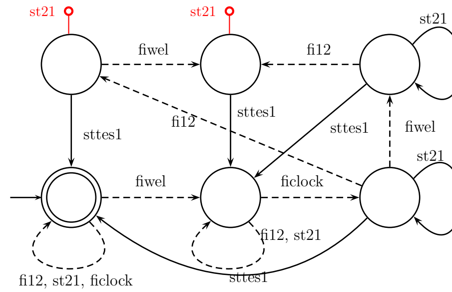
Figure 3.17: Reduced coordinator for model 2 without rework: $C_{red,1}$



In most cases the developer will have a vague idea, which parts of the plant do not provoke a conflict. To delete these events in order to obtain a smaller automaton which may be easier to reduce seems worth a try. Afterward one has to check, if the result leads to a nonblocking system.

In this work, we build the synchronous product of C_1 and S_{33} ($S_{conf2} = C_1||S_{33}$) and deleted *stdri* and *fiwel* (S'_{conf2}). The supervisor could be reduced. The reduced supervisor is presented in Figure 3.18.

Figure 3.18: Reduced coordinator for model 2 without rework: $C_{red,2}$



As you can see in Table 3.9, only one coordinator for the third model was created, because the first one already prevent the conflict.

Table 3.9: Coordinators for model 2 and 3

	model 2, wr	model 2, r1	model 2, r 2	model 3, wr
$S = \|\|_i S_i$	3318	12237	37580	5922
S_{mon}	3264	12156	37472	5760
S_{conf1}	159	295	265	309
C_1	147	277	247	267
$C_{red,1}$	4	5	4	4
S_{conf2}	564	1077	1591	-
S'_{conf2}	244	441	677	-
C_2	238	432	668	-
$C_{red,2}$	6	8	19	-

3.4 Comparison

Before we will discuss how to implement the results on a PLC in Chapter 4, we will briefly resume the obtained reduced supervisors.

First let us compare the size of all reduced local supervisors for model 1, 2 and 3 for the manufacturing cell, without the option to rework the pieces in Table 3.10.

Table 3.10: Reduced supervisors for model 1, 2 and 3 without rework

Model	1	2	3
R_{11}	2	2	2
R_{12}	2	2	2
R_{13}	2	2	2
R_{14}	2	2	2
R_2	2	2	2
R_{31}	4	4	4
R_{32}	4	4	4
R_{33}	4	4	4
R_{34}	5	5	5
R_4	7	15	15
R_5	2	2	2
R_6	-	-	3
$C_{red,1}$	-	4	4
$C_{red,2}$	-	6	-

Besides R_4 the local supervisors differ little or not at all. But model 2 and 3 need additional supervisors to control the robot and prevent conflicts. It seems to be surprising, that supervisors of nearly the same size are needed to control the system with three different models for the robot, which differ a lot in terms of potential moves of the robot. Even to control model 3 of the robot with its fairly basic movements requires a manageable set of small supervisors.

The size of all local supervisors, which are needed to control the system with the possibility to rework the pieces, are shown in Table 3.11.

It might be interesting to see, that whether handle reworked pieces in the first or the second way only make a small difference. If an interruption of the standard routine to just test the reworked pieces and take them to the corresponding outcome the required supervisors are a little bit larger than applying the second possibility (rotating the table always clockwise and ignore reworked pieces at the drilling and the welding station). Which way will be used in practice will depend on the time needed for every procedure and the percentage of reworked pieces.

Although the second model offer a faster proceeding of pieces because the table and the robot sometimes can work at the same time, the amount of the required supervisors and their size is not much higher than for the first model.

Table 3.11: Reduced supervisors for model 1 and 2 with rework

Model	1, rework 1	1, rework 2	2, rework 1	2, rework 2
R_{11}	2	2	2	2
R_{12}	2	2	2	2
R_{13}	2	2	2	2
R_{14}	2	2	2	2
R_2	2	2	2	2
R_{31}	5	4	5	4
R_{32}	11	4	11	4
R_{33}	10	9	10	9
R_{34}	8	5	8	5
R_4	14 -	7 7	26 -	17 15
R_5	4	4	4	4
R_6	9	2	11	2
R_{AB1}	-	-	5	4
R_{AB2}	-	-	8	19

CHAPTER 4

Implementation

In the last chapter we have seen the supervisors, which are needed to control the given system of a manufacturing cell. Now we want to discuss how these results can be implemented on a PLC. In this chapter we will see, how to implement a DES in general and how it was realized in detail to control the manufacturing using an Altus and a Siemens PLC.

4.1 Introduction

In Chapter 2 and Chapter 3 we have seen, how the manufacturing cell could be modeled as a set of automata and how it could be controlled by a set of supervisors. Now we want to use these results in order to control and drive the real system and test the computed supervisors.

An implementation using a microprocessor or a programmable logic controller (PLC) would be a fairly reasonable approach. In this project, we used two PLC's, one of the Brazilian company Altus, [1], and one produced by the German firm Siemens, [6].

Some general aspects should be considered independent of the used hardware. The general idea to create the source code will be very similar. For example in which sequence the different parts of the code are implemented and the structure might be analog. The hierarchical structure will be explained in Subsection 4.2.1.

PLC's may accept different programming languages. As in this project Ladder Diagram, Instruction List and Structured Text were used, we will introduce them in Subsection 4.2.4.

After talking about general information in Section 4.2, we will discuss how the system was implemented on an Altus and a Siemens PLC in detail in Section 4.3.

4.2 General information

This section is based on [17],[10] and [15].

4.2.1 Hierarchical structure

The highest hierarchical rank has the supervisor. It register the occurring controllable and non controllable events, which might change its state. According to the current state the supervisor will disable some controllable events. If more than one supervisor are implemented an event will be disabled, if it is disabled in at least one supervisor.

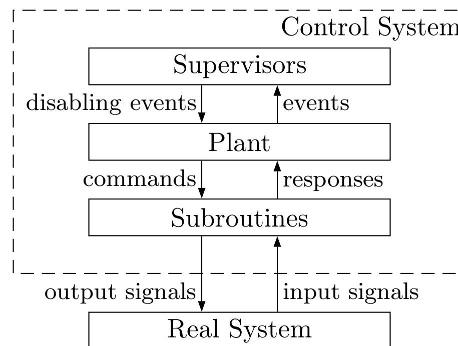
If the plant is in a state, where a controllable and not disabled event can occur, it will call the corresponding subroutine and execute the corresponding transition in the plant.

Once called the subroutine, it will send the suitable output signals and read the input signals coming from the real system and send responses to the plant.

If the plant is in a state, where a non controllable event may occur and the according response is given from a subroutines, it will execute the corresponding transition and send the information about the occurrence of this event to the supervisor.

The scheme in Figure 4.1 presenting the hierarchical structure is taken from [17].

Figure 4.1: Hierarchical structure



Respecting this structure the program used to control the plant was realized in the following way:

1. execute transitions of supervisors
2. compute disabled events
3. if an uncontrollable event occurred: execute the corresponding transition of the plant and go back to step 1
4. if not: call a controllable event, if it is possible and not disabled and go back to step 1

There are two possible ways to organize the calling of subroutines.

First the subroutines can be called by setting the corresponding bit in memory to one. Once started they operate independently and set an according bit in memory when finished the operational procedure.

Inputs are read by subroutines as well. If one detect the occurrence of an uncontrollable event, it will set the corresponding bit in memory and hence submit the information about this event to the program described above.

Another possibility would be to execute every routine, whose start bit was set once in the program, before starting the main program and execute possible transitions of supervisors again.

4.2.2 Initializing the system

Before the program can work properly the initial states of all supervisors and plants need to be set.

Furthermore one might provide an additional part of the program which set the system into the corresponding physical initial state before entering to the standard procedure.

4.2.3 Implementation of transitions

Transitions of supervisors

If a supervisor is in a certain state and an event occurred, which would change its state, we might execute the corresponding transition by setting the bit corresponding to the source state to 0 and the one corresponding to the target state to 1.

Not all transitions in an automaton representing a supervisor need to be implemented. Selfloops do not change the sate of the supervisor and so do not change the set of disabled events. The transition *arr1* from and to state *1* does not change the state of the supervisor presented in Figure 3.5 on page 46 and won't be implemented later on.

Selfloops of controllable events, however, are very important to find out which controllable events need to be disabled in which state. In every state, where no transition with an controllable event, which is part of the supervisor's event set, can be found, need to be disabled in this state. So if the occurrence of a controllable event does not change the current state but should not be disables in this situation need to be added as an selfloop to this state but won't be implemented.

Looking at Figure 3.4 on page 46 we can see, that the event *fineg* cause a transition from the initial state *0* to *n* and from *n* to *nn*. If the program executes the transition from *0* to *n* first and afterwards the one from *n* to *nn*, it might cause a problem.

After checking that the supervisor is in state *0* and detecting the event *fineg*, the transition will be executed, state *n* will be set and *0* will be reset. When checking the conditions for the execution of the second transition, they will be true, because state *n* is already set and the transition will be executed. So although the event occurred just once, it will provoke two transitions and hence probably change the behavior of the entire system.

To avoid this problem it would be possible to make sure, that all transitions always in the right order. Also check first if the current state is *n* and execute the transition to state *nn* before checking the transition from*0* to *n*.

Another way would be to use an additional variable to assure, that only one transition can be executed at a time.

Transitions of the plant

As the plant is represented by a set of automata as well, we need to implement its transitions, too.

An event just might occur, if the plant is in the according state. If the automaton in Figure 2.5 on page 19 is in state *1* the occurrence of the events *stclock*, *stcounter* and *ficounter* is not possible.

If a bit corresponding to the end of an procedure (*fi*$\cdots$) or another uncontrollable event (e.g. *arr1*) has been set by a subroutine and the plant is in a state, where a transition with this event exists, the transition will be executed, the bit corresponding to the event inside the main program will be set (to be able to change states of supervisors by running the program the next time) and reset the bit, which was set by the subroutine. If the bit set by the subroutine is not set or the plant is not in the according state, the variable corresponding to the event inside the main program will be reset.

After checking the occurrence of an uncontrollable event, now we want to execute possible controllable events. Those can occur only if the event is not disabled and the plant is in a corresponding state. In case an event is physically possible and not disabled a bit, which will call the subroutine, and the target state will be set and the source sate will be reset.

Looking at the model for the ports in Figure 2.7 on page 20, we see, that selfloops need to be implemented now as well to make it possible, that events like *arr1* or *sigdri* can occur.

The occurrence of one event can change the states of several supervisors and so change the set of disabled events, too. Thus it is important to execute all possible transitions of the supervisors and update the set of disabled events after the occurrence of every event.

Using an additional variable might solve this problem and make sure, that only one event can occur every time the main program is executed.

The idea of an uncontrollable event is, that is not possible to prevent it. So if a bit corresponding to an uncontrollable event was set, it need to be detected very fast and so we want to check the occurrence of uncontrollable events before executing the part of the main program which calls the controllable events.

4.2.4 Programming languages

According to the International Electrotechnical Commission standard IEC 61131 two graphical and two textual PLC programming language standards are defined:

- Ladder diagram (LD), graphical
- Function block diagram (FBD), graphical
- Instruction list (IL), textual
- Structured text (ST), textual

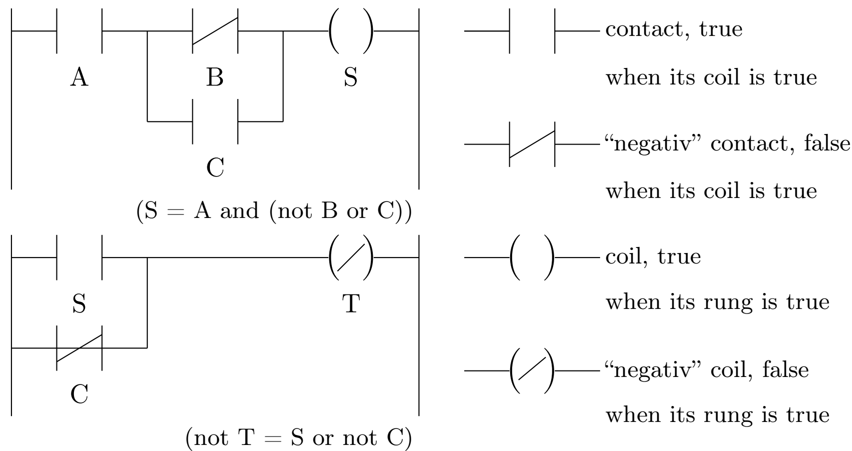
As in this project we only used Ladder diagram (subroutines), Instruction List (main program) and Structured text (main program), we want to introduce those languages briefly.

Ladder diagram

All subroutines were written by Luis Gustavo Marquez and Guilherme Siviero Lise using Ladder diagram. It is a method of drawing electrical logic schematics. It is composed of *rungs*, which consists basically of *contacts* and *coils*. Contacts are the inputs of a rung, may refer to input signals or bits in memory and can be true or false. A coil, the output of a rung, may represent and output signal or a bit in memory and depends on the contacts in the same rung.

An example is given in Figure 4.2.

Figure 4.2: Ladder diagram: an example



Instruction list

Instruction list is a low level textual language. Some of the most common commands and an example are given above.

- LD load
- N not
- A and
- O or
- S set
- R reset

Example 4.2.1 (Instrction list). The following commands would mean *reset S and set T, if A and not B.*

```
LD  A
AN  B
R   S
S   T
```

IDES can be used to create the main program automatically. If the code for the subroutines is given as IL as well, it is possible to generate not even the code for the supervisor but the whole program.

Structured text

As the Altus PLC did not accept Instruction list but ony Ladder diagram and Structured text, this programming language was used as well in this project and will be introduced briefly.

Structured text is a high level programming language. Iteration loops (REPEAT–UNTIL; WHILE–DO), conditional executions (IF–THEN–ELSE; CASE) and fundamental functions (SQRT(), SIN()) can be used.

More information is available in the book *Automating Manufacturing Systems with PLCs* by Hugh Jack, [13].

In order to automate the generation of the main program source code, the program IDES2ST was written within this project. All automata representing supervisors and parts of the plant, should be saved as automata in IDES. To learn more about IDES2ST see Chapter A.

4.3 Manufacturing cell

4.3.1 Initialization and reinitialization

To assure, that the system is physically in the initial state before entering the proper procedure, an initial procedure might run.

As we modeled the robot to be located next to the table with the open grabber, the initializing routine opens first the grabber till detect the corresponding signal. When the grabber is open, the robot arm should rotate counterclockwise towards position *0* and rotate back until arriving at position *2* next to the table.

When the main procedure is running quite a time, steps may be lost or disturbances may manipulate the actual position of the robot. Thus to reinitialize the system while processing the standard program, the robot arm coming from position *2* always rotates towards position *0* before moving to position *1* in order to get a new piece from port 1.

4.3.2 Siemens PLC

The first model without rework was implemented on a *Siemens S7 - 200* PLC. The implementation was realized by the program *Step 7 – Micro/WIN 32*, which accepts only Instruction list, Ladder diagram and Function block diagrams.

The code for the main program was generated by *IDES*. Examples are shown in Example 4.3.1, Example 4.3.2 and Example 4.3.3.

Example 4.3.1 (Implementation of a transition of a supervisor using IL). For the transition from state *0* to *1* by the event *stclock* in Figure 3.1 on page 45

IDES would generate the following code:

```
LD   SUP0S0
AN   evtblk
A    stclock
R    SUP0S0, 1
S    SUP0S1, 1
S    evtblk, 1
```

So, if the variable *SUP0S0* is true (supervisor *0* (= R11) is in state *0*), *evtblk* is false (no transition has been executed in this supervisor so far) and the event *stclock* occurs, the state *0* will be reset, state *1* set and *evtblk* will be set. *evtblk* might be reset before entering the program code of the next supervisor. To point out, that the variables *SUP0S0*, *SUP0S1* and *evtblk* are boolean *, 1* was added.

Example 4.3.2 (Implementation of a uncontrollable plant transition using IL). For the transition from state *1* to *0* by the event *fipos* in Figure 2.6 on page 19 *IDES* would generate the following code:

```
LDN  evtblk
A    PS0S1
A    Afipos
=    fipos
R    Afipos, 1
R    PS0S1, 1
S    PS0S0, 1
S    evtblk, 1
```

If the variable P0S1 is true (plant 0 (= test) is in state 1), *evtblk* is false (no plant transition has been executed so far) and the aviable *Afipos* is true (it has been set by the subroutine to announce the positiv testresult), the state *1* will be reset, state *0* set and *evtblk* will be set. The variable *fipos* will be true once to execute the corresponding transitions in the supervisors and false in the next execution of the main program. As *fipos* should occur only once, *Afipos* will be reset. *evtblk* need to be reset before entering the program code of the plant.

Example 4.3.3 (Implementation of a controllable plant transition using IL). For the transition from state *0* to *1* by the event *sttes1* in Figure 2.6 on page 19 *IDES* would generate the following code:

```
LDN  evtblk
A    PS0S0
AN   Dsttes1
=    sttes1
R    PS0S0, 1
S    PS0S1, 1
S    evtblk, 1
```

The event *sttes1* can occur, if the plant is in the corresponding state (*PS0S0* is true), the event is not disabled (*Dsttes1* is false) and no event has occurred before in this run of the main program (*evtblk* is false).

The Siemens PLC is equipped with a memory of 4 kilobyte. The implementation of the first model with the possibility to rework pieces or of the second or third models required more space in memory. Hence the Altus PLC with a memory of 512 kilobyte was used to implement the remaining models.

4.3.3 Altus PLC

Model 1 with the rework part in both ways, model 2 and model 3 have been implemented on a *PO 3147* PLC made by the Brazilian company Altus. The communication between the PLC and the PC as well as the programming of the subroutines was made with *MasterTool Extended Edition MT 8000 Advanced 5.11*.

This program accepts Ladder diagram, Function bloc diagram and Structured text. As it was not possible to find a suitable tool to manage the automatic transformation from the main program written in Instruction list by *IDES* into Structured Text, the tool *IDES2ST* was written within this project to generate automatically the code for the main program in Structured text. More information about the program and the source code is available in Chapter A on page 71.

Because the PLC is equipped with a memory of 512 kilobyte, memory limitations did not cause problems at first view. But every sequence written in Structured text need to be put in a function block, which is not allowed to be greater than 32 kilobyte. As the main program written in Structured text used more space in memory than written in Instruction list, the main programs for the second and the third model grew larger than that and needed to be cut into smaller pieces.

By setting additional auxiliary variables we assured, that every part of the main program is executed separately and in the right order.

After the implementation of the first model, all subroutines for the third model were written. As all events of the second model can be seen as sequences of events used in model 3, an interface was written in Structured text to organize the calling of the corresponding subroutines of model 3 while working with the second model.

The interface sets a bit in memory, which corresponds to the event in model 2 ($n_st\cdot$), if the event ($e_st\cdot$) was called by the main program. On the same time it starts the subroutine of the first event of model 3, which is part of model 2. When the first subroutine has finished it starts the next and so on. When the last subroutine has finished the bit in memory corresponding to the $fi\cdot$ − event is set. An example is shown in Example 4.3.4.

An additional auxiliary variable was needed to make sure, that the interface is executed right after the last part of the main program and before entering the subroutines.

Example 4.3.4 (Interface sequence).

```
(*e_st21*)
IF(e_st21) THEN
          n_st21 := 1;
          st21 := 1;
END_IF;
IF(n_st21 AND fi21) THEN
```

```
        fi21 := 0;
        stclo := 1;
END_IF;
IF(n_st21 AND ficlo) THEN
        fiope := 0;
        n_st21 := 0;
        Ae_fi21 := 1;
END_IF;
```

Conclusions

In the first chapter of this *Studienarbeit* we have seen some basic definitions and theorems about Discrete Event Systems and how they can be modeled as automata. Furthermore we introduced the Supervisory Control Theory to control systems modeled as automata and three software tools, which were used in this project.

In the following chapter we explained how the testbed of a manufacturing cell, upon which the practical part is based, operates and which specifications should be respected to assure the desired behavior. The cell can proceed pieces of work at several workstations and rework pieces, which have been disapproved the first time.

Smaller parts of the testbed have been modeled in just one way. For the robot four different models have been presented. They differ not only in terms of size but also in potential moves and restrictions. To control the system in divers ways with or without the possibility to rework pieces, different specifications for three of the four models were created.

The comparison of the models and the corresponding specifications we have seen in the last part of this chapter.

Since the creation of monolithic supervisors involve a set of problems, modular supervisors for each specifications were computed. In chapter 3 we can see, that contrary to monolithic supervisors all of them could be computed and reduced to a reasonable number of states and transitions. Although a modular approach implicated major advantages like the easier implementation and debugging process due to their size, some problems appeared.

For the second and the third model the local supervisors could lead to situations, from which it was not possible to reach a marked state in all modular supervisors without violating at least one specification. We have seen how these blocking situations can be solved and the resulting additional supervisors.

After comparing the different resulting supervisors for three models, the implementation of the system has been discussed in the fourth chapter. We have seen general issues to respect while implementing a Discrete Event System modeled as automata and controlled by supervisors on a programmable logic controller (PLC). We introduced three different programming languages, which can be used, and explained how they were used to implement the subroutines and main programs in order to control the manufacturing cell.

While modeling the system in the second chapter we could see, that great parts of the plant could be modeled in a very similar or even the same way. Most specifications presented in this chapter can be summarized in small groups of analogue or equal automata as well. That suggests to use these similarities in order to simplify the modeling process both of the plant and the specifications.

Due to the fact that once the plant and the specifications have been modeled properly the computation of the supervisors follows a determined procedure it would be interesting to automate this routine. Minimal changes in parts of the plant or on specifications sometimes asked for a total rewriting of parts of the plant or specifications and the creation of new supervisors. In some cases small errors during the manual modeling of automata or the computation of supervisors caused huge problems in the total system and were hard to find.

The template design shown in [16] might help to solve these problems and improve the procedure to model and control a Discrete Event System.

Appendix A

IDES2ST

A.1 Introduction

Within this project the program *IDES2ST* was written to generate automatically code for the main program in Structured text. The program need as in input two directories. The first should contain all supervisors or reduced supervisors saved as an *IDES*–file, the second all parts of the plant as an *IDES*–file.

The same events should have the same names in the supervisors and the parts of the plant! As event names in *IDES* may start with a number as well and all variables in Structured text must begin with a letter, *e_* is added to every event name in the generated code.

Variable names for states of supervisors and plants are given corresponding to the *IDES* – file name and the state id with an additional prefix.
(*s_ <filename>_ St<stateid>* or *p_ <filename>_ St<stateid>*)

As no information about disabled events is available in the first view, the program disables every event in a state, where the event, which is part of the supervisor automaton, build no outgoing transition from this certain state . So it is important not to delete any selfloop in an supervisor automaton.

While declaring variable names for events and states, the connection to spaces in memory is maid automatically starting with the space in memory 0, bit 0.

A.2 IDES2ST.java

```
package codeGenIdes2St;

import java.io.File;
import java.io.FileFilter;

public class Ides2St {

        public static void main(String[] args) {
                //error message if more or less then
                    folders are found
                if(args.length != 2) {
```

```
        System.out.println("Too␣few␣or␣
            too␣many␣input␣arguments!!");
        System.exit(1);
}
//error message if first folder does not
    exist
File dir1 = new File(args[0]);
if(!dir1.exists()) {
        System.out.println("The␣first␣
            input␣directory␣does␣not␣
            exist:");
        System.out.println(dir1.toString
            ());
        System.exit(0);
}
//error message if second folder does
    not exist
File dir2 = new File(args[1]);
if(!dir2.exists()) {
        System.out.println("The␣second␣
            input␣directory␣does␣not␣
            exist:");
        System.out.println(dir1.toString
            ());
        System.exit(0);
}
//initialize supervisors and plants
File[] supervisorFiles, plantFiles;
//test if files in folders are xmd files
try {
        supervisorFiles = dir1.listFiles
            (new FileFilter() {
                        public boolean
                            accept(File
                            pathname) {
                                if(
                                    pathname
                                    .
                                    isFile
                                    () &&

                                    pathname
                                    .
                                    toString
                                    ().
                                    endsWith
                                    (".
                                    xmd")
                                    )
                                    return
```

```
                                                true
                                                ;
                                          else
                                                return

                                                false
                                                ;
                                    }
                        }
            );
            plantFiles = dir2.listFiles(new
                FileFilter() {
                                    public boolean
                                        accept(File
                                        pathname) {
                                          if(
                                                pathname
                                                .
                                                isFile
                                                () &&

                                                pathname
                                                .
                                                toString
                                                ().
                                                endsWith
                                                (".
                                                xmd")
                                                )
                                                return
                                                true
                                                ;
                                          else
                                                return

                                                false
                                                ;
                                    }
                        }
            );
            //create control system
            ControlSystem cs = ControlSystem
                .fromFileLists(
                supervisorFiles, plantFiles);
            //find disabled states
            cs.findStatesWhereEventDisabled
                ();
            //generate ST code
            cs.printST();
```

```
                //security error message
                } catch (SecurityException e1) {
                        System.out.println("Security␣
                           Exception␣while␣reading␣out␣
                           directories.");
                //error message while reading the files
                } catch (Exception e2) {
                        System.out.println("Error␣while␣
                           reading␣out␣directories.");
                        e2.printStackTrace();
                }
        }
}
```

A.3 ControlSystem.java

```
package codeGenIdes2St;

import java.io.File;
import java.util.HashSet;
import java.util.Iterator;

import org.kxml2.io.KXmlParser;
import org.kxml2.kdom.Document;

class ControlSystem {
        //a control system contains
        private HashSet<Automaton> supervisors;
        private HashSet<Automaton> plants;
        private HashSet<Event> ctlEvents;
        private HashSet<Event> unCtlEvents;

        //initialize parser
        protected static KXmlParser parser;
        protected static Document doc;

        //create control system
        protected static ControlSystem fromFileLists(
           File[] supervisorFiles, File[] plantFiles) {
                //initialize parser
                ControlSystem.parser = new KXmlParser();
                //initialize control system
                ControlSystem cs = new ControlSystem();
                //control system contains supervisors
                   and plants
                cs.supervisors = new HashSet<Automaton
                   >();
                cs.plants = new HashSet<Automaton>();
                cs.ctlEvents = new HashSet<Event>();
                cs.unCtlEvents = new HashSet<Event>();
```

```
        //initialize automaton
        Automaton a;
        //first read supervisors
        for(int i=0; i<supervisorFiles.length; i
            ++) {
                File f = supervisorFiles[i];
                a = new Automaton(f, false, cs);
                cs.addSupervisor(a);
        }
        //then read plants
        for(int i=0; i<plantFiles.length; i++) {
                File f = plantFiles[i];
                a = new Automaton(f, true, cs);
                cs.addPlant(a);
        }
        return cs;
}

//add automaton to set of supervisors
protected void addSupervisor(Automaton a) {
        supervisors.add(a);
}

//add automaton to set of plants
protected void addPlant(Automaton a) {
        plants.add(a);
}

//add event to set of controllable or
    uncontrollable events
protected void addEvent(Event e) {
        if(e.isControllable()) ctlEvents.add(e);
        else unCtlEvents.add(e);
}

//look for event in set of controllable and
    uncontrollable events
protected Event getEvent(String name) {
        Iterator<Event> it = ctlEvents.iterator
            ();
        while(it.hasNext()) {
                Event e = it.next();
                if(e.getName().equals(name))
                    return e;
        }
        it = unCtlEvents.iterator();
        while(it.hasNext()) {
                Event e = it.next();
                if(e.getName().equals(name))
                    return e;
```

```
        }
        return null;
}

//find states, where event is disabled
protected void findStatesWhereEventDisabled() {
        Iterator<Event> it = ctlEvents.iterator
            ();
        while(it.hasNext()) {
                Event e = it.next();
                e.findStatesWhereEventDisabled(e
                    );
        }
}

//generate ST code
protected void printST() {
        //generate ST code
        System.out.println("PROGRAM main\n");
        //define states variables
        int b = 0;
        int m = 0;
        System.out.println("(*define all 
            variables:*)\n VAR");
        System.out.println("\tilc_inited AT %m"+
            m+"."+b+" : BOOL;");
        b+=1;
        System.out.println("\tevt_blk AT %m"+m+"
            ."+b+" : BOOL;");
        b+=1;
        System.out.println("\n\t(*define states 
            variables:*)");
        Iterator<Automaton> it = supervisors.
            iterator();
        while(it.hasNext()) {
                Automaton a = it.next();
                int x;
                x = a.printStates(b, m);
                b = x%8;
                m = (x-b)/8;
        }
        it = plants.iterator();
        while(it.hasNext()) {
                Automaton a = it.next();
                int x;
                x = a.printStates(b, m);
                b = x%8;
                m = (x-b)/8;
        }
        //define event variables
```

```
System.out.println("\n\t(*define event 
    variables:*)");
Iterator<Event> it2 = ctlEvents.iterator
    ();
while(it2.hasNext()) {
        Event e = it2.next();
        int x;
        x = e.printEvent(true, b, m);
        b = x%8;
        m = (x-b)/8;
}
it2 = unCtlEvents.iterator();
while(it2.hasNext()) {
        Event e = it2.next();
        int x;
        x = e.printEvent(false, b, m);
        b = x%8;
        m = (x-b)/8;
}
System.out.println("END_VAR\n");
//set initial states
System.out.println("(*set initial states
    *)");
System.out.println("IF (NOT ilc_inited) 
    THEN");
it = supervisors.iterator();
while(it.hasNext()) {
        Automaton a = it.next();
        a.printInit();
}
it = plants.iterator();
while(it.hasNext()) {
        Automaton a = it.next();
        a.printInit();
}
System.out.print("\tilc_inited := 1;");
System.out.print("\nEND_IF;\n");
//supervisor transitions
System.out.print("\n(*Supervisors:*)");
it = supervisors.iterator();
while(it.hasNext()) {
        Automaton a = it.next();
        a.printTransitionSub();
}
//disable events
System.out.println("\n(*Disabled events
    :*)");
it2 = ctlEvents.iterator();
while(it2.hasNext()) {
        Event e = it2.next();
```

```
                e.printDisabled();
            }
            //release events
            System.out.println("(*Release Event:*)")
                ;
            System.out.println("evt_blk := 0;");
            it = plants.iterator();
            while(it.hasNext()) {
                Automaton a = it.next();
                a.printTransitionPlantUnCtl();
            }
            it = plants.iterator();
            while(it.hasNext()) {
                Automaton a = it.next();
                a.printTransitionPlantCtl();
            }
            System.out.print("\nEND_PROGRAM");
    }
}
```

A.4 Automaton.java

```
package codeGenIdes2St;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashSet;
import java.util.Iterator;

import org.kxml2.kdom.Document;
import org.kxml2.kdom.Element;
import org.xmlpull.v1.XmlPullParserException;

class Automaton {
    //contains
    private String automatonName;
    private State iniState;
    private HashSet<State> states;
    private HashSet<Event> unCtlEvent;
    private HashSet<Event> ctlEvent;
    private HashSet<Transition> ctlTransitions;
    private HashSet<Transition> unCtlTransitions;
    private ControlSystem controlSystem;
    private boolean automatonType;

    //create automaton
    Automaton(File file, boolean type, ControlSystem
         cs) {
```

```
//initialize sets
states = new HashSet<State>();
unCtlEvent = new HashSet<Event>();
ctlEvent = new HashSet<Event>();
ctlTransitions = new HashSet<Transition
    >();
unCtlTransitions = new HashSet<
    Transition>();
controlSystem = cs;
automatonType = type;
//use filename for name of automaton
automatonName = file.getName();
automatonName = automatonName.replace(".
    xmd", "");
if(type) automatonName = "p_" +
    automatonName;
else automatonName = "s_" +
    automatonName;
//try to find file
try {
        ControlSystem.doc = parse(file);
} catch (FileNotFoundException e1) {
        System.out.println("File not
            found!");
} catch (IOException e2) {
        System.out.println("IO Error
            while trying to open file!");
} catch (XmlPullParserException e3) {
        System.out.println("Error while
            parsing xml document. is it
            well formed?");
}
//file can be read successfully
if(ControlSystem.doc != null){
        try{
                Element root =
                    ControlSystem.doc.
                    getRootElement ();
                Element data = root.
                    getElement("", "data"
                    );
                int i = 0,k;
                while((k=data.indexOf(""
                    , "state", i)) != -1)
                     {
                        //read state
                            data
                        Element s = data
                            .getElement(k
                            );
```

```
        String id = s.
            getAttributeValue
            ("", "id");
        Element prop = s
            .getElement("
            ", "
            properties");
        boolean initial
            = (prop.
            indexOf("","
            initial",0)
            != -1) ? true
             : false;
        boolean marked =
             (prop.
            indexOf("","
            marked",0) !=
             -1) ? true :
             false;
        //save state
            data
        State state =
            new State(id,

            automatonName
             + "_St" + id
            , initial,
            marked);
        addState(state);
        i=k+1;
}

i = 0;
while((k=data.indexOf(""
    , "event", i)) != -1)
     {
        //read event
            data
        Element e = data
            .getElement(k
            );
        String id = e.
            getAttributeValue
            ("", "id");
        Element prop = e
            .getElement("
            ", "
            properties");
        boolean
            observable =
```

```
    (prop.indexOf
    ("","
    observable"
    ,0) != -1) ?
    true : false;
boolean
    controllable
    = (prop.
    indexOf("","
    controllable"
    ,0) != -1) ?
    true : false;
String name = ""
    ;
if(e.indexOf("",
    "name", 0)
    != -1) name =
    e.getElement
    ("", "name").
    getText(0);
if(name.length()
    ==0) name =
    id;
name = "e_" +
    name;
// save event
    data
Event tmpEvent =
    cs.getEvent(
    name);
Event event =
    new Event(id,
    name,
    observable,
    controllable)
    ;
//if event does
    not exist yet
    in control
    system
if(tmpEvent ==
    null) {
        cs.
            addEvent
            (
            event
            );
        tmpEvent
            =
```

```
                                event
                                ;
                }
                //if event
                    already
                    exists in
                    control
                    system
                addEvent(event);
                if(automatonType
                     == false)
                    tmpEvent.
                    addSupervisor
                    (this);
                i=k+1;
}

i = 0;
while((k=data.indexOf(""
    , "transition", i))
    != -1) {
                //read
                    transition
                    data
                Element t = data
                    .getElement(k
                    );
                String id = t.
                    getAttributeValue
                    ("", "id");
                String sourceId
                    = t.
                    getAttributeValue
                    ("", "source"
                    );
                String targetId
                    = t.
                    getAttributeValue
                    ("", "target"
                    );
                String eventId =
                     t.
                    getAttributeValue
                    ("", "event")
                    ;
                //get source
                    state, target
                     state and
                    event of
```

```
                                        transition by
                                         id
                                    State source =
                                        getStateById(
                                        sourceId);
                                    State target =
                                        getStateById(
                                        targetId);
                                    Event event =
                                        getEventsById
                                        (eventId);
                                    //String
                                        eventName =
                                        event.getName
                                        ();
                                    //save
                                        transition
                                        data
                                    Transition
                                        transition =
                                        new
                                        Transition(id
                                        , source,
                                        target, event
                                        );
                                    if(event.
                                        isControllable
                                        ())
                                        addCtlTransition
                                        (transition);
                                    else
                                        addUnCtlTransition
                                        (transition);
                                    i=k+1;
                            }
                    } catch (Exception e) {
                            e.printStackTrace();
                            System.out.println("
                                Error while parsing 
                               XML document");
                    }
            }
    }

    //create document
    protected Document parse (File file) throws
        FileNotFoundException, IOException,
        XmlPullParserException {
            FileReader reader = new FileReader (file
                );
```

```
        ControlSystem.parser.setInput(reader);
        Document d = new Document();
        d.parse(ControlSystem.parser);
        return d;
}

//add initial state
protected void addState(State s) {
        states.add(s);
        if(s.isInitial()) iniState = s;
}

//add event to set of controllable or
   uncontrollable events
protected void addEvent(Event e) {
        if(e.isControllable()) ctlEvent.add(e);
        else unCtlEvent.add(e);
}

//add controllable transition
protected void addCtlTransition(Transition t){
        ctlTransitions.add(t);
}

//add uncontrollable transition
protected void addUnCtlTransition(Transition t){
        unCtlTransitions.add(t);
}

//get state by id
protected State getStateById(String id) {
        Iterator<State> it = states.iterator();
        while(it.hasNext()) {
                State s = it.next();
                if(s.getId().equals(id)) return
                    s;
        }
        return null;
}

//get event by id
protected Event getEventsById(String id) {
        //first uncontrollable events
        Iterator<Event> it = unCtlEvent.iterator
            ();
        while(it.hasNext()) {
                Event e = it.next();
                if(e.getId().equals(id)) return
                    e;
        }
```

```
            //then controllable events
            it = ctlEvent.iterator();
            while(it.hasNext()) {
                    Event e = it.next();
                    if(e.getId().equals(id)) return
                        e;
            }
            return null;
    }

    //get name of automaton
    protected String getName(){
            return automatonName;
    }

    //find disabled states
    protected void findStatesWhereEventDisabled(
        Event event) {
            Iterator<State> it = states.iterator();
            while(it.hasNext()) {
                    State s = it.next();
                    if(s.isDisabled(event)) {event.
                        addStateWhereDisabled(s);}
            }
    }

    //variable declaration states
    protected int printStates(int b, int m) {
            Iterator<State> it1 = states.iterator();
            while(it1.hasNext()) {
                    State s = it1.next();
                    s.printStates(b, m);
                    b +=1;
                    if(b%8 == 0){
                            b = 0;
                            m +=1;
                    }
            }
            return m*8+b;
    }

    //variable declaration events
    protected int printEvents(boolean type, int b,
        int m){
            if(type){
                    //uncontrollable events
                    Iterator<Event> it3 = ctlEvent.
                        iterator();
                    while(it3.hasNext()) {
                            Event e = it3.next();
```

```
                                int x;
                                x = e.printEvent(true, b
                                    , m);
                                b = x%8;
                                m = (x-b)/8;
                        }
                        //controllable events
                        Iterator<Event> it4 = unCtlEvent
                            .iterator();
                        while(it4.hasNext()) {
                                Event e = it4.next();
                                int x;
                                x = e.printEvent(false,
                                    b, m);
                                b = x%8;
                                m = (x-b)/8;
                        }
                }
                return m*8+b;
        }

        //set initial states
        protected void printInit(){
                Iterator<State> it1 = states.iterator();
                while(it1.hasNext()) {
                        State s = it1.next();
                        s.printInit();
                }
        }
        //print transition of supervisors
        protected void printTransitionSub(){
                System.out.println("\n(*"+automatonName+
                    ":*)");
                System.out.println("evt_blk := 0;");
                Iterator<Transition> it =
                    unCtlTransitions.iterator();
                while(it.hasNext()) {
                        Transition t = it.next();
                        t.printTransitionSub();
                }
                it = ctlTransitions.iterator();
                while(it.hasNext()) {
                        Transition t = it.next();
                        t.printTransitionSub();
                }
        }

        //print uncontrollable transitions of all plants
        protected void printTransitionPlantUnCtl(){
```

```
			Iterator<Transition> it =
			    unCtlTransitions.iterator();
			System.out.println("(*uncontrollable 
			    events of  "+automatonName+":*)");
			while(it.hasNext()) {
				Transition t = it.next();
				t.printTransitionPlantUnCtl();
			}
		}

		//print controllable transitions of all plants
		protected void printTransitionPlantCtl(){
			Iterator<Transition> it = ctlTransitions
			    .iterator();
			System.out.println("(*controllable 
			    events of "+automatonName+":*)");
			while(it.hasNext()) {
				Transition t = it.next();
				t.printTransitionPlantCtl();
			}
		}
}
```

A.5 State.java

```
package codeGenIdes2St;

import java.util.HashSet;
import java.util.Iterator;


class State {
		//a state contains
		private boolean initial;
		private boolean marked;
		private String id;
		private String name;
		private HashSet<Transition> outgoingCtl;
		private HashSet<Transition> outgoingUnCtl;

		//create state
		State(String id, String name, boolean initial,
		    boolean marked) {
			//state contains
			outgoingCtl = new HashSet<Transition>();
			outgoingUnCtl = new HashSet<Transition
			    >();
			this.initial = initial;
			this.marked = marked;
			this.id = id;
```

```
        this.name = name;
}

//test if state is initial state
protected boolean isInitial() {
        return initial;
}

//test if e is disabled in state
protected boolean isDisabled(Event e) {
        Iterator<Transition> it = outgoingCtl.
            iterator();
        while(it.hasNext()) {
                Transition t = it.next();
                String refEvent = t.getEventName
                    ();
                if(refEvent.equals(e.getName()))
                    return false;
        }
        return true;
}

//add a transition to set of outgoing
    controllable transitions
protected void addOutgoingCtlTrans(Transition t)
     {
        outgoingCtl.add(t);
}

//add a transition to set of outgoing
    uncontrollable transitions
protected void addOutgoingUnCtlTrans(Transition
    t) {
        outgoingUnCtl.add(t);
}

//return state id
protected String getId() {
        return id;
}

//return state name
protected String getName() {
        return name;
}

//variable declaration states
protected void printStates(int b, int m) {
        System.out.println("\t"+name+" AT %m"+m+
            "."+b+" : BOOL;");
```

```
        }

        //set initial state
        protected void printInit(){
                if(initial)System.out.println("\t"+name+
                    " := 1;");
        }
}
```

A.6 Event.java

```
package codeGenIdes2St;

import java.util.HashSet;
import java.util.Iterator;

class Event {
        //contains
        private String id;
        private String name;
        private boolean observable;
        private boolean controllable;
        private HashSet<State> statesWhereDisabled;
        private HashSet<Automaton> supervisors;

        //create event
        Event (String id, String name, boolean
           observable, boolean controllable) {
                this.id = id;
                this.name = name;
                statesWhereDisabled = new HashSet<State
                    >();
                supervisors = new HashSet<Automaton>();
                this.observable = observable;
                this.controllable = controllable;
        }

        //test if event is controllable
        protected boolean isControllable() {
                return controllable;
        }

        //get event id
        protected String getId() {
                return id;
        }

        //get event name
        protected String getName() {
                return name;
```

```
	}

	//
	protected void addSupervisor(Automaton
	    supervisor){
		supervisors.add(supervisor);
	}

	protected void findStatesWhereEventDisabled(
	    Event event){
		Iterator<Automaton> it = supervisors.
		    iterator();
		while(it.hasNext()) {
			Automaton a = it.next();
			a.findStatesWhereEventDisabled(
			    event);
		}

	}
	//create set of states, where event is disabled
	protected void addStateWhereDisabled(State s) {
		statesWhereDisabled.add(s);
	}

	//test if two events are equal
	protected boolean equals(Event e) {
		if(id.equals(e.getId())) return true;
		else return false;
	}

	//variable declaration events
	protected int printEvent(boolean controllable,
	    int b, int m) {
		if(controllable){
			System.out.print("\t"+name+"␣AT␣
			    %m␣:␣BOOL;\n");
			System.out.println("\tD"+name+"␣
			    AT␣%m"+m+"."+b+"␣:␣BOOL;\n");
			b +=1;
			if(b%8 == 0){
				b = 0;
				m +=1;
			}
		}
		else {
			System.out.print("\t"+name+"␣AT␣
			    %m"+m+"."+b+"␣:␣BOOL;\n");
			b +=1;
			if(b%8 == 0){
				b = 0;
```

```
                    m +=1;
                }
                System.out.println("\tA"+name+" 
                    AT %m : BOOL;\n");
            }
            return m*8+b;
        }

        //print disabled events
        protected void printDisabled(){
            System.out.println("(*"+name+":*)");
            boolean firstElement=true;
            System.out.print("IF(");
            Iterator<State> it = statesWhereDisabled
                .iterator();
            while(it.hasNext()) {
                State s = it.next();
                if(firstElement)firstElement=
                    false;
                else System.out.print(" OR ");
                System.out.print(s.getName());
            }
            System.out.println(") THEN");
            System.out.println("\tD"+name+" := 1;");
            System.out.println("ELSE");
            System.out.println("\tD"+name+" := 0;");
            System.out.println("END_IF;\n");
        }
}
```

A.7 Transition.java

```
package codeGenIdes2St;

import java.util.Iterator;

class Transition {
        //contains
        private String id;
        private State sourceState;
        private State targetState;
        //private String eventName;
        private Event event;
        //private ControlSystem cs;

        //create transition
        Transition(String id, State source, State target
            , Event event) {
                this.id = id;
                this.sourceState = source;
```

```
		this.targetState = target;
		//this.eventName = eventName;
		this.event = event;

		//add to set of outgoing transitions of
			source state
		if(event.isControllable()) sourceState.
			addOutgoingCtlTrans(this);
		else sourceState.addOutgoingUnCtlTrans(
			this);
	}

	//get event
	protected String getEventName() {
		String eventName = event.getName();
		return eventName;
	}

	//get event name
	protected Event getEvent(){
		return event;
	}

	//get transition id
	protected String getId() {
		return id;
	}

	//get source state
	protected State getSourceState() {
		return sourceState;
	}

	//get target state
	protected State getTargetState() {
		return targetState;
	}

	//print transitions of supervisors
	protected void printTransitionSub(){
		if(sourceState.getId() != targetState.
			getId()){
			System.out.println("IF (NOT 
				evt_blk AND "+sourceState.
				getName()+" AND "+event.
				getName()+") THEN");
			System.out.println("\tevt_blk :=
				 1;");
```

```
                        System.out.println("\t"+
                            sourceState.getName()+" := 0;
                            ");
                        System.out.println("\t"+
                            targetState.getName()+" := 1;
                            ");
                        System.out.println("END_IF;\n");
                }
        }

        //print uncontrollable transitions of all plants
        protected void printTransitionPlantUnCtl(){
                //if(sourceState.getId() != targetState.
                    getId()){
                if(true){
                        System.out.println("IF (NOT
                            evt_blk AND " +sourceState.
                            getName()+" AND A"+event.
                            getName()+") THEN");
                        System.out.println("\t"+event.
                            getName()+" := 1;");
                        System.out.println("\tA"+event.
                            getName()+" := 0;");
                        System.out.println("\tevt_blk :=
                             1;");
                        System.out.println("\t"+
                            sourceState.getName()+" := 0;
                            ");
                        System.out.println("\t"+
                            targetState.getName()+" := 1;
                            ");
                        System.out.println("ELSE");
                        System.out.println("\t"+event.
                            getName()+" := 0;");
                        System.out.println("END_IF;\n");
                }
        }

        //print controllable transitions of all plants
        protected void printTransitionPlantCtl(){
                //if(sourceState.getId() != targetState.
                    getId()){
                if(true){
                        System.out.println("IF (NOT
                            evt_blk AND " +sourceState.
                            getName()+" AND NOT D"+event.
                            getName()+") THEN");
                        System.out.println("\t"+event.
                            getName()+" := 1;");
```

```
                                    System.out.println("\tevt_blk␣:=
                                        ␣1;");
                                    System.out.println("\t"+
                                        sourceState.getName()+"␣:=␣0;
                                        ");
                                    System.out.println("\t"+
                                        targetState.getName()+"␣:=␣1;
                                        ");
                                    System.out.println("ELSE");
                                    System.out.println("\t"+event.
                                        getName()+"␣:=␣0;");
                                    System.out.println("END_IF;\n");
                        }
            }
}
```

Bibliography

[1] Altus webpage.

[2] CEFETS' s webpage.

[3] http://www.csd.uwo.ca/research/grail/.

[4] IDES lab webpage, Queen' s University.

[5] José E. R. Cury' s webpage.

[6] Siemens webpage.

[7] W. M. Wonham' s webpage.

[8] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 1999.

[9] Chen, Y.-L., S. Lafortune and F. Lin. Modular Supervisory Control with Priorities for Discrete Event Systems. In *Proceedings of the 34th Conference on Decision and Control*, pages 409 – 415, December 1995.

[10] 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*, Tampa, Florida, USA, December 1998.

[11] José M. E. González. *Aspectos de síntese de supervisores para sistemas a eventos discretos e sistemas híbridos*. PhD thesis, Universidade Federal de Sante Catarina, April 2000.

[12] José M. E. González and José E.R. Cury. Exploiting Symmetry in the Synthesis of Supervisors for Discrete Event Systems. *IEEE Transactions on automatic control*, 46(9):1500–1505, December 2001.

[13] Hugh Jack. *Automating Manufacturing Systems with PLCs*. 2007.

[14] K. C. Wong, J. G. Thistle, H.-H. Hoang and R. P. Malhamé. Conflict resolution in modular control with applications to feature interaction. In *Proceedings of the 34th Conference on Decision and Control*, pages 416 – 421, December 1995.

[15] Ryan James Leduc. PLC implementation of a DES supervisor for a manufacturing testbed: An implemention perspective. Master's thesis, University of Toronto, 1996.

[16] Lenko Grigorov. Template Design of Discrete-Event Systems. Technical report 2007-538, School of Computing, Queen's University, Canada, 2007.

[17] Max H. de Queiroz, José E. R. Cury. Synthesis and implementation of local modular supervisory control for a manufacturing cell. In *WODES 2002*, 2002.

[18] Rajinderjeet Singh Minhas. Complexity reduction in Discrete Event Systems. Master's thesis, University of Toronto, 2002.

[19] Institute of Electrical and Electronics Engineers. IEEE Standard Dictionary of Electrical and Electronic Terms.

[20] Carl Adam Petri. Kommunikation mit Automaten. *Schriften des Rheinisch-Westfälischen Institutes für instrumentelle Mathematik an der Universität Bonn*, 1962.

[21] P.J. Ramadge and W.M. Wonham. Supervision of Discrete Event Processes. In *Proceedings of the 21th IEEE Conference on Decision and Control*, pages 1228–1229, 1982.

[22] Darrell R. Raymond and Derick Wood. Grail: A C++ Library for Automata and Expressions. *Journal of Symbolic Computation*, 17(4):341–350, 1994.

[23] Christianne Reiser. O Ambiente GRAIL para Controle Supervisório de Sistemas a Eventos Discretos: Reestruturação e Implementação de Novos Algoritmos. Master's thesis, Universidade Federal de Sante Catarina, 2005.

[24] R. Su and W. M. Wonham. Supervisor Reduction for Discrete-Event Systems. *Discrete Event Dynamic Systems*, 14(1):31–53, 2004.

[25] A. F. Vaz and W. M. Wonham. On supervisor reduction in discrete-event systems. *International Journal of Control*, 44(2):475–491, 1986.

[26] Sarah-Jane Whittaker. Does size matter? The effects of supervisor reduction on minimal communication between distributed discrete-event agents. Master's thesis, Queen's University, 2005.

[27] K. C. Wong and W. M. Wonham. Modular Control and Coordination of Discrete-Event Systems. *Discrete Event Dynamic Systems*, 8(3):241 – 273, 1998.

[28] W.M. Wonham. *Notes on Control of Discrete Event Systems*. Dept. of Elec. and Comp. Eng., 2001.