

MASTER

Supervisory control of thema park vehicles

Forschelen, Stefan T.J.

Award date:
2010

[Link to publication](#)

Disclaimer

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Supervisory control of theme park vehicles

S.T.J. Forschelen

SE 420592

Master's Thesis

Supervisor: Prof.dr.ir. J.E. Rooda

Coaches: Ing. T. van Els (NBG)

Dr.ir. J.M. van de Mortel - Fronczak

EINDHOVEN UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF MECHANICAL ENGINEERING
SYSTEMS ENGINEERING GROUP

March 4, 2010

FINAL ASSIGNMENT

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mechanical Engineering
Systems Engineering Group

May 2009

Student	S.T.J. Forschelen
Supervisor	Prof.dr.ir. J.E. Rooda
Advisors	Ing. T. van Els (NBG) Dr.ir. J.M. van de Mortel-Fronczak
Start	March 2009
Finish	March 2010
<u>Title</u>	Supervisory Control of theme park vehicles

Subject

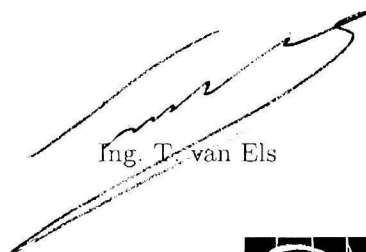
In order to keep customers interested in revisiting, theme parks have to renew their attractions from time to time. This means that there is a demand for new, innovative and spectacular rides. A new development in this area is an attraction with Multi Movers: automated guided vehicles that interact with each other, their environment and a central control unit. Because of their interactivity and the fact that these vehicles have to be very safe for their passengers, the prime design focus is on collision avoidance. Moreover, it is desirable to achieve a high passenger throughput for the whole system.


In the framework of supervisory control theory (SCT), a method is defined for the synthesis of supervisory control systems according to specifications of system components and requirements represented by automata. The supervisor synthesized is by construction guaranteed to be non-blocking. Whenever the system components or requirements change, a new supervisor can be synthesized using adapted specifications. Additionally, the use of models enables the application of model-based techniques and tools for thorough system analysis and systematic testing, which help to improve the system overview for the engineers.

Assignment

The purpose of this project is to evaluate how the model-based engineering paradigm in combination with supervisory control synthesis can contribute to the product development within NBG. To this end, the supervisor design for the Multi Mover is chosen as the case study. To implement the model-based engineering concept, a set up should be proposed that allows for application of model-based techniques.


Prof.dr.ir. J.E. Rooda


Ing. T. van Els


Dr.ir. J.M. van de Mortel-Fronczak

Systems
Engineering



Department of Mechanical Engineering

Preface

The thesis you are reading at the moment is a result of a full year of research to obtain my master's degree in Mechanical Engineering. I have attended the Eindhoven University of Technology almost seven years, and I have learned a lot during these years. One of the highlights was my membership of the board of W.S.V. Simon Stevin, which I joined after my second year in Eindhoven. A full year of organizing activities and making friends was a valuable experience.

Subsequently, I decided to do my bachelor final project at the Systems Engineering Group. My coaches Joost van Eekelen and Erjen Lefeber made me enthusiastic about the research that is performed at the Systems Engineering Group. As a result, it was not a very difficult choice to join the Systems Engineering Group in order to obtain my master's degree. The elective courses were very interesting, and my internship at the University of Auckland in New Zealand was a wonderful experience.

The result of this project was not possible without the help of many people. First of all, I want to thank Tjeu Naus and professor Rooda for giving me the opportunity to work on this interesting subject of supervisory control at NBG Industrial Automation. NBG has offered me a very positive working environment with very friendly colleagues. Then, I want to thank professor Rooda for his intensive supervision of this project. He taught me valuable lessons about facing challenges in life and I am confident that this project was not such a success without his help. Furthermore, the constructive feedback of Tim van Els and Asia van de Mortel was of great support. I really appreciated that the door was always open when I had considerations about this project, which I worked on with pleasure. Next, Albert Hofkamp and Ivo Sieben helped me a lot with coding issues and I am grateful for all their help. Lastly, I want to thank Mieke Lousberg for all the practical help during my time at the Systems Engineering Group.

Obtaining my master's degree was impossible without the support of my family. Therefore, I want to thank my parents for giving me an academic education. Finally, my special thanks go to Lieke, for always being my support in life.

Stefan Forschelen

Eindhoven, February 2010

Summary

High-tech companies are often challenged to increase the functionality and quality of a product, while at the same time they try to reduce the time-to-market and product costs. However, current practice shows that this is not straightforward. As a result, there is a need for new engineering processes. In this project, we concentrate on the engineering process of the part of the control system that is responsible for the coordination of components.

Supervisory control theory uses formal models of the uncontrolled system and its control requirements to synthesize a so-called supervisor. This supervisor is mathematically correct with respect to the formal models of the uncontrolled system and the control requirements. This shifts the validation process from debugging controller code to debugging the models of the uncontrolled system and the control requirements. Furthermore, the use of formal models enables the application of model-based techniques and tools for thorough system analysis and systematic testing, which help to improve the system overview for the engineers. As a result, supervisory control theory is expected to enhance the product development process, since this theory allows us to synthesize the part of control software that is responsible for coordinating components. The purpose of this project is to investigate how supervisory control synthesis can contribute to the product development. The case study that is used in this project is a theme park vehicle that follows a wire integrated in the floor referred to as the multimover. This vehicle concept offers the possibility for new ride concepts with crossings, switches, junctions and driving into and out of dead-end tracks. Supervisory control theory is used in this project to synthesize a supervisor that ensures safety, which includes coordination of different components, such as anticipating on emergency and error signals and an accurate proximity handling.

Within supervisory control theory, two frameworks are distinguished that are used widely for synthesizing supervisors, namely the event-based framework of Ramadge and Wonham [Won84, Ram87] and the state-based framework of Ma and Wonham [Ma05]. In this project, both frameworks are used to synthesize a supervisor for the multimover. Furthermore, modelling and synthesis aspects of both frameworks are compared with each other. On the one hand, the event-based framework can be extended with distributed or hierarchical supervision. On the other hand, the state-based framework is more convenient for modelling the control requirements, since it allows logical expressions. However, only centralized supervisors can be synthesized using this framework. In this project, supervisors are synthesized using both frameworks. To have advantages of both frameworks, an automatic conversion of logical expressions to finite state machines is proposed. Due to this conversion, control requirements can be formulated as automata and logical expressions and still use the event-based framework for synthesizing a distributed or a hierarchical supervisor. The synthesized supervisors are implemented in the current control software environment of theme park vehicles. The implementation set-up is developed in such a way, that supervisors synthesized using either one of both frameworks can easily be embedded. This implementation is validated by means of performing several test cases on a test set-up. Implementation testing showed the same behaviour as the simulations, and the conclusion is drawn that safety is assured satisfactorily by the synthesized supervisors.

The integration of supervisory control synthesis in the engineering process makes the control software more evolvable. If the system or its requirements change, only a couple of models have to be adapted and a new supervisor is synthesized. Furthermore, since the intended behaviour is specified in a modelling language instead of a software programming language, engineers can have a better understanding of the control soft-

ware, which can lead to an easier validation. In addition, the use of formal models enables the application of model-based techniques, such as discrete-event simulation and hybrid simulation, which allows to detect design errors in an early stage of the product development process. As a result, less prototypes might possibly be developed. The conclusion is drawn that supervisory control synthesis has the potential to enhance the product development process and is suitable for engineering controllers that coordinate components. However, the absence of livelock in the implementation is currently not guaranteed. This could possibly be eliminated by the synthesis of supervisors which are optimal in terms of timed behaviour.

Samenvatting

High-tech bedrijven worden vaak uitgedaagd om de functionaliteit en de kwaliteit van een product te verhogen, terwijl tegelijkertijd de kosten en de marktintroductietijd van een product moeten worden verlaagd. In de huidige praktijk blijkt echter dat dit niet eenvoudig is. Als gevolg hiervan is er een vraag naar nieuwe productontwikkelprocessen. In dit project concentreren we ons op het ontwikkelproces van een gedeelte van het besturingssoftware, dat verantwoordelijk is voor het coördineren van systeemcomponenten.

Supervisory control theory gebruikt modellen van het ongecontroleerde systeem en de eisen waaraan het systeem moet voldoen om een zogenaamde supervisor te synthetiseren. Deze supervisor is wiskundig correct ten opzichte van de formele modellen van het ongecontroleerde systeem en de eisen waaraan dit systeem moet voldoen. Dit verschuift het validatieproces van het debuggen van softwarecode naar het debuggen van de modellen van het ongecontroleerde systeem en zijn eisen. Verder maakt het gebruik van modellen het mogelijk om modelgebaseerde technieken en tools te gebruiken voor een uitgebreide systeem-analyse en systematische tests, wat bijdraagt aan een beter systeemoverzicht voor de ingenieurs. Verwacht wordt dat supervisory control theory het productontwikkelproces kan verbeteren, omdat deze theorie het mogelijk maakt om het gedeelte van de besturingssoftware te synthetiseren dat verantwoordelijk is van het coördineren van componenten. Het doel van dit project is om te onderzoeken hoe supervisory control synthese kan bijdragen aan de productontwikkeling. De case dat in dit project is gebruikt is een pretparkvoertuig, dat een draad kan volgen die in de vloer is aangebracht. Dit voertuigconcept maakt het mogelijk om over kruisingen, wissels en splitsingen te rijden. Dit pretparkvoertuig wordt een multimover genoemd. In dit project gaat het om het synthetiseren van een supervisor die de veiligheid van de multimover waarborgt, wat neerkomt op het anticiperen op foutsignalen en een accurate nabijheidsafhandeling.

Twee raamwerken kunnen worden herkend in supervisory control theory die vaak worden gebruikt voor het synthetiseren van supervisors, namelijk het event-based raamwerk van Ramadge en Wonham [Won84, Ram87] en het state-based raamwerk van Ma en Wonham [Ma05]. Beide raamwerken zijn in dit project gebruikt om een supervisor voor de multimover te synthetiseren. De modelleer- en syntheseaspecten van beide raamwerken zijn met elkaar vergeleken. Aan de ene kant kan het event-based raamwerk worden uitgebreid om naast gecentraliseerde supervisors ook gedistribueerde en hiërarchische supervisors te synthetiseren. Aan de andere kant biedt het state-based raamwerk meer modelleergemak, omdat de eisen niet alleen met eindige toestandsmachines kunnen worden gemodelleerd, maar ook met logische expressies. Echter, alleen gecentraliseerde supervisors kunnen met het state-based raamwerk worden gesynthetiseerd. Om de voordelen van beide raamwerken te kunnen benutten, is een automatische conversie voorgesteld die simpele logische expressies converteert naar eindige toestandsmachines. Middels deze conversie is het mogelijk om de eisen met logische expressies te modelleren, en toch een gedistribueerde of hiërarchische supervisor te synthetiseren. De supervisors zijn geïmplementeerd in de bestaande softwareomgeving van de multimover. Het implementatieprototype is zo gemaakt dat supervisors, gesynthetiseerd met een van beide raamwerken, makkelijk kunnen worden geïmplementeerd. De implementatie is gevalideerd door verschillende testcases uit te voeren op een testopstelling. De tests vertoonden hetzelfde gedrag als de simulaties en de conclusie is getrokken dat de gesynthetiseerde supervisors kunnen worden gebruikt om de veiligheid van de multimover te waarborgen.

De integratie van supervisory control synthese in het productontwikkelproces maakt de besturingssoftware meer flexibel. Als het systeem of de eisen zijn veranderd, hoeven er enkel een paar modellen te worden

aangepast. Een nieuwe supervisor kan dan meteen worden gesynthetiseerd. Aangezien het beoogde gedrag is gespecificeerd in een modelleertaal in plaats van in een programmeertaal, kunnen ingenieurs de besturingssoftware beter begrijpen, wat tot een makkelijkere validatie kan leiden. Het gebruik van formele modellen maakt het mogelijk om modelgebaseerde technieken toe te passen, zoals discrete simulatie en hybride simulatie. Dit kan fouten in een vroeg stadium van het productontwikkelp proces helpen detecteren en de ontwikkelkosten van prototypes mogelijk verlagen. Supervisory control synthese zou het productontwikkelp proces kunnen verbeteren en is geschikt voor het ontwikkelen van besturingssoftware dat verantwoordelijk is voor het coördineren van systeemcomponenten. De afwezigheid van livelock in de implementatie kan echter niet worden gegarandeerd. Dit zou mogelijk kunnen worden geëlimineerd door synthese van supervisors die optimaal zijn qua tijdgedrag.

List of symbols

Table 1: List of symbols.

Symbol	Description
D	Design
$\rightarrow \{ E \}$	Event set predicate
G	Automaton representing a plant
$G / \approx_{\Sigma'}$	Abstraction of automaton G
H	Automaton representing a requirement
$L(G)$	Language represented by automaton G
$\overline{L(G)}$	Prefix closure of $L(G)$
$L_m(G)$	Marked language represented by automaton G
M_P	Model of uncontrolled plant
M_S	Model of the supervisor
M_{R_S}	Model of the requirements of the supervisor
R	Requirements
S	Automaton representing a supervisor
\mathbf{X}	Set of states
\mathbf{X}_m	Set of marker states
\mathbf{x}	State
\mathbf{x}_0	Initial state
\mathbf{x}_m	Marker state
$\mathbf{x} \downarrow$	State predicate
Z	Realization
ε	Empty string
Σ	Alphabet, set of events
Σ'	Abstraction alphabet
Σ^*	Kleene closure of Σ
Σ_c	Set of controllable events
Σ_o	Set of observable events
Σ_u	Set of uncontrollable events
Σ_{uo}	Set of unobservable events
ξ	Transition map
σ	Event
s	Sequence of events

Contents

Assignment	i
Preface	iii
Summary	v
Samenvatting	vii
List of symbols	ix
1 Introduction	1
1.1 Traditional engineering	2
1.2 Model-based engineering	3
1.3 Objectives	4
1.4 Outline	5
2 Supervisory control theory	7
2.1 Automata and languages	8
2.2 Language properties	10
2.3 Event-based supervisory control	12
2.4 State-based supervisory control	18
2.5 Synthesis-based engineering	22
2.6 Toolchain	23
3 Case study: the multimover	25
3.1 Functionality	25
3.2 Plant models	27
3.3 Modules	32
3.4 Requirement models	33
3.5 Supervisor synthesis	35
3.6 Supervisor validation	36

4 Frameworks for supervisor synthesis	39
4.1 Synthesis aspects	39
4.2 Modelling aspects	40
4.3 Conversion of state-based expressions to automata	41
4.4 Experiment	46
4.5 Discussion	47
5 Implementation	49
5.1 From supervisor to controller	49
5.2 Implementation aspects	51
5.3 Prototype implementation	54
5.4 Validation of implementation	57
5.5 Experiment	57
5.6 Synthesis-based engineering: evaluation	58
6 Conclusions and suggestions for further research	63
6.1 Conclusions	63
6.2 Suggestions for further research	65
Bibliography	67
A Formal models	71
A.1 Plant models	71
A.2 Requirement models	74
A.3 Event list	79
A.4 State list	80
B Logic expression converter	83
B.1 Source code	83
C Supervisor synthesis	89
C.1 Event-based supervisor synthesis	89
C.2 State-based supervisor synthesis	93
D Implementation	99
D.1 Communication delay example	99
D.2 Explanation lookup tables	100
D.3 CIF to C conversion	100

Chapter 1

Introduction

Development of high-tech systems is a challenging task, since functionality is often increasing due to more demanding markets and increased competition. As a result, the design of a high-tech system is becoming more and more complex. Additionally, more complex systems tend to have a longer development cycle and as a consequence, it is difficult for industrial companies to remain competitive in ever changing markets. The performance of a system in relation to the market can be indicated by the following Key Performance Indicators (KPI):

- Functionality
- Time-to-market
- Cost
- Quality

In order to get more competitive, reducing cost and time-to-market is mandatory, while at the same time functionality and quality should increase. However, current practice shows that the relation between these indicators is often a trade-off. If, for example, one would improve the quality of a system, often the cost and time-to-market are increased. For a new system generation, the functionality and quality are intended to increase, due to customer demands. As a result, the time to market and cost are likely to increase as well. In order to withstand these negative effects, there is a demand for new engineering processes. In this project, we focus on the engineering process of NBG Industrial Automation.

NBG Industrial Automation, located in Nederweert, provides services in the field of electronics, embedded software and PC-software for industrial and medical applications. One of many products of NBG is the multimover, a theme park vehicle that follows a wire that is integrated in the floor. The multimover is a relatively new concept, since the vehicle acts and drives according to a scene program that is specified by the theme park. The scene program specifies at what speed the vehicle should ride at a certain position, when it should follow other vehicles etc. This concept makes the multimover a very flexible vehicle that can be used in theme parks, museums and in other recreational activities. The multimover is an example of a high-tech system and is used as a case study in this project.

In the following sections, we focus on two engineering processes. First, the traditional engineering process and subsequently, the model-based engineering process are explained and discussed. Then, the outline of this thesis is presented.

1.1 Traditional engineering

The traditional engineering process is usually based on Rook's V-model [Roo86], implementing a 'divide and conquer' strategy. The system is decomposed into smaller parts that are developed separately. When all parts are available, the parts are integrated to construct the system. For simplicity, only two levels of hierarchy are considered in the following discussion. The higher level is referred to as the system and the lower level is referred to as the components.

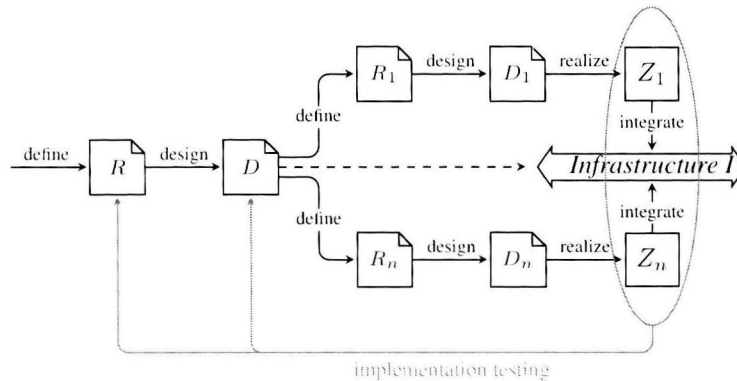


Figure 1.1: The traditional system development process

Figure 1.1 shows a graphical representation of the traditional engineering process, introduced by [Bra08], adapted by [The08]. Note that the focus in this picture is more on the representations of the systems and less on the different phases of the system development as in the traditional V-model. In the initial phase, the requirements of the system are defined, mostly together with the customer. The system requirements R define its functionality. These requirements can include constraints, for example, on safety and performance. In the next phase, a system design D of the system is made that satisfies the system requirements R . The system design D specifies the architecture, the decomposition of the system, the internal behaviour, and the technologies used. Furthermore, an infrastructure I has to be designed that specifies the interaction between all components.

After the system as a whole is designed, the system can be decomposed and the requirements for the components R_1 through R_n and component designs D_1 through D_n can be set up. The component designs D_1 through D_n specify how the components satisfy their requirements. Note that in this figure, the development process is simplified. In practice, these different phases are not sequential, but can start before the previous phase is finished. Furthermore, the development process has an iterative nature due to changing requirements or design errors that are discovered. If all components are designed and the designs are verified, the component realizations Z_1 through Z_n are made. These realizations have to be checked if they are according to the component designs D_1 through D_n and if the realizations satisfy the component requirements R_1 through R_n . This can be done by verification, validation and testing. Since the notions of verification and validation are not univocal, the definitions from [Hop93] are adopted:

- Verification means 'building the system right': substantiating that the system correctly implements its specification.
- Validation means 'building the right system': substantiating that the system performs with an acceptable level of accuracy by comparing its performance with test cases or human experts.

When the realizations of all components are ready and checked, they can be integrated by means of an infrastructure I . An infrastructure I is considered to be everything that connects components. Now, the integrated system can be tested whether it conforms to the intended system design D and whether it satisfies the system requirements R . At the end, the system is validated whether or not it meets all customer demands.

Traditionally, requirements and designs are mostly described by documents only. This can have several disadvantages [The08]:

- Documents may contain ambiguous or inconsistent information. Furthermore, documents may be incomplete or outdated. As a result, it is difficult to detect inconsistencies and to get a good system overview.
- Due to the informal nature of these documents, it is difficult to process these documents automatically.
- Documents are a static piece of information which makes it difficult to express and analyse the dynamic system behaviour. As a result, determining the integrated system behaviour based on component documentation only, is a difficult task.

Since documentation alone is not well suited to check the correctness of the system to be built, the behaviour of the system can only be verified systematically with testing when the complete system is realized. Formal models can be integrated in the engineering process to overcome these disadvantages. Using models in the engineering process is called model-based engineering. In the next section, the model-based engineering paradigm is explained.

1.2 Model-based engineering

The model-based engineering (MBE) system development process as described in [Bra08], is shown in Figure 1.2. The main difference between the traditional engineering process and the model-based engineering process is the inclusion of models in the development process. Before making realizations, all or some components can be modelled. Making models enables the use of a range of model-based analysis techniques and tools to support the development process.

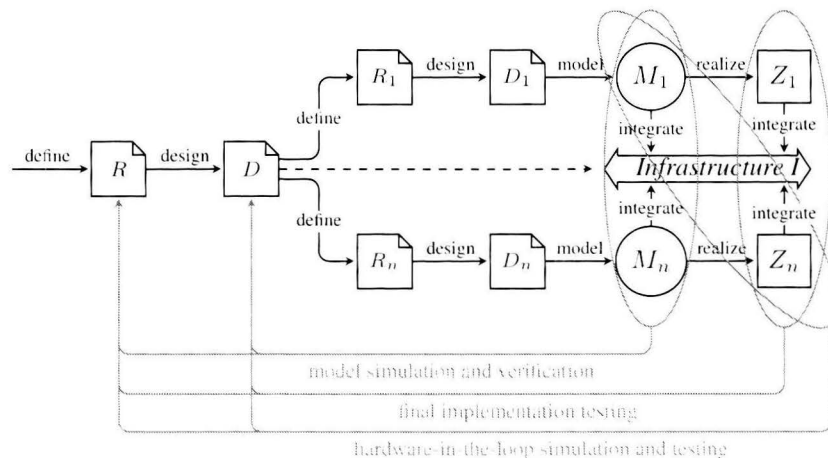


Figure 1.2: The model-based engineering system development process

We consider a model to be an abstract representation of a real component or system. Introducing models in the development process has several advantages [Bra08]:

- Models support a systematic approach to specify component and system behaviour with more consistency and less ambiguity than documents. Ambiguity is a lack of clear and exact use of words, so that more than one meaning is possible. Consider for example the following sentence: “He ate the cookies on the couch.”, which could mean that he ate the cookies which were on the couch, or it

could mean that he was sitting on the couch when he ate the cookies. The constructs used in formal models have a semantics defining precisely what they mean.

- Models make it easier to analyse dynamic behaviour as well as the performance of components or systems.
- By simulating and validating models, errors can be detected in an early stage of the system development process when no component is realized, which decreases risk in the system development process. Furthermore, simulating and validating models increases overall confidence in the correctness of the system.

If the models are validated and simulated and the conclusion is drawn that the system has the correct functionality, components can be realized. However, unless exhaustive, simulation can only show that the system might have correct behaviour and cannot guarantee correctness of the model. Formal verification techniques can be used to prove properties of models and can guarantee the correctness of models.

In the case that only some component realizations are available, it is possible to apply hardware-in-the-loop-simulation. In hardware-in-the-loop-simulation, the real hardware of the embedded system is used and tested with models in which the environment is simulated [Bro03]. The models of the components that are not available yet can replace their component realization. With this hardware-in-the-loop simulation and testing, the overall behaviour can be analysed in an early stage of the development process, which decreases risk and increases confidence in the correctness of the system. If all components are realized, all models can be replaced by the components. The final implementation can be verified if the complete system fulfills its system requirements R and corresponds with its system design D .

However, the correctness of a model M_i does not guarantee the correctness of a realization Z_i . Furthermore, in some systems, the requirements change over time. As a result, each changed system requirement results in changed requirements and designs for the components. The processing of these changes could be time consuming and error-prone, since these changes have to be made by hand mostly. By integrating supervisory control theory in the development process, these drawbacks can be partially addressed.

Supervisory control theory, explained in Chapter 2, enables us to generate a certain part of the control software and eliminates the manual design. The control software, derived according to this theory, is mathematically correct with respect to models of the components and models of control requirements. As a consequence, the design and the implementation do not need to be tested against the control requirements. This changes the development process from implementing and debugging the design and implementation, to designing and debugging the requirements. This means that the verification for the corresponding part can be eliminated, the engineer can focus on validating the system.

The expectations are that the addition of supervisory control theory in the model-based engineering process enhances the product development process. This brings us to the objectives of this project, stated in the following section.

1.3 Objectives

The objective of this project is to investigate the applicability of supervisory control theory in the product development process of NBG. The multimover is chosen as a case study for this project. A supervisor needs to be synthesized and implemented on a real hardware platform. From this case study, conclusions about the applicability of supervisory control synthesis at NBG can be drawn.

To this end, the following steps need to be carried out:

- Define models of a multimover and requirements associated with it that are needed for synthesizing a supervisor.

- Implement this supervisor within the current software environment and investigate the advantages and disadvantages of this set-up.
- Propose a toolchain that allows model-based engineering including supervisor synthesis.
- Investigate how supervisory control synthesis can be used within the product development of NBG and study the applicability.

In the next section, the outline of this thesis is presented.

1.4 Outline

In Chapter 2, the basics of supervisory control theory are explained. Two supervisory control frameworks are discussed, since these frameworks are most applicable, being the event-based framework of Ramadge-Wonham [Ram87] and the state-based framework of Ma-Wonham [Ma05]. Subsequently, the engineering process with inclusion of supervisory control theory is explained. At the end, a toolchain is proposed that allows for automatic generation of the controller software.

The functionality of the multimover and the models from which a supervisor can be obtained are explained in Chapter 3. In Chapter 4, modelling convenience of both supervisory control frameworks is discussed and an automatic conversion of models that can be used for synthesis is proposed.

A prototype of the implementation of the synthesized controller software is discussed in Chapter 5. Furthermore, potential pitfalls of this implementation and the applicability at NBG are addressed. This thesis ends with conclusions and suggestions for further research, presented in Chapter 6.

Chapter 2

Supervisory control theory

Supervisory control theory is used in this project to generate a specific part of the control software. In this chapter, the theoretical basics are explained that are needed to understand supervisory control theory.

A high-tech system can be divided in roughly two parts: the physical uncontrolled hardware and the control system. A schematic overview of a high-tech system is given in Figure 2.1. In this figure, the control system is depicted in blue.

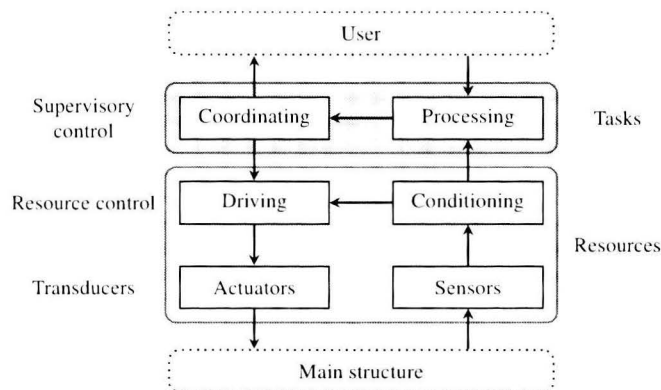


Figure 2.1: Schematic overview of a high-tech system

At the bottom level of this figure, the main structure is depicted, containing the physical hardware components. Sensors and actuators are mounted on these hardware components to monitor the position or state of these components and to actuate these components. These sensors and actuators are also called the transducers of the system. The sensor signals have to be processed and the actuators have to be controlled with feedback control that assures that the actuators reach the desired position in a desired way. This happens at the level of resource control. Above the resource control level, we have supervisory control. It coordinates the individual components and gives the desired functionality to the system. Supervisory control includes scheduling, planning and dispatching functions [Pat89]. In this thesis, we only concentrate on the supervisory control part of a high-tech system and which is referred to as the supervisor.

Supervisory control theory (SCT) is initiated by P.J. Ramadge and W.M. Wonham at the University of Toronto, Canada [Ram87, Won84, Won08]. This theory allows to synthesize a model of the supervisor from formal models of the uncontrolled system and formal models of the requirements. A graphical representation of this synthesis is shown in Figure 2.2a. First, the uncontrolled system (from now on: plant) is formally specified in terms of automata (M_P). A plant automaton describes the physically possible behaviour of the system to be controlled. Then, the requirement specifications of the supervisor are formally

defined in terms of automata (M_{R_S}). A model of the supervisor (M_S) is generated from these formal models.

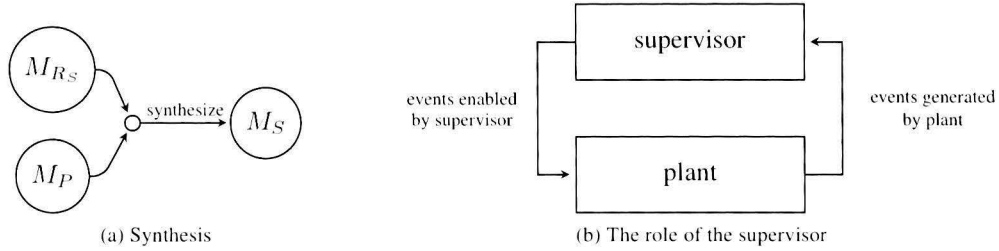


Figure 2.2: Supervisory control synthesis

The resulting supervisor can be used to supervise an uncontrolled plant (see Figure 2.2b). The supervisor can only react to observable events that are generated by the uncontrolled plant. The supervisor influences the behaviour of the plant by disabling certain controllable events. The method guarantees that the system consisting of the derived supervisor and the uncontrolled plant fulfills the requirements. If the supervised model does not contain the desired functionality, the models of the uncontrolled plant or the requirements are inadequate.

Different methods have been developed that allow for an automatic synthesis of a supervisor. The original framework, the event-based supervisory control framework [Ram87, Won84], is explained in Section 2.3. Since the main obstacle of this framework is computation complexity, several extensions have been developed that enhance the supervisor synthesis. A different approach is the state-based supervisory control framework by Ma and Wonham [Ma05, Ma06]. This approach makes use of state tree structures that provide concurrency and hierarchy. This framework is explained in Section 2.4. Section 2.5 describes how supervisory control theory can be embedded in the model-based engineering process. A toolchain that allows for model-based engineering is given in Section 2.6.

Supervisory control theory makes use of formal language theory and finite state machines, so-called automata. The assumption is made that the reader has some basic knowledge of set theory. The next section explains what automata are.

2.1 Automata and languages

2.1.1 Automata

An automaton is a model of behaviour composed of a finite number of states and transitions between those states. Each state should describe its behaviour in some measurable way [Cas07]. The transitions of an automaton are labeled by events. An event may be identified with a specific action taken (e.g. open the door, release the button etc.) and should be thought of as occurring instantaneously.

Definition 2.1 (Automaton). An automaton is a quintuple

$G = (\mathbf{X}, \Sigma, \xi, \mathbf{x}_0, \mathbf{X}_m)$, with

- \mathbf{X} : the finite state set.
- Σ : the finite alphabet i.e. the event set.
- $\xi : \mathbf{X} \times \Sigma \rightarrow \mathbf{X}$: the transition map. This map shows which transitions are possible at each state.
- $\mathbf{x}_0 \in \mathbf{X}$: the initial state.

- $\mathbf{X}_m \subseteq \mathbf{X}$: the set of marker states. The marker states are used to describe completed tasks. They represent a set of states which we want always to be reachable by any behaviour [Mal03].

In a physical system, not all of the events can be influenced by a supervisor. This is captured within supervisory control theory by dividing events in two classes: controllable events and uncontrollable events. Controllable events are controlled by the supervisor (e.g. events of actuators). The controllable event set is denoted by Σ_c . Uncontrollable events cannot be controlled by the supervisor (e.g. a break-down event) and the uncontrollable event set is denoted by Σ_u .

Events can also be divided into observable and unobservable events, but this distinction is not used in this thesis. The reader can assume that all events mentioned in this thesis are observable events, except for the event τ , which is used for automaton abstraction (see Section 2.3.3).

In this thesis, event labels are written italic (e.g. *breakdown*) and state labels are written bold (e.g. **Busy**). Furthermore, automata can also be presented graphically. In our graphical representation, states are denoted by vertices, initial states are denoted by an unconnected incoming arrow and marked states are denoted by filled vertices. Controllable and uncontrollable events are drawn with solid and dashed edges respectively.

Example 2.1.1

In this example, a small automaton is presented that models a machine. In the initial state, the machine is idle. In this state, the machine can start processing a product. After some time, the product is finished and becomes idle again. Furthermore, a breakdown is possible when the machine is processing a product. If this happens, the machine has to be repaired before it can start processing a product again. Since the operator is not able to stop the machine when it is busy, nor can he prevent the machine from breaking down, these events are modelled as uncontrollable events. An automaton that represents this behaviour is:

$G = (\mathbf{X}, \Sigma, \xi, \mathbf{x}_0, \mathbf{X}_m)$, with

$$\begin{aligned} \mathbf{X} &= \{\text{Idle}, \text{Busy}, \text{Down}\} \\ \Sigma &= \{\text{start}, \text{finished}, \text{breakdown}, \text{repaired}\} \\ \Sigma_c &= \{\text{start}, \text{repaired}\} \\ \Sigma_u &= \{\text{finished}, \text{breakdown}\} \\ \xi &: \xi(\text{Idle}, \text{start}) = \text{Busy}, \xi(\text{Busy}, \text{finished}) = \text{Idle}, \\ &\quad \xi(\text{Busy}, \text{breakdown}) = \text{Down}, \xi(\text{Down}, \text{repaired}) = \text{Idle} \\ \mathbf{x}_0 &= \text{Idle} \\ \mathbf{X}_m &= \{\text{Idle}\} \end{aligned}$$

A graphical representation of this automaton is given in Figure 2.3. ☒

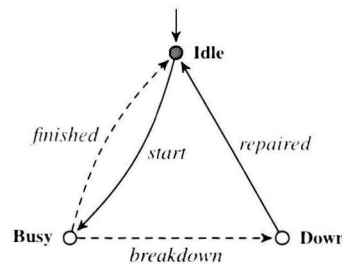


Figure 2.3: The famous small machine model

2.1.2 Languages

Automata represent languages. The language $L(G)$ generated by $G = (\mathbf{X}, \Sigma, \xi, \mathbf{x}_0, \mathbf{X}_m)$ contains all finite sequences of events. A sequence of events forms a *string*.

Definition 2.2. $L(G) := \{s \in \Sigma^* : \xi(\mathbf{x}_0, s) \text{ is defined}\}$, with Σ^* the Kleene closure of Σ , i.e. the collection of all finite sequences of events taken from Σ .

The marked language $L_m(G)$ by G is the set of strings (sequence of events), starting from the initial state and ending at a marker state of that corresponding automaton:

Definition 2.3. $L_m(G) := \{s \in L(G) : \xi(\mathbf{x}_0, s) \in \mathbf{X}_m\}$.

The prefix closure of L denoted by \overline{L} consists of all prefixes of strings in L :

Definition 2.4. $\overline{L} := \{s \in \Sigma^* : (\exists t \in \Sigma^*)[st \in L]\}$.

The natural projection of a language can be found by replacing all events that are in the language Σ but not in the projection language Σ' by the empty string ϵ :

Definition 2.5. Given Σ and $\Sigma' \subseteq \Sigma$. A mapping $P : \Sigma^* \rightarrow \Sigma'^*$ is called the natural projection with respect to (Σ, Σ') , if

$$\begin{aligned} P(\epsilon) &= \epsilon \\ (\forall \sigma \in \Sigma) P(\sigma) &:= \begin{cases} \sigma & \text{if } \sigma \in \Sigma' \\ \epsilon & \text{otherwise} \end{cases} \\ (\forall s\sigma \in \Sigma^*) P(s\sigma) &= P(s)P(\sigma) \end{aligned}$$

Now we know the basics of automata and languages, some properties of automata are explained that are needed to understand the synthesis procedure.

2.2 Language properties

In this section, some properties of languages are explained. First, nonblocking is discussed and subsequently, controllability is discussed.

2.2.1 Nonblocking

To explain the property nonblocking, first reachability and co-reachability are explained.

A certain state $\mathbf{x} \in \mathbf{X}$ is *reachable* if it can be reached by a sequence of events from the initial state \mathbf{x}_0 . An automaton is said to be *reachable* if all its states in this automaton are reachable. In Figure 2.4a, state 4 is not reachable, since no sequence of events leads to this state, starting from the initial state.

A certain state $\mathbf{x} \in \mathbf{X}$ is *co-reachable* if a marker state $\mathbf{x} \in \mathbf{X}_m$ can be reached from this state. An automaton is *co-reachable* if all its states are co-reachable. In Figure 2.4a, state 3 is not co-reachable, since no sequence of events from this state can lead to a marker state.

An automaton is *nonblocking* if all states that are reachable, are also co-reachable. The automaton depicted in Figure 2.4b is nonblocking, since from every reachable state, a marker state can be reached. Note that this automaton is not reachable, since state 4 cannot be reached.

Definition 2.6 (Nonblocking). G is nonblocking, if $\overline{L_m(G)} = L(G)$.

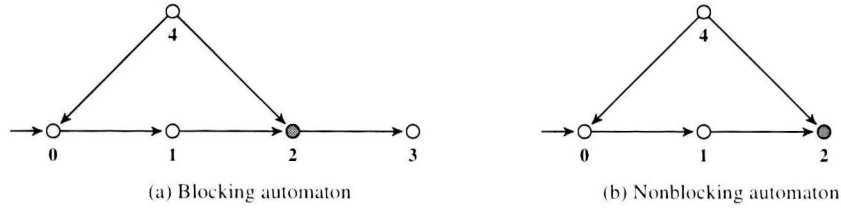


Figure 2.4: Illustration of the nonblocking property

2.2.2 Controllability

Requirement models specify the desired behaviour of the plant that should be matched by adding a supervisor. The supervisor can disable certain controllable events such that the controlled system contains the desired behaviour. Note that the requirement models could require to prevent certain uncontrollable events from happening. Nevertheless, uncontrollable events cannot be disabled by a supervisor. If this is the case, controllable events should be disabled which lead to states with uncontrollable events that have to be disabled. As a result, with the controllability property, no sequence of events exist that can lead to states that are violating the nonblocking property or certain requirements [Mor07].

A language \bar{K} is controllable with respect to an automaton G and uncontrollable alphabet Σ_u , if and only if for all sequences of events possible in both G and \bar{K} , after which an uncontrollable event is allowed by G , holds that it is also allowed by \bar{K} .

Definition 2.7 (Controllability). A language \bar{K} is *controllable* w.r.t. G and Σ_u if

$$(\bar{K}\Sigma_u) \cap L(G) \subseteq \bar{K}.$$

Example 2.2.1

In Figure 2.5, two automata are given, automaton G (Figure 2.5a) and a language K represented by automaton S (Figure 2.5b). Both automata have the same alphabet $\Sigma = \{a, b, c, d\}$ with $\Sigma_c = \{a, d\}$ and $\Sigma_u = \{b, c\}$. We want to check whether the language represented by automaton S is controllable w.r.t. automaton G .

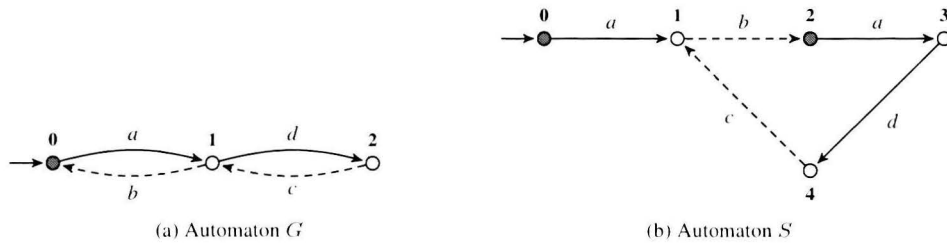


Figure 2.5: Illustration of the controllability property

Both automata accept the string aba . However, if b is appended to this string, only automaton G can accept this string. Automaton S cannot accept this string, since there is no event b possible after aba . Since b is uncontrollable, we can draw the conclusion that the language K , represented by automaton S , is not controllable w.r.t. automaton G and Σ_u . \square

The language of a supervisor that is synthesized with this theory is always controllable w.r.t. the plant models and the uncontrollable alphabet Σ_u . This means that whenever a plant automaton allows a given uncontrollable event to occur, the supervisor automaton also allows that uncontrollable event to occur in its

corresponding state. If a state is found in which an uncontrollable event is allowed by a plant automaton but not by the supervisor automaton, this corresponds to a ‘bad state’, i.e. a state in which the controllability condition is violated. The supervisor synthesis procedure disables as many controllable events as needed in the supervisor to ensure that this ‘bad state’ is not reachable. Note that this could result in an empty supervisor.

Both properties, nonblocking and controllability are important for synthesizing supervisors. In the remainder of this chapter, two frameworks are discussed that allow for supervisor synthesis. The first framework, the event-based framework of Ramadge-Wonham [Ram87] is explained in the next section.

2.3 Event-based supervisory control

In this section, the event-based supervisory control framework, initiated by Ramadge and Wonham [Ram87, Won08], is explained. The event-based synthesis procedure makes use of the automaton product. This automaton product is explained below.

2.3.1 Automaton product

To build a more complex automaton, the product of two automata can be computed. This automaton product is based on synchronization of shared events. This means that such an event can only happen if both original automata are able to do that event. A state of the automaton product is a marker state if both states in the original automata where it is referring to are also marker states.

Definition 2.8. Given automata $G_1 = (\mathbf{X}_1, \Sigma_1, \xi_1, \mathbf{x}_{0,1}, \mathbf{X}_{m,1})$ and $G_2 = (\mathbf{X}_2, \Sigma_2, \xi_2, \mathbf{x}_{0,2}, \mathbf{X}_{m,2})$, the automaton product $G_1 \times G_2$ is the automaton $(\mathbf{X}_1 \times \mathbf{X}_2, \Sigma_1 \cup \Sigma_2, \xi_1 \times \xi_2, (\mathbf{x}_{0,1}, \mathbf{x}_{0,2}), \mathbf{X}_{m,1} \times \mathbf{X}_{m,2})$ where

$$\xi_1 \times \xi_2((\mathbf{x}_1, \mathbf{x}_2), \sigma) \begin{cases} (\xi_1(\mathbf{x}_1, \sigma), \mathbf{x}_2) & \text{if } \sigma \in \Sigma_1 - \Sigma_2 \text{ and } \xi_1(\mathbf{x}_1, \sigma) \text{ is defined} \\ (\mathbf{x}_1, \xi_2(\mathbf{x}_2, \sigma)) & \text{if } \sigma \in \Sigma_2 - \Sigma_1 \text{ and } \xi_2(\mathbf{x}_2, \sigma) \text{ is defined} \\ (\xi_1(\mathbf{x}_1, \sigma), \xi_2(\mathbf{x}_2, \sigma)) & \text{if } \sigma \in \Sigma_1 \cap \Sigma_2 \text{ and } \xi_1(\mathbf{x}_1, \sigma) \text{ and } \xi_2(\mathbf{x}_2, \sigma) \text{ are defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Notice that the automaton product is commutative and associative. An example is given below how the automaton product works in practice.

Example 2.3.1

In Figures 2.6a and 2.6b, two automata are given. The alphabet of automaton G is $\{a, c\}$ and the alphabet of automaton H is $\{b, c\}$. As a result, c is a shared event between both automata. The resulting automaton product is given in Figure 2.6c. The state names of the automaton product refer to the states of the original automata G and H . As can be seen, the event c can only occur if both events b and a are done first.

⊠

2.3.2 Supervisor synthesis

Ramadge and Wonham have proven that, for a plant G and a requirement H , there always exists a supremal controllable sublanguage $K_{sup} \subseteq L_m(G) \cap L_m(H)$ [Ram87, Won84]. The supervisor represents this supremal controllable sublanguage K_{sup} . As a consequence, the synthesized supervisor is:

- **nonblocking**, i.e. from every state of the supervisor, a sequence of event can lead to a marker state.

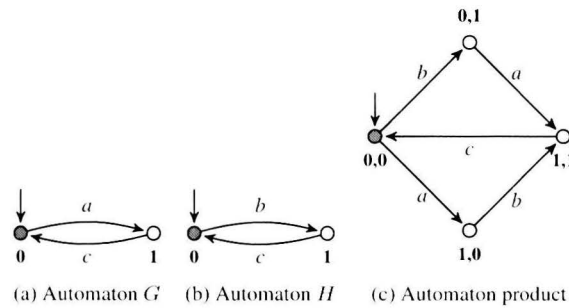


Figure 2.6: Example of an automaton product

- **controllable w.r.t. the plant G and Σ_u .** This means that whenever a plant model accepts an uncontrollable event, the supervisor also accepts this uncontrollable event.
- **maximally permissive,** i.e. the supervisor only disables events to states that do not satisfy the non-blocking or controllability property or are not allowed due to requirement models. Other behaviour is always possible.

An example of how supervisor synthesis is performed is given below.

Example 2.3.2

In Figure 2.7, three automata are given that represent two machines and one automatic guided vehicle (AGV). The two automata of both machines contain a controllable event *start* that represents the starting command and an uncontrollable event *finished*. The purpose of the AGV is to bring finished products from the first machine to the second machine and finished products from the second machine to the storage room. Since it is unknown where the product has to be stored inside the storage room, the event *store* is uncontrollable and cannot be controlled by a supervisor.

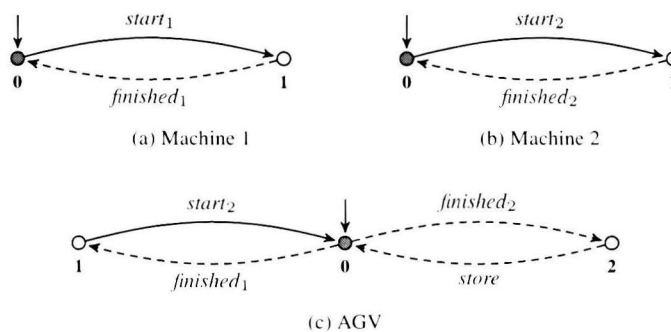


Figure 2.7: Plant models of Example 2.3.2

To synthesize a supervisor, the automaton product of these three automata has to be computed. This marks the so-called legal behaviour of the system. The calculation of this automaton product results in Figure 2.8a. Note that common events are synchronized.

A supervisor has to be controllable and nonblocking. To fulfill the nonblocking property, the supervisor has to ensure that states 6 and 7 are not active since from these states, a marker state cannot be reached. However, the uncontrollable event *finished₁* cannot be disabled, otherwise Definition 2.7 would be violated.

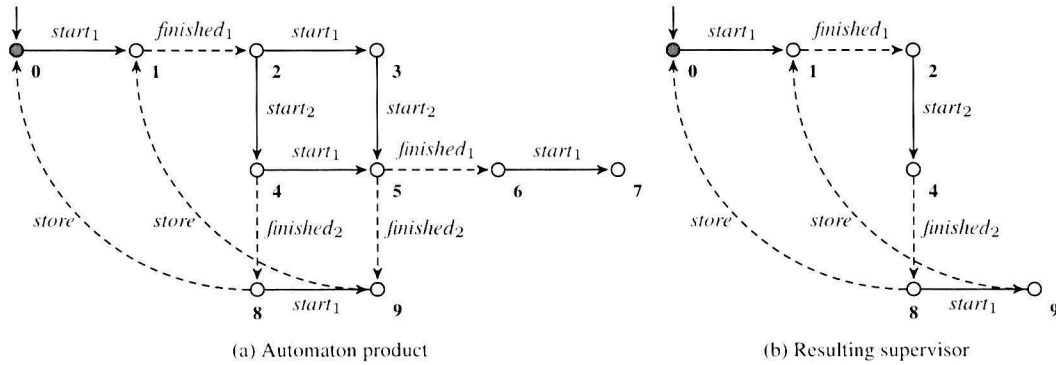


Figure 2.8: Automaton product and supervisor of Example 2.3.2

As a result, the supervisor has to prevent that state 5 becomes active. Therefore, all controllable events ($start_1$ and $start_2$) that lead to state 5 are disabled. After this, state 3 is not co-reachable anymore and the supervisor has to disable the controllable event $start_1$ to prevent deadlock. If all these events are disabled, the resulting automaton fulfills the nonblocking and controllability property and is a proper supervisor. The resulting supervisor is depicted in Figure 2.8b. \boxtimes

A common challenge in synthesizing a supervisor is that the state space of the system grows exponentially in the number of its components. Furthermore, a lot of requirements can also result in a so-called state-space explosion. As a result, the tooling may not be able to synthesize a supervisor due to memory constraints. To tackle this problem, a different supervision architecture can be used, e.g. hierarchical interface-based supervision [Led05] and distributed / modular supervision [Su09b, Su09c]. In this thesis, only distributed supervision is applied.

2.3.3 Distributed supervision

With distributed supervision, the control problem is divided into subproblems. One supervisor is synthesized for each subproblem and they cooperate together as a ‘team’ to give the complete functionality to the complete system. Distributed supervision is particularly interesting for two reasons: potentially low synthesis complexity and high flexibility, since a change in the system may result in only a small number of relevant local supervisors to be updated [Su09c]. In this subsection, we apply two different approaches to synthesize distributed supervisors, namely coordinated distributed supervision [Su09c] and aggregate distributed supervision [Su09b]. Please note that, for simplicity reasons, only distributed supervision with two distributed supervisors is discussed. For a more general and formal explanation of distributed supervision, the reader is referred to [Su09c, Su09b].

In Figure 2.9, the supervision architecture with distributed supervision is depicted [Cas07]. As can be seen, more local supervisors are synthesized to solve a certain subproblem. Note that these local supervisors do not need to supervise the complete plant. We assume that the plant spontaneously generates events. Each supervisor influences the behaviour of the plant by disabling certain controllable events. In distributed supervision, all supervisors are coupled with each other by parallel composition (depicted in Figure 2.9 with \parallel). In a parallel composition, an event can only be executed if all automata, in which this event is contained in its language, can execute it simultaneously. Thus, automata are synchronized on common events [Cas07].

Before both approaches for synthesizing distributed supervisors can be explained, another automaton operation has to be explained, namely automaton abstraction.

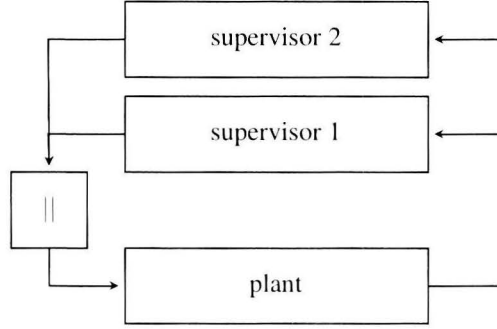


Figure 2.9: Distributed supervision

Automaton abstraction

Automaton abstraction [Su08a, Su08b] can simplify an automaton by removing certain transitions. This abstracted automaton has the important property that if an automaton abstraction is nonblocking, the original automaton is also nonblocking under the following condition. To preserve the blocking property, the automaton that needs to be abstracted, has to be *standardized*. To standardize an automaton, an extra event τ has to be brought in, which is uncontrollable and unobservable. A standardized automaton is an automaton which has an initial state with only τ -transitions from this initial state. Furthermore, this initial state is not allowed to be marked and no events should lead to the initial state.

Definition 2.9. The automaton $G = (\mathbf{X}, \Sigma \cup \{\tau\}, \xi, \mathbf{x}_0, \mathbf{X}_m)$ is standardized if

$$\begin{aligned} & \mathbf{x}_0 \notin \mathbf{X}_m \\ & (\forall \mathbf{x} \in \mathbf{X})[\xi(\mathbf{x}, \tau) \neq \emptyset \Leftrightarrow \mathbf{x} = \mathbf{x}_0] \\ & (\forall \sigma \in \Sigma)\xi(\mathbf{x}_0, \sigma) = \emptyset \\ & (\forall \mathbf{x} \in \mathbf{X})(\forall \sigma \in \Sigma \cup \{\tau\})\mathbf{x}_0 \notin \xi(\mathbf{x}, \sigma) \end{aligned}$$

For abstraction, an abstraction alphabet, denoted by Σ' , needs to be specified in order to know which events should be abstracted. At the end, the abstracted automaton is simplified by combining states that are equivalent under *weak bisimilarity*. Two states are equivalent under weak bisimilarity if they cannot be distinguished based on ‘observable’ behaviour. This ‘observable’ behaviour is specified by Σ'^* .

Definition 2.10. Given $G = (\mathbf{X}, \Sigma, \xi, \mathbf{x}_0, \mathbf{X}_m)$, let $\Sigma' \subseteq \Sigma$ and $P : \Sigma^* \rightarrow \Sigma'^*$ be the natural projection. A marking weak bisimulation relation on \mathbf{X} with respect to Σ' is an equivalence relation $R \subseteq \mathbf{X} \times \mathbf{X}$ such that, $R \in \{(\mathbf{x}, \mathbf{x}') \in \mathbf{X} \times \mathbf{X} \mid \mathbf{x} \in \mathbf{X}_m \Leftrightarrow \mathbf{x}' \in \mathbf{X}_m\}$ and

$$(\forall (\mathbf{x}, \mathbf{x}') \in R)(\forall s \in \Sigma^*)(\forall y \in \xi(\mathbf{x}, s))(\exists s' \in \Sigma'^*)P(s) = P(s') \wedge (\exists y' \in \xi(\mathbf{x}', s'))(y, y') \in R$$

The largest marking weak bisimulation relation on \mathbf{X} with respect to Σ' is called marking weak bisimilarity on \mathbf{X} with respect to Σ' .

An automaton G abstracted with abstraction alphabet Σ' is denoted with $G / \approx_{\Sigma'}$.

Definition 2.11. Given $G = (\mathbf{X}, \Sigma, \xi, \mathbf{x}_0, \mathbf{X}_m)$ and $\Sigma' \subseteq \Sigma$. The automaton abstraction of G with respect to the marking weak bisimulation $\approx_{\Sigma'}$ is an automaton $G / \approx_{\Sigma'} := (\mathbf{Z}, \Sigma', \delta, \mathbf{z}_0, \mathbf{Z}_m)$ where

$$\begin{aligned} \mathbf{Z} & := \mathbf{X} / \approx_{\Sigma'} := \{ \langle \mathbf{x} \rangle := \{ \mathbf{x}' \in \mathbf{X} \mid (\mathbf{x}, \mathbf{x}') \in \approx_{\Sigma'} \} \mid \mathbf{x} \in \mathbf{X} \} \\ \mathbf{z}_0 & := \langle \mathbf{x}_0 \rangle \\ \mathbf{Z}_m & := \{ \mathbf{z} \in \mathbf{Z} \mid \mathbf{z} \cap \mathbf{X}_m \neq \emptyset \} \\ \delta & : \mathbf{Z} \times \Sigma' \rightarrow 2^{\mathbf{Z}}, \text{ where for any } (\mathbf{z}, \sigma) \in \mathbf{Z} \times \Sigma', \\ & \delta(\mathbf{z}, \sigma) := \{ \mathbf{z}' \in \mathbf{Z} \mid (\exists \mathbf{x} \in \mathbf{z})(\exists u, u' \in (\Sigma - \Sigma')^*)\xi(\mathbf{x}, u\sigma u') \cap \mathbf{z}' \neq \emptyset \} \end{aligned}$$

A small example of an automaton abstraction is given below [Su08a].

Example 2.3.3

In Figure 2.10a, automaton G is depicted with $\Sigma = \{\tau, a, b, c\}$. Note that automaton G is standardized, since from the initial state, the only outgoing edge is a τ -event, the initial state is not marked and no events lead to the initial state. Suppose our abstraction alphabet is $\Sigma' = \{\tau, c\}$. The resulting abstracted automaton $G/\approx_{\Sigma'}$ is depicted in Figure 2.10b. This automaton is constructed by reducing the alphabet of the original automaton G to τ and c . Subsequently, for each state in the original automaton is evaluated which ‘observable’ transitions are possible. For example, if in state **1** event c occurs, the active state can be **1,2,3**, or **4**. This evaluation is done for each state. Furthermore, the states **1, 2** and **3** of automaton G are equivalent under weak bisimilarity. These states cannot be distinguished based on ‘observable’ behaviour. As a consequence, these states are combined and result in state **1** in automaton $G/\approx_{\Sigma'}$. Note that **4** is not equivalent under weak bisimilarity with other states, since state **4** is a marker state and all other states are no marker states. This automaton is a nondeterministic automaton, since states exist with outgoing edges with the same event.

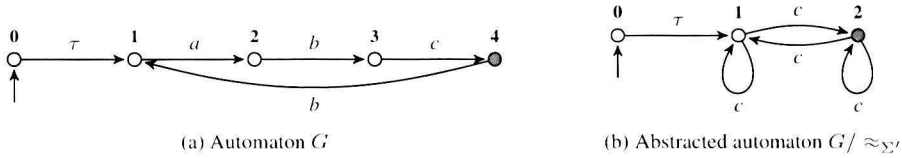


Figure 2.10: Example of automaton abstraction

⊠

Automaton abstraction is needed for synthesizing a distributed supervisor. Below is explained how a coordinated distributed supervisor can be synthesized.

Coordinated distributed supervision

To synthesize a coordinated distributed supervisor, the control problem has to be divided in control sub-problems. A local supervisor can be synthesized for every control subproblem. It is possible that some plant models are needed for synthesis of more than one local supervisor. If all local supervisors are synthesized, a nonconflicting check has to be performed, according to [Su08b]. If the local supervisors are nonconflicting, then the local supervisors form a proper distributed supervisor for the complete control problem.

However, if the local supervisors are conflicting, a coordinator has to be synthesized which solves this ‘conflict’. This coordinator can be synthesized from abstractions of the local supervisors combined with its plant model by means of the product operator. In Figure 2.11, a schematic overview is given for the coordinated supervisor design. In this overview only two local supervisors are synthesized. More local supervisors can be synthesized in the same way.

First, all local supervisors S_1, S_2 are synthesized out of plant models G_1, G_2 and requirement models H_1, H_2 . After this, an abstraction of the automaton product of each local supervisor S_i and their plant models G_i is computed with a certain abstraction alphabet Σ'_i . This abstraction alphabet Σ'_i must contain τ and shared events. A coordinator S can be synthesized from this abstraction product and, if necessary, requirement models H . If a coordinator S is synthesized, the local supervisors in combination with the coordinator are nonconflicting. A multiple-level multiple-coordinator distributed supervisor can be computed in the same way.

The main difficulty of coordinated distributed supervision is the choice of an abstraction alphabet. There are no explicit guidelines for constructing an abstraction alphabet, however, so-called boundary events are

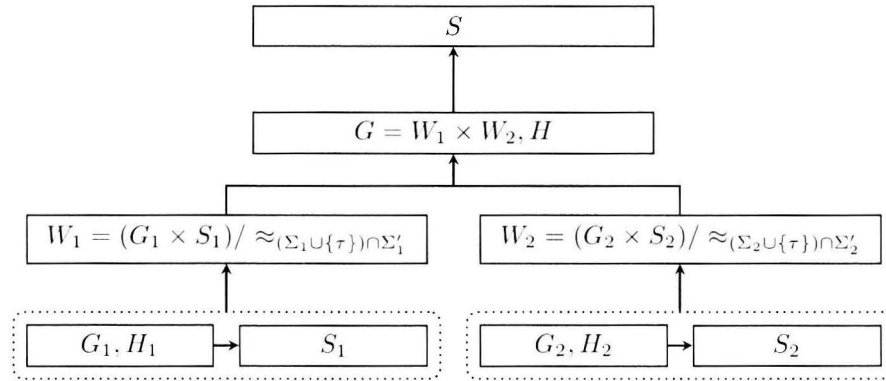


Figure 2.11: Coordinated distributed supervisor design

not likely to be abstracted. Boundary events are shared events that are used in more local supervisors. These events are not likely to be abstracted, since only shared events can create a conflict between local supervisors.

Aggregated distributed supervision

Aggregated distributed supervision uses an aggregative synthesis approach that computes nonblocking distributed supervisors. The key to the success of this approach is the automaton abstraction technique, that removes irrelevant internal transitions at each synthesis stage so that nonblocking supervisor synthesis can be carried out on relatively small abstracted models [Su09b].

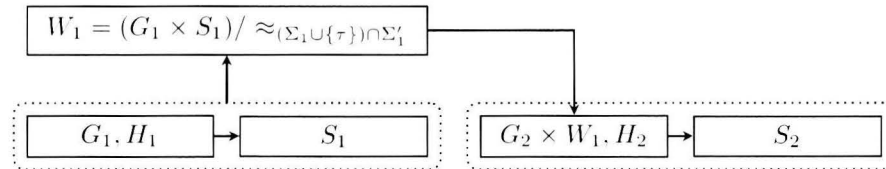


Figure 2.12: Aggregated distributed supervisor design

In Figure 2.12, a schematic overview is given of the aggregated distributed supervisor design. Please note that this figure shows only the design of a distributed supervisor design with two local supervisors.

First, a local supervisor S_1 can be computed from plant models G_1 and requirement models H_1 . Again, an abstraction alphabet Σ'_1 has to be constructed (which contains τ and shared events) and an automaton abstraction needs to be computed from the automaton product of the plant models G_1 and the local supervisor S_1 . The resulting automaton abstraction W_1 is used as a part of the plant model in the synthesis of the subsequent local supervisor. An aggregated distributed supervisor with more than two local supervisors can be computed in the same way. This algorithm always terminates and [Su09b] proves that the local supervisors are a nonblocking distributed supervisor of the complete plant G under all requirements if every local supervisor is nonempty.

The main difficulty of this approach is the order of the plant models for which a local supervisor is synthesized, such that it yields a solution. In [Su09b], an example of this approach is given for a linear cluster tool. This linear cluster can be divided into four submodules and subsequently, the order of synthesis of the local supervisors is found easily. However, more complex manufacturing systems may not have such a structured setting and, as a result, it may be difficult to get a good order of components.

With distributed supervision, we conclude the section where the event-based supervisory control framework is explained. In the next section, the other framework, i.e. the state-based supervisory control framework is explained.

2.4 State-based supervisory control

The state-based framework of Ma and Wonham [Ma05], is an extension of the original event-based framework. This framework uses, similar to the event-based framework, discrete-event systems for describing certain behaviour of the environment. However, a difference between both frameworks is that in the state-based framework, every discrete-event system belongs to a state tree structure (STS), a formalism that is computationally efficient for monolithic supervisor synthesis [Cai08]. To illustrate this, [Ma05] estimates that, based on the STS formalism, optimal nonblocking supervisory control design can be performed for systems of state size 10^{24} and higher.

2.4.1 Plant models

The entire state space of a system can be depicted by a state tree. State trees make use of different types of states:

- simple states: states with no child states.
- AND superstates: states with child states and represent a cluster of parallel states. That is, for the plant to be in the AND superstate, it must be at all child states simultaneously. In other words, AND superstate model parallel processes.
- OR superstates: states with child states and represent a set of exclusive states. That is, for the plant to be in the OR superstate, it must be at exactly one state of the child states.

An example of a state tree is given below.

Example 2.4.1

Consider the machine model of Example 2.1.1. Imagine we have two of these machines working in parallel. The complete state space of this system can be modelled by a state tree as depicted in Figure 2.13. This state tree consists of one AND superstate (**System**), which consists of two OR superstates (**M1** and **M2**). These OR superstates represent machine **M1** and **M2** which work in parallel. Each OR superstate consists of three simple states, which represent three states of the machine model, **Idle**, **Busy** and **Down**. Note that each of these states is prefixed with the name of the machine, e.g. **M1** or **M2**, in order to distinguish all states of both machines. Since each machine can be in one state simultaneously, each machine is modelled as one OR superstate. ☒

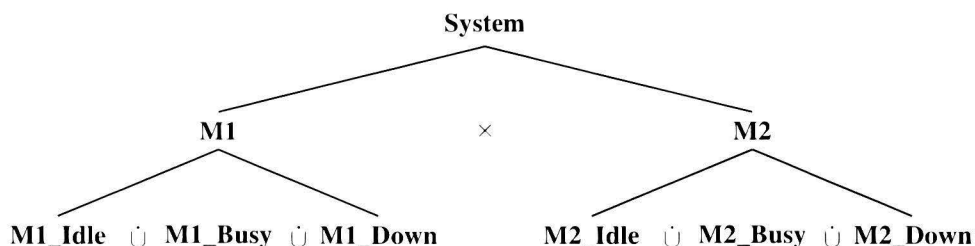


Figure 2.13: State tree of Example 2.4.1

In this thesis, only one type of state tree is used. This type of state tree consists of 3 levels. On the top level, we have one AND superstate. This AND superstate consists of one or more OR superstates. An OR superstate consists of simple states. Furthermore, the simple states of two OR superstates are always disjoint. The transition relation between simple states can be defined with *holons*, a local transition structure that describes the inner dynamics of OR components.

For an OR superstate, a holon represents the transitions between the child states of this superstate. In essence, a holon is a generalized automaton. Thus a set of system states organized in a state tree, equipped with holons describing the system dynamics. Since we are using holons the same way as automata in this project, we call holons from now on automata.

Example 2.4.2

In Figure 2.14, two automata are depicted that represent the behaviour of the two machines of the state tree depicted in Figure 2.13. As we can see, the behaviour of each OR superstate is described with one automaton and the states of each automaton correspond with the child states of the corresponding OR superstate,

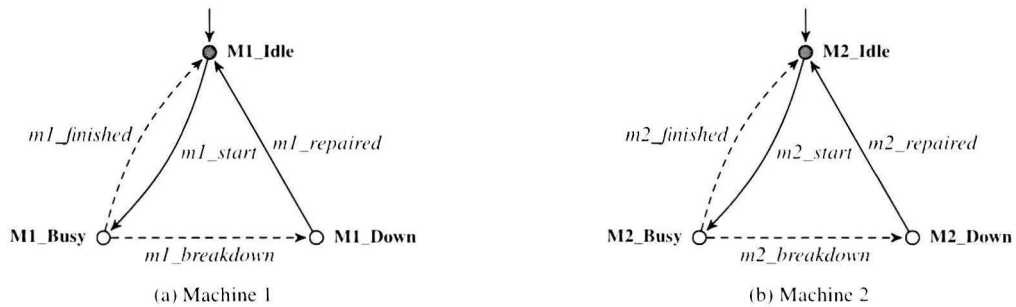


Figure 2.14: The automata of both machines of the state tree of Figure 2.13

⊗

2.4.2 Requirement models

Unlike the event-based framework, requirements can also be formulated in the state-based framework as conditions over *states*. Note that in the event-based framework, requirements can only be formulated by automata which specify sequences of events. The state-based framework allows to express requirements in the following three ways:

- Type 1: Mutual state exclusion. This type of requirement specifies which set of states may not be active simultaneously.
- Type 2: State-transition exclusion. This type of requirement specifies that a transition is not allowed if a certain set of states is active.
- Type 3: Memory module. This type of requirement is an ordinary event-based requirement, specified with an automaton.

[Jac09] has extended the allowed expressions of the state-based framework. Instead of the two state-based expressions (type 1 and type 2), three generalized state-based expressions are proposed. These generalized state-based expressions are described as logical expressions, based on propositional logic. We assume that the reader knows the basics of propositional logic. More information about propositional logic can be found in [Man85]. These expressions are converted to type 1 and type 2 expressions. In this thesis only the syntax is given. For a formal proof of the conversion, the reader is referred to [Jac09]. Before the three generalized state-based expressions are presented we introduce the used operators and predicates of [Jac09] in Table 2.1.

The first general state-based expression is a state-formula SF . This allows us to formulate any logical expression of state predicates, that must be satisfied by the plant under supervision.

Table 2.1: The operators and predicates for the used logical expressions

Operator	Description	Predicate	Description
\wedge	Conjunction (AND)	$\mathbf{x} \downarrow$	True if and only if the plant under supervision is in state 'x'.
\vee	Disjunction (OR)	$\rightarrow E$	True if and only if at least one of the events in the set of events E is enabled by the supervisor.
\neg	Negation (NOT)	$\nrightarrow E$	True if and only if each event in the set of events E is disabled by the supervisor.
\Rightarrow	Implication		

(a) Operators

(b) Predicates

Definition 2.12. A state-formula SF is defined as follows:

$$\begin{aligned}
 Op &::= \vee | \wedge | \Rightarrow, \\
 Pr &::= \mathbf{x} \downarrow, \\
 SF &::= Pr | \neg SF | SF Op SF.
 \end{aligned}$$

Example 2.4.3

Reconsider Example 2.4.1, where the plant model is given of two machines that work in parallel. The automata that specify the internal structure of the OR superstates **M1** and **M2** is given in Figure 2.3, but now with prefixes **M1** and **M2** for all state names and prefixes $m1$ and $m2$ for all event names. In this example, a logical expression is made that specifies that both machine may not be busy at the same time, i.e. $\neg \mathbf{M1_Busy} \downarrow \wedge \mathbf{M2_Busy} \downarrow$. \boxtimes

The next generalized requirement expression allows us to formulate any logical expression over state predicates to imply that a set of events must be disabled by the supervisor if the plant under supervision is in a state in which the state-formula is true.

Definition 2.13. A generalized state-transition exclusion (GST) is defined as

$$GST ::= (SF \Rightarrow \nrightarrow E),$$

Example 2.4.4

Reconsider the previous example. The situation of both machines being busy can also be prevented with disabling the event to start one machine if the other is busy. This can be specified by the following state-transition exclusions $\mathbf{M1_Busy} \downarrow \Rightarrow \nrightarrow \{ m2_start \}$ and $\mathbf{M2_Busy} \downarrow \Rightarrow \nrightarrow \{ m1_start \}$. \boxtimes

The last expression states that if at least one of the events in set E is enabled, the state-formula must be correct.

Definition 2.14. A generalized transition-state formula (GTS) is defined as

$$GTS ::= (\rightarrow E \Rightarrow SF),$$

Example 2.4.5

Reconsider the previous examples. The situation of both machines being busy can also be prevented if a specification assures that if a machine is started, the other may not be busy. This can be specified by the following transition state-formula: $\rightarrow \{ m1_start \} \Rightarrow \neg \mathbf{M2_Busy} \downarrow$ and $\rightarrow \{ m2_start \} \Rightarrow \neg \mathbf{M1_Busy} \downarrow$. \boxtimes

2.4.3 Supervisor synthesis

The state-based framework has a very powerful synthesis procedure applicable to systems of state size 10^{24} and higher. The reason for this powerful computation power is that the state-based framework does not perform the reachability analysis. The controllability condition is modified into a *weak controllability* condition, meaning that reachability is no longer a property of the controllability analysis.

The outcome of the synthesis is a state feedback control (SFBC) map. This means that feedback is given based on the *state* of the system. This SFBC is encoded with binary decision diagrams (BDD). A BDD is an acyclic directed graph which represents a boolean function, i.e. the outcome can be either 0 or 1. The supervisor synthesis produces control functions (encoded as a BDD) for each controllable event $\sigma \in \Sigma_c$. These control functions f_σ are evaluated with the state of the system. The outcome can be either 0, which means this event is disabled by the supervisor, or 1, which means that the event is enabled by the supervisor.

Example 2.4.6

Consider Example 2.4.1, where a state tree is constructed, which represents the state space of two machines that are working in parallel. The transition structures of the OR superstates **M1** and **M2** are specified by the automata depicted in Figure 2.14. However, all events of these automata are prefixed with the machine number (e.g. *m1_idle*, *m2_idle*, etc.), such that the languages of both automata are disjoint. Assume that both machines are not allowed to be busy simultaneously. In this example, a state-based supervisor is synthesized that prevents the machines to be busy simultaneously. The mutual exclusion requirement can be formulated with the state-formula $\neg (\mathbf{M1_Busy} \downarrow \wedge \mathbf{M2_Busy} \downarrow)$. The synthesis produces control functions, defined by BDDs, for all controllable events. Two relevant BDDs are given in Figure 2.15.

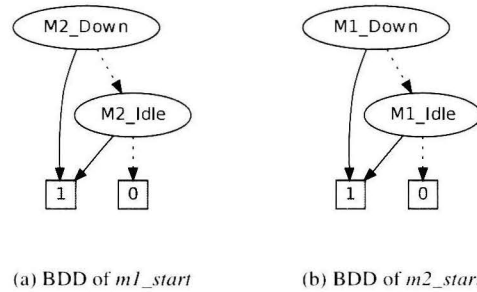


Figure 2.15: BDD functions of two controllable events

In Figure 2.15a, the control function of the controllable event *m1_start* is depicted, encoded as a BDD. This BDD contains four nodes and edges between these nodes. True and false edges are depicted with solid and dashed lines, respectively. To evaluate a BDD, nodes have to be evaluated. First, the initial node at the top is evaluated. If **M2_Down** is active, i.e. if machine 2 is down, the true edge is followed to the next node. Otherwise, the false edge is followed. The BDD evaluation terminates when node 0 or 1 is reached. If the BDD evaluation has 0 as outcome, the controllable event *m1_start* is disabled by the supervisor. If the BDD evaluation has 1 as outcome, the controllable event *m2_start* is enabled by the supervisor. Note that the only path to 0 is if machine 2 is neither down nor idle, i.e. the machine is busy. This is according to the specification, machine 1 may not be started when machine 2 is busy. The BDD for the controllable event *m2_start* can be evaluated in the same way. \boxtimes

Note that this control map is slightly different than in the event-based framework. The event-based framework uses automata to store the complete closed-loop language. As a result, a huge amount of memory is needed. The state-based framework uses a state feedback control map, which is encoded efficiently using

binary decision diagrams (BDD). In other words, a state-based supervisor gives feedback based on the *state* of the system and an event-based supervisor gives feedback based on the *language* of a system.

Now we have explained both supervisory control frameworks, we can discuss the integration of these supervisory frameworks in the model-based engineering process. This is explained in the next section.

2.5 Synthesis-based engineering

To integrate supervisory control synthesis in the model-based engineering process, the system has to be decomposed in a different manner as in the model-based engineering process. Let us decompose the system into a plant P and a supervisor S . Note that this clear separation between plant and supervisor, is mostly not evident in traditional engineering. Although supervisory requirements are present, they are mostly intermixed with regulative control requirements.

Figure 2.16 gives us a graphical representation of this framework [Sch09]. S/P means a plant P under supervision of a supervisor S .

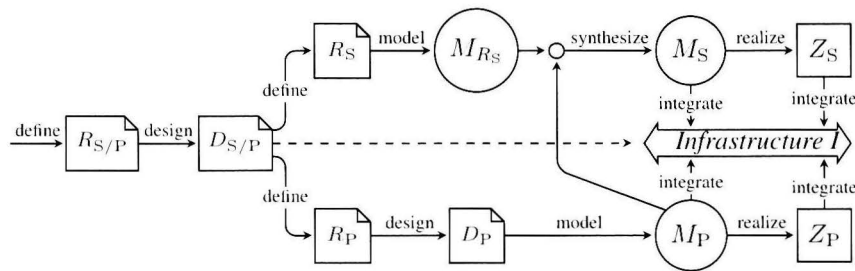


Figure 2.16: Supervisory controller synthesis framework

First, the requirements $R_{S/P}$ of the controlled system are defined. After this, a design $D_{S/P}$ of the system is made and decomposed into a uncontrolled plant and a supervisory controller. After decomposition we can set up the requirements of the supervisor R_S and the requirements of the uncontrolled plant R_P . The requirements R_S of the supervisor are formally modelled resulting in a model M_{R_S} of the control requirements. From the plant requirements R_P , a design D_P and one or more models M_P can be made. A discrete-event model can be used to synthesize a supervisor. The model of the supervisor M_S can be generated with supervisory control theory, with as input the discrete-event model of the plant and the model of the control requirements M_{R_S} . These plant models can also be used to simulate the behaviour of the uncontrolled plant under supervision of the model of the supervisor. After all models are derived, the analysis techniques of the model-based engineering paradigm can be used to test the system in an early stage of the system development process.

This means that in synthesis-based engineering, properties which are checked afterwards in traditional and model-based engineering, are used as input for generation of a design of a component that is correct by construction. As a consequence, the design and implementation do not need to be tested against the requirements, i.e. the verification can be eliminated. This changes the development process from implementing and debugging the design and the implementation, to designing and debugging the requirements.

Advantage of this integration is that in case of changes in the required functionality only the control requirements M_{R_S} have to be updated and the uncontrolled plant M_P might change. The supervisor is regenerated and correct w.r.t. the requirements by construction. In other words, the system is more evolvable.

Note that with synthesis-based engineering, still no formal link exists between the plant models and the plant realization. The plant model allows us to discover errors in an early stage, but the realization is usually made informally.

2.6 Toolchain

The CIF (Compositional Interchange Format) language [Bee08, Bee07] has been developed to provide a generic modelling formalism and to establish inter-operability of a wide range of tools by means of model transformations to and from CIF. The CIF language is based on hybrid automata. The language supports hierarchy and modularity to deal with large scale systems, by providing operators for model re-use, parallel models and nested models. Processes can interact by shared variables, by communication via shared channels and by synchronization by means of shared actions. Furthermore, arbitrary differential algebraic equations are supported, for the modelling of continuous time behaviour. The CIF-toolset¹ provides tools for simulation and translations to various verification tools. Moreover, it can be used in hardware-in-the-loop simulations and control prototyping. Furthermore, it facilitates code generation for various platforms.

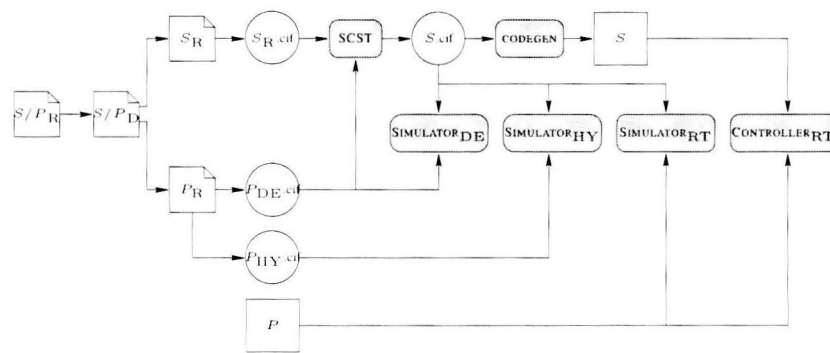


Figure 2.17: Toolchain based on CIF

In Figure 2.17, a toolchain, based on CIF, is depicted that allows for synthesis-based engineering [Sch09]. In this figure, tools are depicted in blue, models with circles and realizations with rectangles. The requirements and design of the plant P under supervision of a supervisor S are documented with S/P_R and S/P_D , and S_R and P_R denote the requirement documents of the plant and supervisor, respectively. The uncontrolled discrete-event behaviour of the plant are formally defined by means of automata, resulting in a CIF model $P_{DE}.cif$. The control requirements S_R are formally defined by means of automata or logical expressions, resulting in a CIF model $S_R.cif$. From these models, a supervisor can be synthesized with a supervisor control synthesis tool (SCST) and translated to the CIF language, resulting in a model of the supervisor $S.cif$. Two tools are used for supervisor synthesis and translations exist to and from these two used file formats:

- Supervisor Synthesis Package² (SSP) for the event-based supervisory control framework.
- Ma-Wonham's NBC tool³ for the state-based supervisory control framework.

The CIF simulator can be used to simulate the model of the supervisor $S.cif$ together with the discrete event model of the plant $P.cif$ in order to analyse its behaviour with respect to the control requirements. Furthermore, the discrete-event model of the plant can be replaced by the hybrid CIF model of the plant $P_{HY}.cif$. The model of the plant can also be replaced by the real hardware in order to test if the real hardware corresponds with the models. If this does not correspond, the real hardware or the models could be inadequate. The model of the supervisor can also be used for code generation (CODEGEN) and it can be implemented on a real-time control platform which is connected to the actual hardware of the plant.

This section concludes the explanation of the supervisory control theory and supporting tools. Two frameworks are discussed that allow for supervisor synthesis. Furthermore, the integration of supervisory control

¹Downloadable at <http://dev.se.wtb.tue.nl/projects/chi-tooling/downloads>

²Downloadable at <http://dev.se.wtb.tue.nl/projects/chi-tooling/downloads>

³Downloadable at <http://se.wtb.tue.nl/sewiki/wonham/start>

synthesis to the engineering process is explained in Section 2.5 and the toolchain that supports the supervisory controller design is presented in Section 2.6. In the next chapter, supervisory control theory is applied to an industrial case in order to obtain a model of the supervisor.

Chapter 3

Case study: the multimover

In the previous chapter, the theory is explained that is needed for synthesizing supervisors. In this chapter, supervisory control theory is applied to an industrial case of NBG Industrial Automation.

In the amusement park business there is a demand for new rides that diverge from the conventional roller coaster or ferris wheel. A relatively new concept is the multimover: a vehicle that drives around while following an invisible track. The track layout is defined by an underground electrified wire that can be detected by the vehicle. This offers the possibility for revolutionary new ride concepts with crossings, switches, junctions and driving into and out of dead-end tracks.

Vehicles can interact with each other in such a way that the passengers have influence on the ride experience, for example with target shooting systems and similar competitive features. By gaining a certain score, new scenes can be unlocked. This interactivity and the fact that the passengers cannot see the actual track makes the ride more exciting because of the unexpectedness of the vehicle's actions. A picture of a multimover is given in Figure 3.1.



Figure 3.1: The multimover

3.1 Functionality

Multimovers are Automated Guided Vehicles that can follow an electrical wire integrated in the floor. This track wire produces a magnetic field that can be measured by track sensors. Next to the track wire, floor codes are positioned, that can be read by means of a metal detector. These floor codes give additional information about the track, e.g. the start of a certain scene program, a switch, junction or a dead-end. The

scene program, which is read by the scene program handler, defines when the vehicle should ride at what speed, when it should stop, rotate, play music and in which direction the vehicle should move (e.g. at a junction).

An operator is responsible for powering up the vehicle and deploying it into the ride manually. The operator also controls the dispatching of the vehicles in the passenger board and unboard area. The vehicle can receive messages from Ride Control. Ride Control coordinates all vehicles and sends start/stop commands to these vehicles. These messages are sent with wireless signals or by means of the track wire. Multimovers are not able to communicate with other vehicles.

Safety is an important aspect of this vehicle. Therefore, several sensors are integrated in this vehicle to avoid collisions. First, proximity sensors are integrated in the vehicle to avoid physical contact with other objects. We can distinguish two types of proximity sensors. A *long* proximity sensor that senses obstacles in the vicinity of six meter and a *short* proximity sensor that senses obstacles in the vicinity of one meter. The vehicle should ride slower when an object is only detected by a long proximity sensor and stop when an object is detected by the short proximity sensor. This stop is not an emergency stop. When the short proximity sensor does not detect an object any more, the vehicle should start riding automatically. Secondly, a bumper switch is mounted on the vehicle that can detect physical contact with other objects. The vehicle should respond to this with an emergency stop. If an emergency stop is declared, an operator has to deploy the vehicle back into the ride. Finally, an emergency stop has to be declared when the battery power is too low or when a system failure occurs. The vehicle should not become active when the bumper switch is still active or the battery power is still too low.

3.1.1 Supervisory control requirement

The functionality that is described above, is the functionality of the closed-loop system e.g. the hardware and the controller software. Before the design of the supervisory controller can be made, requirements of the supervisory controller should be set up. An overview of the control architecture with supervisory control is given in Figure 3.2.

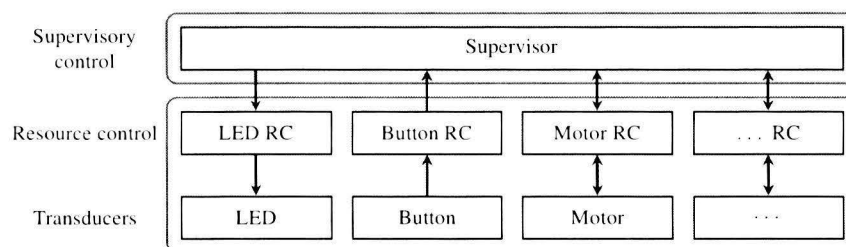


Figure 3.2: The control architecture

At the lowest level, we have the components (transducers) of the multimover. In this figure, a LED, a button and a motor are depicted. The next level is the level of resource control. This resource control contains feedback control of these individual components, e.g. a PID-controller for a motor. The upper level, supervisory control, coordinates the discrete behaviour of all components. The main requirement of this supervisory controller is safety. This supervisory control requirement has three aspects:

- **Proximity handling** The supervisory controller has to assure that the multimover does not collide with other vehicles or obstacles. To this end, proximity sensors are integrated at the front and back which can detect an obstacle if it is within a certain range of the multimover. To avoid collisions, the multimover should drive with a safe speed and stop if the obstacle is too close to it.
- **Emergency handling** The system should stop immediately and should be powered off when a collision occurs. To detect collisions, a bumper switch is mounted on the multimover. The same applies

when the battery level is too low. The LED interface should give a signal when an emergency stop has been performed. The multimover should be deployed back into the ride by an operator manually.

- **Error handling** When a system failure occurs (e.g. a malfunction of a motor), the system should stop immediately and should be powered off to prevent any further wrong behaviour. The LED interface should give a signal that an emergency stop has been performed. The multimover should be deployed back into the ride by an operator manually.

3.1.2 Components

A graphical overview of components that are relevant to this project and their states is given in Figure 3.3.

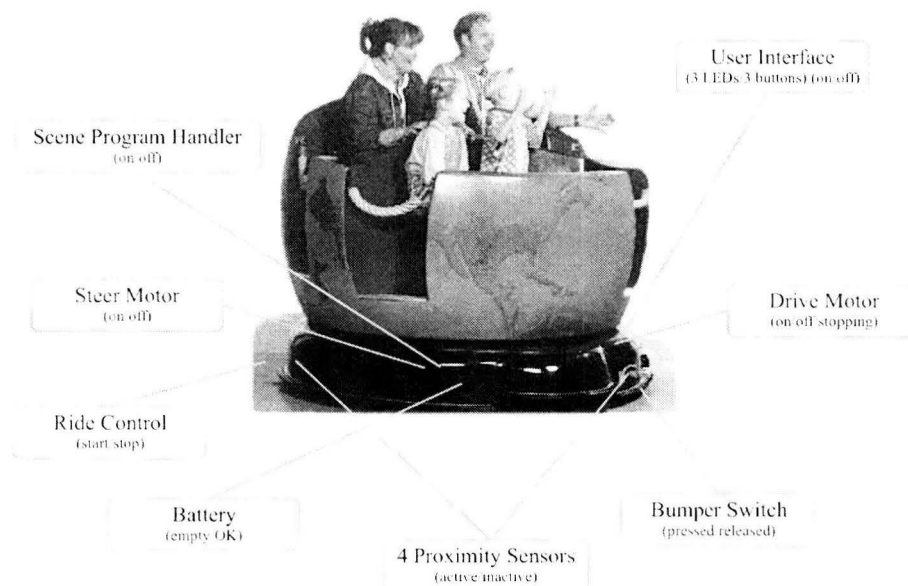


Figure 3.3: Relevant components of the multimover

3.2 Plant models

The plant models are models of the actual components and their low-level control. These models are needed to fulfill the supervisory requirement, which is stated in the previous section. The control architecture of the multimover is given in Figure 3.2. The plant models represent the actual behaviour of the transducers and their resource control. Within supervisory control theory, plant models are defined by automata. Each transducer and its resource control are modelled by one automaton. Automata consist of states and transitions labeled by (controllable and uncontrollable) events. In the following list, the representations of the states and events of plant models are given. Unfortunately, we cannot generalize these modelling guidelines, since they are case specific.

- States of the plant models represent all relevant states of each resource (e.g. on, off, empty, active, etc.).
- Controllable events represent relevant discrete commands/tasks (function calls) to the resource control (e.g. *enable*, *disable*, etc.). These actions can be controlled by the supervisory controller.

- Uncontrollable events represent messages that are sent from the resource control to the supervisory controller (e.g. a failure notification, a sensor event, etc.). These events cannot be controlled by the supervisory controller.

When models are made, assumptions have to be made. The assumptions that are made in our models are listed below.

Assumption 3.1. The plant models are made with the assumption that the resource control of the multimover is working correctly. This means that if a command is given, the command is carried out correctly. For example, if a drive motor is being enabled, we assume that the resource control of the drive motor switches on the drive motor.

Assumption 3.2. The communication between the plant and the supervisor is infinite fast. This means that if an event occurs at the plant (e.g a button is pressed), the supervisor is synchronized immediately. Furthermore, this assumption means that events cannot overtake each other and cannot get lost.

In this section, the components depicted in Figure 3.3 are divided into two groups of components, namely the buttons and sensors that monitor the state or position of a certain part of the multimover, and the actuators, which actuate a certain part of the multimover. Furthermore, a plant model is introduced that models the state of the multimover, since a lot of control requirements are based on the state of the multimover. The event and state names that are presented in the remainder of this section are simplified, i.e. without prefixes, for clarity reasons. All used event and state names are disjoint. For a complete overview of the plant models, the author refers to Appendix A.

3.2.1 Buttons, sensors and Ride Control

In this subsection, the components are discussed that can detect a certain change of state of the multimover. First of all, three buttons are integrated in the multimover and are used to reset the vehicle and to deploy the vehicle into the ride. Moreover, several sensors are integrated in the multimover to detect certain changes in the outside world. Proximity sensors can detect the proximity of an obstacle. A bumper switch can detect physical contact with an obstacle. Furthermore, a battery meter is integrated in the vehicle to measure the battery level of the multimover.

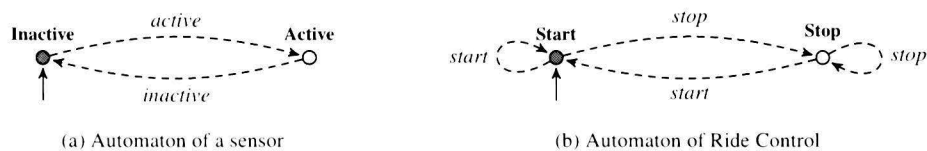


Figure 3.4: Automata of input components

Each button and sensor is modelled by one automaton. These automata have all the same structure and an example is depicted in Figure 3.4a. A sensor can generate two events: *active* and *inactive*. Each event labels the transition from one state to another, e.g. if a sensor becomes active, the event *active* is generated. The automaton representing a button have corresponding names, e.g. *on* and *off*.

Ride Control can send a 'general start/stop' command to all the multimovers to start or stop all the multimovers in an attraction. Ride Control sends these commands constantly with a certain interval. Therefore, it is possible that the same command is sent over and over again. This behaviour is captured by the automaton depicted in Figure 3.4b.

Note that these events of the automata presented in this subsection are all uncontrollable events, since the supervisor cannot prevent them from happening.

3.2.2 Actuators

Several actuators are integrated in the multimover that actuate a certain component of the multimover. Interface LEDs are used to show an operator the actual state of the multimover. Furthermore, the drive motor moves the multimover and the steer motor steers the multimover in the direction of the wire integrated in the floor. Lastly, the scene program handler reads the scene program and sends commands to the rotation device, drive motor and audio device.

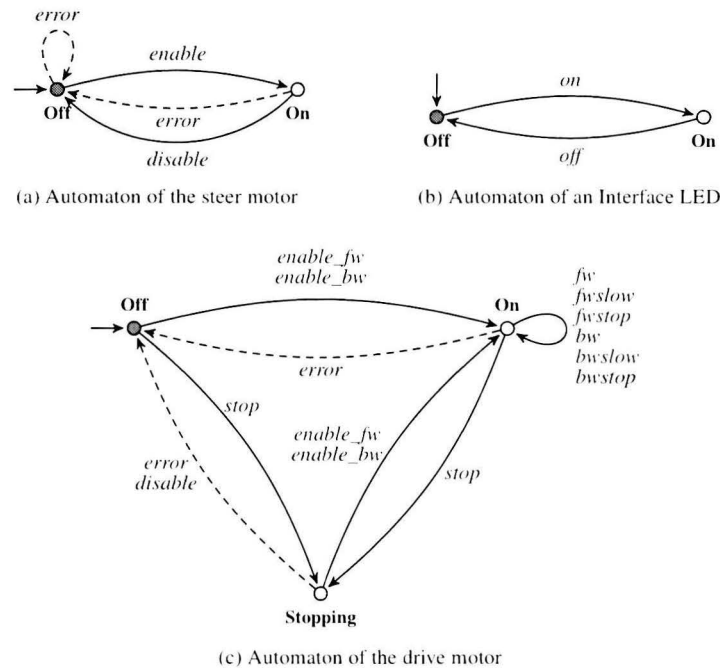


Figure 3.5: Automata of actuators

In Figure 3.5a, the automaton of the steer motor is given. The relevant states of the steer motor are **On** and **Off**. The actuation signals that are important for the supervisory controller are switching on the steer motor (*enable*) and switching off (*disable*). This motor contains a hardware safety if the motor is short-circuited or has a hardware failure. If this hardware safety is activated (*error*), the motor is automatically switched off. Since the hardware safety can also be activated when the motor is switched off and still slowing down, the event *error* is selflooped at state **Off**.

All LEDs of the multimover are modelled by the automaton depicted in Figure 3.5b. The LEDs of the multimover can be in two states: **On** and **Off**. The events *on* and *off* represent the function call of switching on and off of the LED.

In Figure 3.5c, the automaton of the drive motor is given. This automaton is basically the same as the automaton of the steer motor. However, it contains an extra state **Stopping**, since a control requirement of the multimover is that the steering motor may not be switched off if the multimover is still moving (e.g. stopping), for safety reasons. Therefore, an extra event *stop* is introduced that stops the drive motor. If the drive motor has stopped, the uncontrollable event *disable* is done and the drive motor is switched off. Because we want to be able to set the maximum speed of the drive motor, the events *fw*, *fwslow*, *fwstop*, *bw*, *bwslow* and *bwstop* are introduced. Also, the drive motor contains a hardware safety when the motor is short-circuited or has a hardware failure. The motor is automatically switched off when this hardware safety is activated. This is modelled with the event *error*.

Note that we could model the function call to switch on the steer motor if it is already on. However, since we assume that the resource control is working correctly, we have not modelled this behaviour. Furthermore,

if all possible function calls are modelled, one can end up with unnecessary complex plant models.

3.2.3 Multimover model

The multimover itself can also be in three states, namely Emergency, Reset and Active. This can be modelled with an automaton as depicted in Figure 3.6. Emergency denotes the state of the multimover that all components are switched off and the multimover has to be reset manually by pushing the reset button. If the reset button is pushed, the multimover should enter the state **Reset**. From this state, the multimover can be deployed into the ride (**Active**) or can switch back to **Emergency** (if an emergency event occurs). Since a lot of control requirements are based on the state of the multimover, this automaton is introduced for modelling convenience.

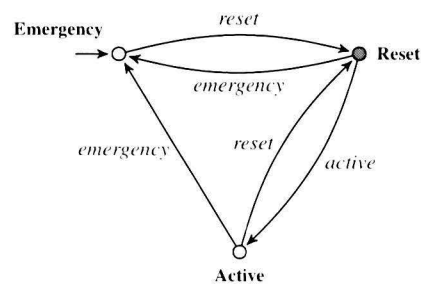


Figure 3.6: Plant model of the multimover

3.2.4 Plant-modelling aspects

Making models of manufacturing systems is a craftsmanship. The original supervisory control theory does not give any information how to model real life systems. In this subsection, some considerations about making discrete models of components are discussed.

Uncontrolled plant vs. partially controlled plant

The plant components can be modelled in different ways. The components can be modelled with some behaviour already restricted (partially controlled), or the physically possible behaviour with no restrictions can be modelled. To give an example of a partially controlled plant, in Figure 3.7, a plant model is depicted that represents the drive motor, steer motor and scene program handler in one automaton. In the beginning, this automaton was suitable to model the behaviour of the multimover. However, this automaton already has some control incorporated in the behaviour, since it is only possible to switch on and off the drive motor, steer motor and scene program handler simultaneously. Furthermore, this plant model assumes that a change of direction initiated by the scene program (*mi_chdir*), is carried out by the drive motor immediately. To get a better understanding of what behaviour is uncontrolled and what is desired (controlled), the automaton of Figure 3.7 is rejected and three new automata are designed for the steer motor, drive motor and scene program handler.

In the end, we have chosen for plant models that represent the uncontrolled behaviour of each component. In this way, plant models are obtained that match exactly the behaviour of the interface of the components. Decomposing the system in an uncontrolled plant and a supervisor gives a clear view of the system's functionality. Nevertheless, the reader has to bear in mind that the supervisor synthesis is slightly more difficult with these unrestricted plant automata, since more behaviour has to be restricted by means of requirement models. However, with distributed supervision, this aspect does not have to cause a problem.

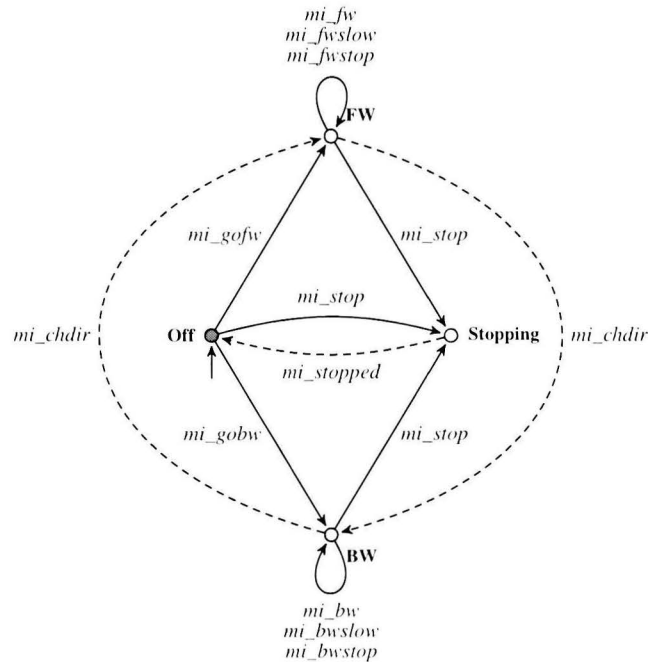


Figure 3.7: An automaton representing three components

Which way of modelling is more suitable depends on the goal: e.g. for the purpose of implementation or not.

Marked behaviour

Marker states are used to describe completed tasks. They represent a set of states which we always want to be reachable by any behaviour [Mal03]. If more states in the same plant models are marked, one can not assure that a certain marker state set is reachable. This is illustrated with an example.

Example 3.2.1

Consider the automaton in Figure 3.8a. For the sake of convenience, all events are controllable. In this automaton we have two marked states, state 0 and 3, since these states represent a completed task (e.g. reset and active). However, the nonblocking property assures that a marked state always is reachable, but not necessarily all of them. If state 2 is active in Figure 3.8a, then marker state 0 is not reachable anymore. In other words, if more marker states are reachable, one cannot assure that a certain marker state is reachable.

Now consider Figure 3.8b. In this automaton, only one state is marked. If we synthesize a supervisor for this automaton, the resulting supervisor would disable event β from state 1 to 2, since state 2 and 3 are not co-reachable. Since only one state is marked, one can always assure that this state is reachable. \boxtimes

In the model of the multimover, only those states are marker states, that are active if the multimover is reset and no sensors are active. Since only these states are marker states, the resulting supervisor always assures that the multimover can be reset.

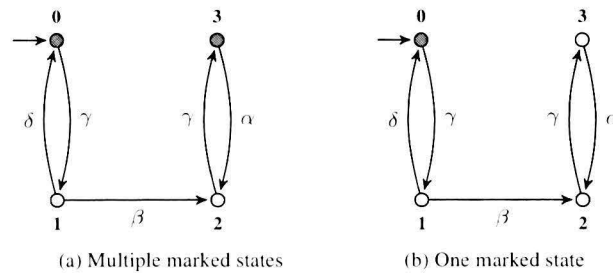


Figure 3.8: Marked behaviour

3.3 Modules

A divide-and-conquer strategy is often applied to get a good overview of control problems. This means that the control problem is cut into pieces and these smaller control problems are solved. We can divide the control problem of the multimover into five subproblems:

- **LED actuation** An operator must be able by looking at the Interface LEDs to check in which state the multimover is. This means that the states of the LEDs represent the current state of the multimover. It is a task of the supervisor to actuate the LEDs according to the state of the multimover. Which LED should be on and which should be off in every state of the multimover is summarized in Table 3.1.

Table 3.1: Active LEDs for each state

	Emergency	Reset	Active
ResetLED	On	Off	Off
ForwardLED	Off	On	Off
BackwardLED	Off	On	Off

- **Motor actuation** The drive motor, steer motor and scene program handler have to be switched on and off according to the state of the multimover. If the multimover is in the state **Active**, all motors can be switched on. If the multimover is in the state **Reset** or **Emergency**, all motors have to be switched off.
- **Button handling** The user interface of the multimover contains three buttons. First, a Reset button is used to reset the vehicle if the multimover is active and deployed into the ride or is in the state **Emergency**. Furthermore, a forward button and a backward button is used to deploy the vehicle into a certain direction. A control task of the supervisor is to enter the corresponding state when a button is pushed.
- **Proximity / Ride Control handling** Four proximity sensors are integrated in the multimover to detect obstacles that are in the vicinity of the multimover. Two proximity sensors are mounted on each side of the multimover. On each side, we can distinguish two types of proximity sensors. A long proximity sensor senses obstacles in the vicinity of six meter. If a long proximity sensor is sensing an object in the traveling direction, the multimover should react on this with slowing down to a safe driving speed. Furthermore, a short proximity sensor is integrated that senses obstacles in the vicinity of one meter. If an obstacle is detected by the short proximity sensor, the multimover should stop in order to prevent a collision.

As already told, Ride Control can send a 'general start/stop' command to all multimovers in order to stop and start the complete ride. Since a 'general stop' command of Ride Control can be considered as a short proximity stop, we can see this as the same control task as proximity handling. If Ride

Control is sending a 'general start' command again, the multimover should start riding automatically (depending of the state of the proximity sensors in the current driving direction).

The control task of the supervisor is to slow down or stop the multimover if a proximity sensor is activated in the travelling direction of the multimover or Ride Control is sending a 'general stop' command.

- **Emergency handling** In order to guarantee the safety of the passengers, the multimover should be deactivated immediately when an emergency situation occurs. We can distinguish the following emergency situations:
 - Battery power too low
 - Bumper switch collision detection
 - Drive motor driver failures
 - Drive motor not connected or defect
 - Wire signal lost
 - Steering motor not connected or defect
 - Steering motor driver failures

It should not be possible to reset the multimover if the bumper switch is still activated or the battery power is still too low. A control task of the supervisor is to enter the **Emergency** state of the multimover when an emergency situation occurs.

Now we have divided the control problem into subproblems, we call the control part that solves each problem a control module. In the next section, requirement models are presented that are used to synthesize a supervisor for the multimover control problem.

3.4 Requirement models

In this section, requirement models are discussed that give the desired functionality to the multimover. For the sake of simplicity, only the requirement models of the emergency handling control module are discussed. For an explanation of all requirement models, the reader is referred to Appendix A. This also holds for an explanation of all used event names and state names.

As already mentioned in the previous chapter, requirements have to be modelled by automata in the event-based approach. The state-based approach allows the user to define requirements also by logical specifications. This section is divided into two subsections to explain the requirements of both approaches.

3.4.1 Event-based model

In the event-based supervisory control framework, requirements can only be modelled with automata. The requirements of the emergency control handling module are depicted in Figure 3.9.

The first requirement, depicted in Figure 3.9a, specifies that the events *mm_active* and *mm_reset* are only allowed to take place if the bumper switch is not activated. This requirement can be modelled by taking the plant automaton of the bumper switch and adding a self-loop with events *mm_active* and *mm_reset* at the state that represents the bumper switch not being activated.

The second requirement, depicted in Figure 3.9b, specifies that the events *mm_active* and *mm_reset* are only allowed to take place if the power level of the battery is sufficient. Again, this requirement can be modelled by taking the plant automaton of the battery and by adding a self-loop with events *mm_active* and *mm_reset* at the state that represents the battery being not empty.

The last requirement, depicted in Figure 3.9c, specifies when the event *mm_emergency* is allowed to occur. The event *mm_emergency* is only allowed to occur after activation of the bumper switch (*bs_press*), the power level of the battery becoming too low (*ba_empty*), a parse error of the scene program (*sh_error*), a failure of the drive motor (*dm_error*) or a failure of the steering motor (*sm_error*). If one (or a sequence) of these ‘emergency events’ takes place, the requirement allows the occurrence of the event *mm_emergency*. If the event *mm_reset* takes place, occurrence of the event *mm_emergency* is not allowed. Note that this requirement only puts restrictions on the occurrence of the event *mm_emergency*, all other events are allowed to take place in any order without restrictions.

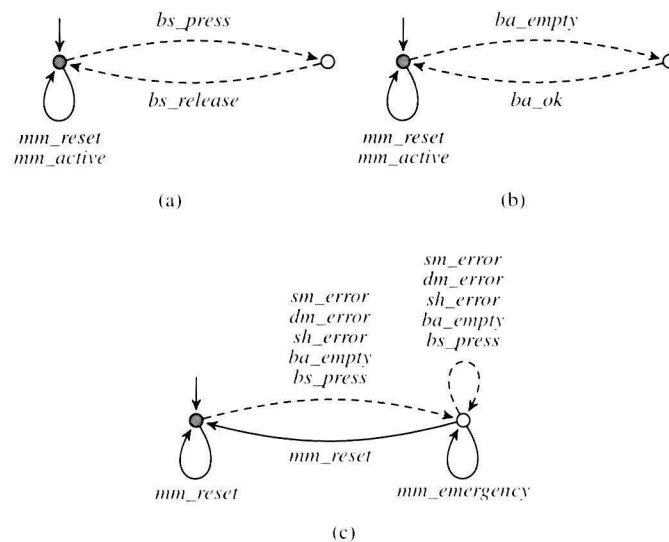


Figure 3.9: Requirement models of the emergency module

3.4.2 State-based model

Within the state-based supervisory control framework, requirements can be modelled by logical expressions and automata.

Logical expressions

[Jac09] proposes three generalized state-based expressions, described as logical expressions based on propositional logic. In the Emergency handling control module, we are only using one type of generalized state-based expression, namely a generalized transition-state formula:

$$\rightarrow \{ mm_reset, mm_active \} \Rightarrow \mathbf{BS_Released} \downarrow \wedge \mathbf{BA_OK} \downarrow$$

This generalized transition-state formula specifies that the multimover may only switch to active or reset (*mm_active* or *mm_reset*) if the battery level is ok (**BA_OK**) and the bumper switch is released (**BS_Released**).

Automata

In the state-based model of the multimover, one memory is used as a requirement in the Emergency handling control module. This memory is the automaton depicted in Figure 3.9c. As already described in the previous subsection, this automaton tracks the sequence of ‘emergency events’ and the event *mm_reset* and determines when the event *mm_emergency* is allowed to occur, based on the previous sequence of events.

3.5 Supervisor synthesis

Now all models are explained, we can synthesize a supervisor for the posed control problem of the multi-mover. This section is divided into two subsections. The first subsection evaluates the supervisors synthesized in the event-based framework. The second subsection evaluates the supervisor that is synthesized in the state-based framework.

3.5.1 Event-based supervisor

As is known, the main obstacle of the event-based framework is the calculation complexity problem. The event-based supervisor synthesis is based on the automaton product. As a result, a centralized event-based supervisor is not possible to synthesize due to a state space explosion. To give an idea of the calculation complexity, the number of states of the automaton product of the plant automata is approximately 50.000 states. One can overcome this problem by using a modular approach.

Coordinated distributed approach

With a modular approach, we divide the control problem into smaller subproblems. For each subproblem, a supervisor is synthesized. Since we have divided the control problem into control modules in Section 3.3, we have synthesized a supervisor for every module. If all supervisors are synthesized, a nonconflicting check has to be performed in order to guarantee the nonconflicting property.

The size (e.g. number of states and number of transitions) of each modular supervisor is listed in Table 3.2. As we can see, small supervisors can be achieved by dividing the control problem into smaller subproblems and synthesize a supervisor for each control problem. The size of each supervisor depends on how many components (e.g. plant automata) are involved. Furthermore, the size of a supervisor also depends on how restrictive the requirements are. If a lot of parallel behaviour is allowed, the number of states and transitions of the supervisor can grow rapidly.

Table 3.2: Size of modular supervisors for each module

Module	# states	# transitions
LED actuation	25	77
Motor actuation	41	222
Button handling	193	1541
Emergency handling	181	2149
Proximity handling	481	4513

Disadvantage of the modular approach is that the nonconflicting check is computationally expensive in comparison to the modular synthesis. To give an illustration, each supervisor can be computed within five seconds, but the nonconflicting check takes about ten minutes. The nonconflicting check can be avoided by using an aggregated modular approach.

Aggregated distributed approach

The main idea of the aggregated modular approach is to synthesize a supervisor, take an abstraction of the automaton product of the synthesized supervisor with the plant models to filter out irrelevant information and use this abstraction model as a plant model for the synthesis of the next supervisor. If all supervisors synthesized with the aggregated modular approach are nonempty, then they are guaranteed to be nonconflicting. However, aggregated modular approach needs a 'good' ordering of modules. To come up with this order can be relatively difficult. To illustrate this, the same supervisors are synthesized in a different order.

The results are listed in Table 3.3. As we can see, the size of the supervisor depends heavily on the order of synthesis.

Table 3.3: Size of modular supervisors depending on synthesis order

Module	Order	# states	# trans.	Order	# states	# trans.
LED actuation	1	25	77	5	41	125
Motor actuation	2	41	222	2	257	1428
Button handling	3	465	3477	4	177	765
Emergency handling	4	89	626	3	118	609
Proximity handling	5	225	1953	1	481	4513

3.5.2 State-based supervisor

The state-based supervisory control framework of Ma and Wonham [Ma05] is known to be efficient for monolithic supervisor synthesis. The synthesis tool produces within a second a BDD for every controllable event. For the supervisory control problem of the multimover, the maximum BDD size is 15 and the minimum BDD size is 1. The size of the BDDs can be reduced by variable ordering. Variable ordering is the ordering of the OR superstates in the state tree. However, since the size of the BDDs was not too large for implementation, variable ordering is not used in order to reduce the size of the BDDs.

3.6 Supervisor validation

The supervisors that are synthesized with both frameworks have to be validated in order to check if the models of the controlled system represents the intended behaviour. This can be done by simulating the behaviour of the plant P under supervision of supervisor S . The role of simulation in the synthesis-based engineering framework is depicted in Figure 3.10. With simulation, the closed-loop behaviour of

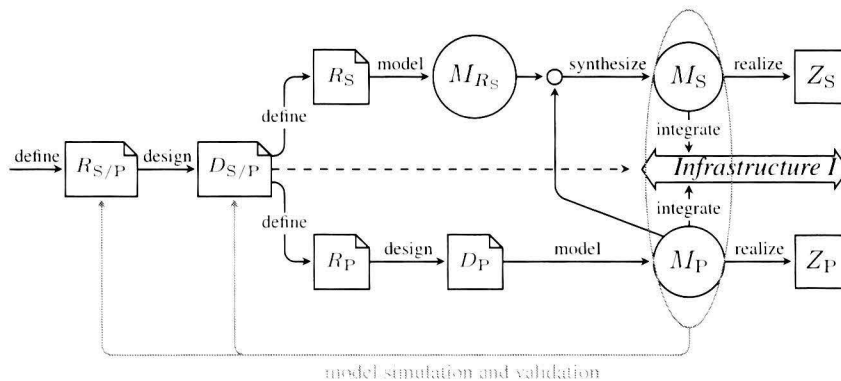


Figure 3.10: Simulation in the synthesis-based engineering framework

the system can be simulated to see if the supervisor and plant conforms to the system design $D_{S/P}$. Note that simulation can find errors, but cannot prove the absence of errors. If the closed-loop behaviour is not conform the system design $D_{S/P}$, the models that are needed for supervisor synthesis (M_P and M_{R_S}) have to be changed. In this section, two simulation techniques are discussed, namely discrete-event simulation and hybrid simulation. Both simulation techniques can be performed with the CIF toolset.

3.6.1 Discrete-event simulation

With discrete-event simulation, the state space of the closed-loop behaviour is explored by a state-space stepper. With a state-space stepper, certain traces in the state space of the closed-loop system can be evaluated, whether the supervisor disables the right transitions in a certain state along the trace. With this technique, even rare situations that are not likely to occur, can be simulated. Discrete-event simulation is used excessively in this project in order to validate the synthesized supervisors.

However, models which are used for synthesis do not contain information about time, position or other continuous information, which can be useful for analysing the dynamic behaviour of a system under control of a supervisor. Simulation of models with discrete *and* continuous behaviour is called hybrid simulation, which is explained in the next subsection.

3.6.2 Hybrid simulation

A more detailed model of the plant can be developed to study the dynamic or timed behaviour of the plant under control of the derived supervisor. This can be done by refining the discrete plant model with continuous behaviour. The following example explains how a discrete plant model can be hybrid.

Example 3.6.1

Consider the automaton depicted in Figure 3.11a, which is a model of a motor, that can be switched on (*enable*) and off (*disable*). This automaton does not contain any timing information or other continuous behaviour. This model is suitable to use for supervisor synthesis.

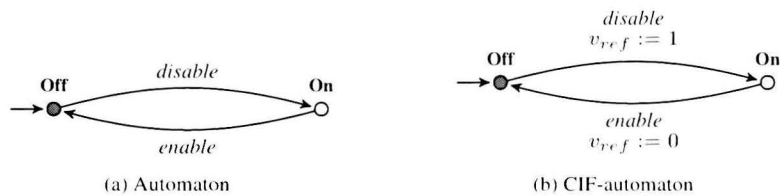


Figure 3.11: Automata used for supervisor synthesis and hybrid simulation

In Figure 3.11b, the same automaton of a motor is depicted, only with an additional action at every event. If the motor is switched on, a certain reference speed v_{ref} is set to 1. If the motor is switched off, the reference speed v_{ref} is set to 0. Then, the next process is a reference tracker, which models the resource control of the motor:

$$\begin{aligned} v < v_{ref} &\rightarrow a := 1 \\ v = v_{ref} &\rightarrow a := 0 \\ v > v_{ref} &\rightarrow a := -1 \end{aligned}$$

This process has three modes:

- 1 the actual velocity v of the motor is lower than the reference speed v_{ref} , the motor should accelerate ($a := 1$).
- 2 the actual velocity v of the motor is equal than the reference speed v_{ref} , the motor should not accelerate ($a := 0$).
- 3 the actual velocity v of the motor is higher than the reference speed v_{ref} , the motor should decelerate ($a := -1$).

The last modelled process contains information about the relationship between the position x , velocity v and acceleration a :

$$\begin{aligned} v &= \dot{x} \\ a &= \dot{v} \end{aligned}$$

These three parallel processes interact with each other by means of shared variables. Suppose that these three processes are a part of a larger model with a supervisor that switches the motor on and off every second for a certain reason. These processes can be used in a hybrid simulation experiment. In Figure 3.12, the results of this experiment are shown. As we can see, the motor is switched off at $t = 0$ and has no initial

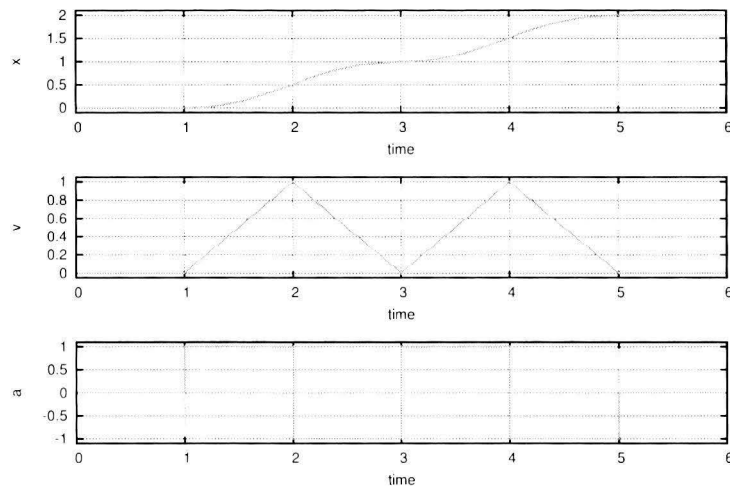


Figure 3.12: The hybrid simulation results

speed. At $t = 1$, the motor accelerates and the position x is increasing. Subsequently, the motor starts to slow down at $t = 2$ and starts accelerating at $t = 3$. Then, the motor slows down at $t = 4$ and stops at $t = 5$. ☒

Hybrid simulation can be useful for simulating models of systems with relevant dynamic or timed behaviour. However, hybrid simulation is not used in order to validate the supervisors of the multimover, since the multimover does not contain dynamic or timed behaviour that was relevant for validation of the supervisors. By only validating the supervisors and plant models with discrete-event simulation, we were able to draw the conclusion that these models represent the components and their desired behaviour satisfactorily.

In this chapter, models are presented which are needed for synthesizing a supervisor. Both supervisory control frameworks [Won84, Ma05] have been used to synthesize a supervisor and the results are presented. Furthermore, simulation techniques are discussed that can be used to validate the supervisor. In the next chapter, both frameworks are discussed more thoroughly, to address the applicability of both frameworks.

Chapter 4

Frameworks for supervisor synthesis

The previous chapter describes how a supervisor with the correct behaviour can be synthesized for the case study, the multimover. Two frameworks exist for supervisor synthesis, namely the event-based framework of Ramadge and Wonham [Ram87, Won84] and the state-based framework of Ma and Wonham [Ma05]. In this chapter, both frameworks are discussed more thoroughly in order to address the applicability of both frameworks.

Two aspects of both frameworks are discussed in the following sections. In Section 4.1, the synthesis aspects of both frameworks are discussed. Then, modelling aspects are discussed in Section 4.2. This section gives motivation to the next section where a conversion of logical expressions to automata is proposed. A conversion tool has been built and an experiment has been carried out in order to check if a conversion of logical expressions to automata is feasible. This is described in Section 4.4. This chapter ends with a discussion of the proposed conversion in Section 4.5.

4.1 Synthesis aspects

In the previous chapters, two frameworks were discussed: the event-based framework of Ramadge and Wonham [Ram87, Won84] and the state-based framework of Ma and Wonham [Ma05]. In this section, the synthesis aspects are addressed.

The main challenge of the event-based supervisory control framework is calculation complexity, since the product of all automata has to be computed to synthesize a supervisor and this computation is polynomial in time. Furthermore, the number of states of this automaton product increases easily and as a result, the product cannot be computed due to memory constraints. To overcome this difficulty, many approaches have been suggested recently. Distributed supervision is introduced to avoid the calculation of the product of all models. A distributed approach can also be useful to get a better overview of the functionality of the system. In addition, a distributed supervisor has more implementation flexibility, since a change in the target system may result in an update of only a couple of local supervisors [Su09c].

However, with distributed supervision, a nonconflicting check has to be performed which is also computationally expensive. The abstraction technique by [Su08a] is a step in the right direction, since this technique removes irrelevant transitions without losing nonblocking information. With a nonconflicting check based on the abstraction technique of [Su08b], coordinated distributed supervisors can be synthesized for systems with large state spaces. If an aggregative approach of synthesizing distributed supervisors is used, nonconflictingness does not need to be checked.

Supervisory control theory guarantees that the synthesized centralized supervisor is optimal in terms of permissiveness. This means that all behaviour is allowed as long as it does not violate the requirements or the nonblocking or the controllability property. However, no proof exists that a distributed supervisor

is maximally permissive, which means that a distributed supervisor can be more restrictive in terms of allowed behaviour.

The state-based framework has an efficient algorithm for synthesizing supervisors by using state-tree structures. However, at some point, the model can be too large for the state-based framework to calculate a supervisor, since this approach is essentially a centralized approach [Su09b]. Furthermore, the state-based framework does not consider supervision under partial observation. Until now, no possibilities exist to work in a distributed way with the state-based framework.

4.2 Modelling aspects

The original event-based framework uses automata to describe plant models and requirement models. A drawback of modelling requirements by automata, is that they may be not intuitive. Furthermore, if the occurrence of a certain controllable event is strongly coupled with a lot of states, one can end up with specifying large requirement automata. Even if an engineer is able to model such a requirement with an automaton, it may be hard to convince other engineers that this automaton really specifies the right behaviour as intended. In addition, system designers are often confronted with the following problem: how do we know that a requirement in automata indeed captures the intended requirement?

Thus, defining requirements in a way that is intuitive and easy to understand is important for the engineer to express the control requirements. The state-based framework is more convenient for modelling control requirements than the event-based framework, since we can use state-based expressions *and* automata to specify the desired behaviour. State-based expressions are expressions with conditions over states, which are often found in system requirements. However, [Jac09] concludes that deriving these state-based expressions suitable for the state-based framework is an error-prone and meticulous task. To this end, some logical specifications are proposed for automatic generation of these state-based expressions. With these logical specifications, the engineer can express requirements by logical specifications, that naturally follow from informal, intuitive requirements. These logical specifications can be converted to the original state-based expressions, which can be used for synthesizing a supervisor. An example of a requirement, specified by an automaton (in the event-based framework) and by a logical specification (in the state-based framework) is given below.

Example 4.2.1

Suppose we want to specify that the multimover only stops in the forward direction when Ride Control is sending a ‘general stop’ command or the short proximity front sensor is active. In this example, we specify this requirement by an automaton and by a logical specification. Note that this requirement is the occurrence of an event dm_fwstop under condition of a set of states.

Automaton

This requirement is modelled by an automaton in Figure 4.1. This requirement is basically the product of the plant automata of Ride Control (see Appendix A.1.6) and the short proximity front sensor (see Appendix A.1.5), only with extra selfloops. It contains extra selfloops of the event dm_fwstop in the states that represent Ride Control sending the ‘general stop’ command or the short proximity front sensor being active. With these selfloops, the alphabet of the requirement automaton is extended with the event dm_fwstop and therefore not allowing the event dm_fwstop at the state that represents Ride Control giving the ‘general start’ command and the short proximity front sensor being inactive.

Logical specification

The requirement stated above can be modelled with the following logical expression:

$\mathbf{RC_Start} \downarrow \wedge \mathbf{PSF_Inactive} \downarrow \Rightarrow \neg \{ dm_fwstop \}$, which states that the event dm_fwstop is not allowed when state $\mathbf{RC_Start}$ and state $\mathbf{PSF_Inactive}$ are active. \boxtimes

During this project, we have noticed that specifying some control requirements with logical expressions is by far more intuitive than specifying them by automata. Nevertheless, the event-based framework can be used for distributed supervision, hierarchical supervision and supervision under partial supervision. To

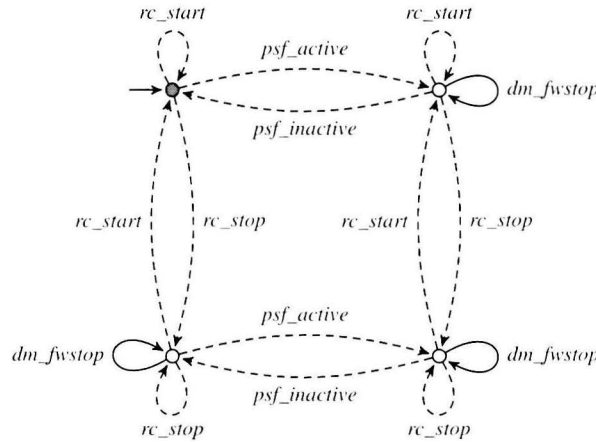


Figure 4.1: Control requirement modelled by an automaton

have both advantages of modelling convenience and supervisor synthesis flexibility, we propose in the next section a conversion of logical expressions to automata.

4.3 Conversion of state-based expressions to automata

In this section, a conversion is presented of the original two types of state-based expressions of the synthesis tool of Ma and Wonham to automata. These state-based expressions are a mutual state exclusion and a state-transition exclusion. In order to place conditions on states, it is convenient to use a logic formalism. Therefore, state predicates of [Jac09] are used. A state predicate $\mathbf{x} \downarrow$ is true if and only if an automaton is in state \mathbf{x} .

4.3.1 Mutual state exclusion

The first type of state-based expression that is converted to an automaton is a type 1 specification of the Ma-Wonham tool. This is a mutual state exclusion, which states that a set of states may not be active at the same time.

Definition 4.1. A mutual state exclusion over n automata G_i with $i = 1, \dots, n$ with corresponding state predicate $\mathbf{x}_i \downarrow$ can be written as a logical expression by

$$\neg \left(\bigwedge_{i \in 1..n} \mathbf{x}_i \downarrow \right) \quad (4.1)$$

Consider we have n automata $G_i = (\mathbf{X}_i, \Sigma_i, \xi_i, \mathbf{x}_{0,i}, \mathbf{X}_{m,i})$ with $i = 1, \dots, n$ where we want to prevent that some states are active simultaneously. The states of the product of all automata $G_1 \dots G_n$ represent all possibilities of state sets of all automata. If a certain state set is illegal, the corresponding state and all in- and outgoing transitions in the automaton product have to be removed. Now, the mutual state exclusion is satisfied by this requirement specified by an automaton. Note that a removal of a state in an automaton could lead to the situation that other states are not reachable anymore. This situation is captured by taking only the *accessible* part of an automaton, denoted by $Ac(G)$ [Cas07].

Since only one state in an automaton can be active simultaneously, only one state predicate is used for each automaton. Each automaton G_i has its own predicate $\mathbf{x}_i \downarrow$, which identifies a state of each automaton. Note that in the following definitions, only those automata are used that include states that we want to prevent

to be active simultaneously. Furthermore, we assume that the initial state of the resulting automaton is not removed.

Definition 4.2. Consider automata $G_i = (\mathbf{X}_i, \Sigma_i, \xi_i, \mathbf{x}_{0,i}, \mathbf{X}_{m,i})$ and states $\mathbf{x}_i \in \mathbf{X}_i$ for $i = 1, \dots, n$. The automaton that corresponds to the mutual exclusion expression $\neg(\mathbf{x}_1 \downarrow \wedge \dots \wedge \mathbf{x}_n \downarrow)$ is defined by

$$\begin{aligned} &Ac(\mathbf{X}_1 \times \dots \times \mathbf{X}_n - \{(\mathbf{x}_1, \dots, \mathbf{x}_n)\}), \\ &\Sigma_1 \cup \dots \cup \Sigma_n, \\ &\xi', \\ &(\mathbf{x}_{0,1}, \dots, \mathbf{x}_{0,n}), \\ &\mathbf{X}_{m,1} \times \dots \times \mathbf{X}_{m,n} - \{(\mathbf{x}_1, \dots, \mathbf{x}_n)\}, \end{aligned}$$

with ξ' is a restriction of $\xi_1 \times \dots \times \xi_n$ to $\mathbf{X}_1 \times \dots \times \mathbf{X}_n - \{(\mathbf{x}_1, \dots, \mathbf{x}_n)\}$ w.r.t. the domain and the codomain.

Example 4.3.1

Consider two users that are using a shared resource. Each user is modelled with a plant model depicted in Figure 4.2, containing a controllable event *take* and an uncontrollable event *release*. Note that in this example, we have two plant models, each representing a user.

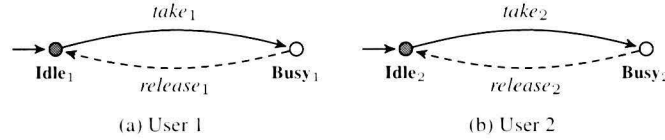


Figure 4.2: Plant models of the mutual state exclusion example

Assume that we want to prevent that both users take the shared resource, i.e. we demand that both users may not be busy at the same time. We can specify this with the following state-based expression: $\neg(\mathbf{Busy}_1 \downarrow \wedge \mathbf{Busy}_2 \downarrow)$, which specifies that the states **Busy**₁ and **Busy**₂ may not be active simultaneously. In Figure 4.3a, the product of both plant automata of Figure 4.2 is depicted. Each state of this automaton product corresponds to a combination of states of the plant automata. In Table 4.3b, the corresponding plant states of both users are given for each state of the automaton product.

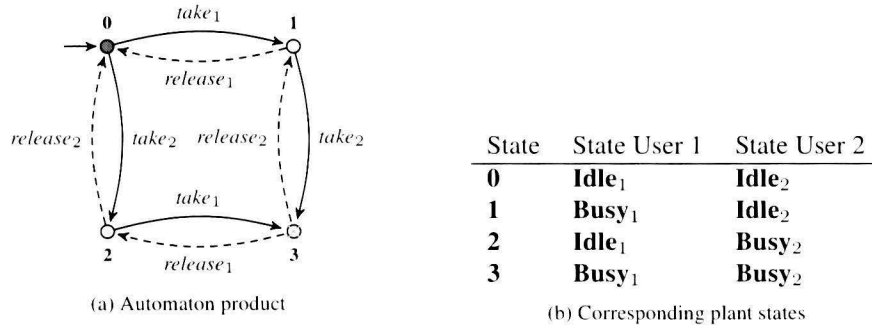


Figure 4.3: Automaton product and corresponding states

State **3** corresponds with the state specified as undesired in the state-based expression, namely that both users are busy. To prevent that this state becomes active, this state and all ingoing and outgoing transitions

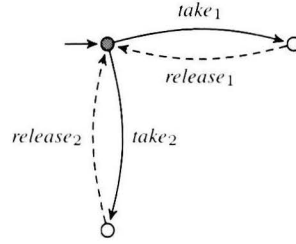


Figure 4.4: Automaton product and resulting requirement automaton

of this state are removed. Since all other states are accessible, no more states or transitions have to be removed. This results in the requirement of Figure 4.4, specified as an automaton.

⊠

With this conversion, it is possible to construct automata that have no marker states. If a supervisor is synthesized with a requirement with no marker states, the synthesis will produce an empty supervisor, since no marker state can be reached at all.

4.3.2 State-transition exclusion

The next state-based expression is a state-transition exclusion, also known as a type 2 specification of the Ma-Wonham tool. This state-based expression states that in a combination of states a certain transition is not allowed.

Definition 4.3. A state-transition exclusion expression over n automata with $i = 1, \dots, n$ and an event e can be written as the following logical expression

$$\left(\bigwedge_{i \in 1..n} \mathbf{x}_i \downarrow \Rightarrow \neg e \right) \quad (4.2)$$

In this subsection, the conversion of this state-based expression to an automaton is stated. This conversion uses also the automaton product. Two different cases of the state-transition exclusion can be distinguished. The first case is when event e , that is not allowed in a certain state set, is already in the alphabet of the automaton product. If this is the case, all outgoing transitions labeled by e have to be removed at the state specified in the state-transition exclusion. The second situation is when the event e , that is not allowed in a certain state set, is not in the alphabet of the automaton product. To capture this expression with an automaton, an extra automaton that contains only one state and a selfloop labeled by e is added to the automaton product. Note that if the event e already is included in the original automaton product, the resulting automaton product will not change. However, if the event is not in the alphabet of the automaton product, the event is added to the alphabet and the automaton product contains selfloops labeled by e in all states. The selfloop labeled by e needs to be removed at the state where the state predicate is true, according to the state-based expression. In the two examples in this subsection, we consider both situations.

Definition 4.4. Consider automata $G_i = (\mathbf{X}_i, \Sigma_i, \xi_i, \mathbf{x}_{0,i}, \mathbf{X}_{m,i})$, and $\mathbf{x}_i \in \mathbf{X}_i$ with $i = 1, \dots, n$, and $\mathbf{x} \in \mathbf{X}_1 \times \dots \times \mathbf{X}_n$, and an automaton $G_e = (\mathbf{X}_e, \Sigma_e, \xi_e, \mathbf{x}_{0,e}, \mathbf{X}_{m,e})$, where $\mathbf{X}_e = \{\mathbf{0}\}$, $\Sigma_e = \{e\}$, $\xi : \xi_e(\mathbf{0}, e) = \mathbf{0}$, $\mathbf{x}_{0,e} = \mathbf{0}$, and $\mathbf{X}_{m,e} = \{\mathbf{0}\}$. The automaton that corresponds to the state-transition

exclusion expression $(\mathbf{x}_1 \downarrow \wedge \dots \wedge \mathbf{x}_n \downarrow \Rightarrow \neg e)$ is defined by

$$\begin{aligned} &Ac(\mathbf{X}_1 \times \dots \times \mathbf{X}_n \times \mathbf{X}_e, \\ &\quad \Sigma_1 \cup \dots \cup \Sigma_n \cup \Sigma_e, \\ &\quad \xi', \\ &\quad (\mathbf{x}_{0,1}, \dots, \mathbf{x}_{0,n}, \mathbf{x}_{0,e}), \\ &\quad \mathbf{X}_{m,1} \times \dots \times \mathbf{X}_{m,n} \times \mathbf{X}_{m,e}), \end{aligned}$$

$$\text{with } \xi'(\mathbf{x}, \sigma) = \begin{cases} \text{undefined} & \text{for } \mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_n) \wedge \sigma = e, \\ (\xi_1 \times \dots \times \xi_n \times \xi_e)(\mathbf{x}, \sigma) & \text{otherwise.} \end{cases}$$

Example 4.3.2

In this example, a state-transition exclusion expression is converted to an automaton which satisfies this expression. Consider the plant model depicted in Figure 4.5a. In this automaton, it is possible to switch on the LED multiple times for a certain reason. Imagine that we want to specify in a requirement that we do not want to switch the LED on if it is already on, e.g. $\mathbf{LED_On} \downarrow \Rightarrow \neg \{ \mathbf{LED_enable} \}$. To construct an event-based requirement, specifying this state-transition exclusion, our intuition is to remove the event $\mathbf{LED_enable}$ at state $\mathbf{LED_On}$.

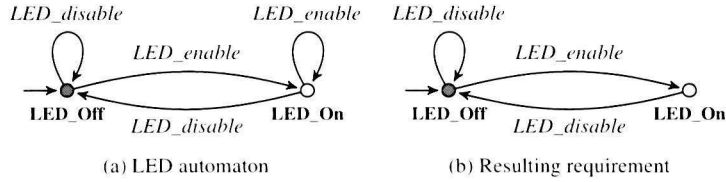


Figure 4.5: A state-transition exclusion example

This is in accordance to Definition 4.4. However, in this example no automaton product needs to be computed, since only one state predicate is specified. Furthermore, the event $\mathbf{LED_enable}$ is already included in the alphabet of the automaton. Therefore, only at the state where the state predicate is true ($\mathbf{LED_On}$), the event $\mathbf{LED_enable}$ needs to be removed. The resulting requirement is depicted in Figure 4.5b. \boxtimes

Example 4.3.3

In this example, a state-transition exclusion expression is converted to an automaton with an event that is not included in the alphabet of automata over which the state predicates are defined. Reconsider Example 4.2.1, where we want to specify that the multimover is not stopping in the forward direction when Ride Control is sending a ‘general start’ command and the short proximity front sensor is inactive. This requirement can be specified with the following state-based expression:

$$\mathbf{RC_Start} \downarrow \wedge \mathbf{PSF_Inactive} \downarrow \Rightarrow \neg \{ \mathbf{dm_fwstop} \}.$$

The plant models that are used are depicted in Appendix A.1.

To construct an automaton that specifies the stated state-transition exclusion, the automaton product of the plant models of Ride Control and the short proximity front sensor is computed, which results in Figure 4.6a. In all states of this automaton, selfloops are added, except for the state where the state formula is true. This results in the automaton, depicted in Figure 4.6b. Note that all states are still accessible and no further adaptations have to be made. \boxtimes

The property of controllability does not have to be taken into account with the construction of automata. Automata can specify whatever the engineer wants. The supervisor synthesis will take care of the controllability property, such that the resulting supervisor is controllable w.r.t. to the plant models and Σ_u

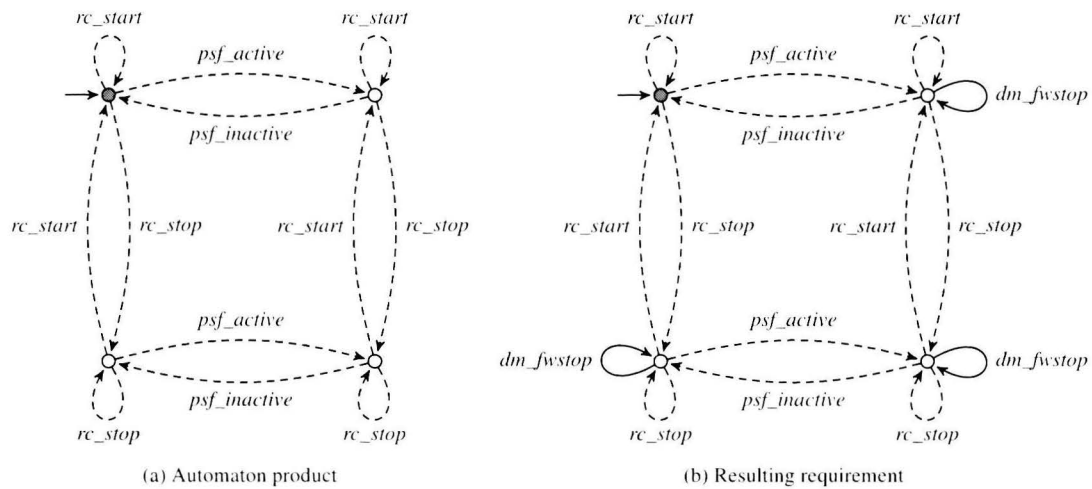


Figure 4.6: Automaton product and resulting requirement

and satisfies all requirements. Therefore, it is also possible to construct requirement automata that disable uncontrollable events in a certain state.

As experience has shown in the last years, often a lot of state-based expressions have to be specified in order to synthesize a satisfactory supervisor for high-tech systems. The conversion that is proposed in the previous section constructs an automaton for every single state-based expression. One can imagine that the construction of a lot of automata can complicate the synthesis of a supervisor, since the event-based synthesis is based on the automaton product. To analyze the performance of the conversion and the synthesis procedure afterwards, an application is built in the programming language Python, that automatically constructs automata out of state-based expressions. A more detailed explanation and the source code of this application is given in Appendix B. Here only the syntax and the results are given.

The conversion is built in the Supervisor Synthesis Package as two functions, that make use of the already existing functions. The first function constructs a requirement with a mutual state exclusion. The syntax is given below. The function needs three arguments, namely 1) the names of the automata that are needed, 2) the state set (between curly brackets) that needs to be excluded, and 3) the name of the resulting automaton.

```
frontend.make_type1_specification('Input_1.cfg', ..., 'Input_n.cfg', '{x_1, ..., x_n}', 'Output.cfg')
```

The second function that is built is to construct automata out of state-transition exclusion expressions. The syntax is given below. The function needs three arguments, namely 1) the names of the automata that are needed to construct the automaton product 2) the name of the automaton containing the event, that has to be excluded, 3) the state set (between curly brackets) and the name of the event that needs to be excluded, and 4) the name of the resulting automaton. Note that the syntax of both state-based expressions is based on syntax of the synthesis tool of Ma-Wonham.

```
frontend.make_type1_specification('Input_1.cfg', ..., 'Input_n.cfg', 'Input_e.cfg', '{(x_1, ..., x_n), e}', 'Output.cfg')
```

The application has been tested with testcases in order to check if the right automata are produced as output. After this, the logical expressions that are used for synthesizing a state-based supervisor for the multimover are converted to automata, to check whether it is possible to synthesize a event-based supervisor with automata and state-based expressions, which are converted to automata. More details and the results of this experiment are discussed in the next section.

4.4 Experiment

In order to check the feasibility of synthesizing event-based supervisors with requirements specified by state-based expressions, the state-based model of the multimover is used. First, the logical specifications are converted with the conversion tool of [Jac09] to type 1 and type 2 expressions for the Ma-Wonham tool. After this, all type 1 and type 2 expressions are converted to automata, using the event-based plant models. Note that the plant models of the synthesis tool of Ma-Wonham and the Supervisor Synthesis Package need to be equivalent, in order to perform a correct conversion. In Table 4.1, the numbers of automata that are generated with the conversion application for each module is stated. Furthermore, the numbers of states and transitions are stated of the automaton product of all converted automata for each module.

Table 4.1: Results of the conversion

Module	# logical expr.	# automata	automaton product	
			# states	# transitions
LED actuation	6	9	4	15
Motor actuation	8	35	29	163
Button handling	2	4	9	31
Emergency handling	1	4	5	11
Proximity handling	8	14	33	257

As we can see, the state space of the resulting automata is relatively small. This is due to the fact that often more or less the same requirement is constructed, only with a different disabled event in a different state. As a result, the complete automaton product synchronizes most of the transitions and only a certain event is disabled. To illustrate this, the computation of the automaton product of the state-based requirements of the motor actuation module is discussed more thoroughly.

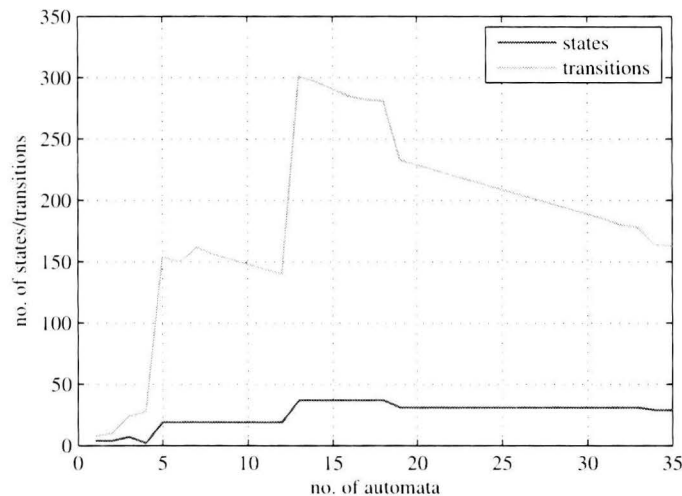


Figure 4.7: Numbers of states and transitions during computation of the automaton product

In Figure 4.7, the numbers of states and transitions are depicted during computation of the automaton product. The automaton product works sequentially, i.e. first the automaton product of the first automaton

and the second automaton is computed. Then, the automaton product is computed of the result of the previous automaton product with the third automaton etc. As we can see, in the beginning, the number of states and transitions of the automaton product is increasing, like expected. After this, the number of transitions is gradually decreasing, due to the removal of transitions. The number of states is staying more or less constant, since the converted state-based expressions are state-transition exclusion expressions, which exclude a *transition* and no states. After addition of automaton 19 and 34 to the automaton product, the number of states have decreased. This is due the fact that a certain part of the state space of the automaton has become not accessible anymore. As a result, this part of the automaton is removed.

An event-based distributed supervisor has been synthesized with these requirement automata. This distributed supervisor is validated by means of discrete-event stepping and the supervisor has the expected behaviour. No differences between the simulation results of the state-based supervisor and the event-based supervisor were encountered.

4.5 Discussion

In the previous section, the results are shown for an example of conversion from logical expressions to automata, that subsequently are used to synthesize an event-based supervisor. As we can see, a satisfactory supervisor is synthesized with more modelling convenience. Furthermore, the state space did not explode, despite of the fact that a lot of automata are generated. One has to bear in mind that this algorithm of converting logical expressions to automata is not optimal. A direct conversion of logical expressions to automata (without conversion to the type 1 and type 2 expressions of Ma-Wonham) might be more efficient. In the case study worked out in this project, a more efficient conversion of logical expressions to automata was not necessary.

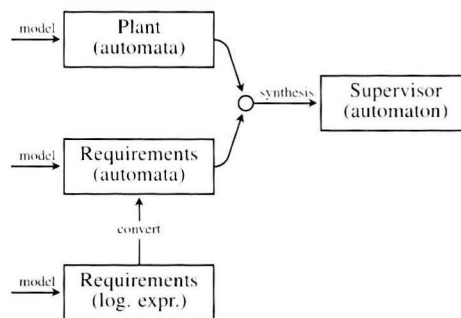


Figure 4.8: The event-based framework extended with logical expressions

In Figure 4.8, a graphical overview is given of the conversion of logical specifications to automata. With this conversion of state-based expressions to automata, we are now able to specify requirements with automata *and* logical expressions. To investigate the applicability of this conversion, it is recommended to apply it in other industrial cases.

Logical expressions provide a powerful method for specifying the exclusion of certain situations. However, logical expressions are not suitable to memorize certain sequences of events. This can be done by automata. Together, they form a sufficient modelling framework for specifying the requirements of the multimover. [Seo07] proposes to specify state-based specifications by temporal logic. Temporal logic is known to be a richer language than propositional logic and can be, as a result, more suitable to specify the control requirements of complex high-tech systems. This can be investigated in further research.

In this chapter, we have discussed both supervisor frameworks more thoroughly. We have seen that both frameworks have their advantages and disadvantages for synthesizing supervisors. Furthermore, the state-based framework is often recognized as more convenient for modelling requirements, since requirements can not only be modelled by automata, but also by logical expressions. However, in this chapter is shown

that logical expressions can be converted to automata to synthesize event-based supervisors. The next chapter describes an implementation of the supervisors of both frameworks in the current control software of the multimover.

Chapter 5

Implementation

In the previous chapters, supervisory control theory has been explained and used to synthesize supervisors for the multimover. Both the event-based framework and the state-based framework have been used to synthesize a supervisor. The behaviours of the supervisors have been validated by means of discrete-event simulation. The next step in the synthesis-based engineering process of Figure 5.1 is to implement the supervisor. This chapter describes the implementation of both supervisors in the current control software of the multimover.

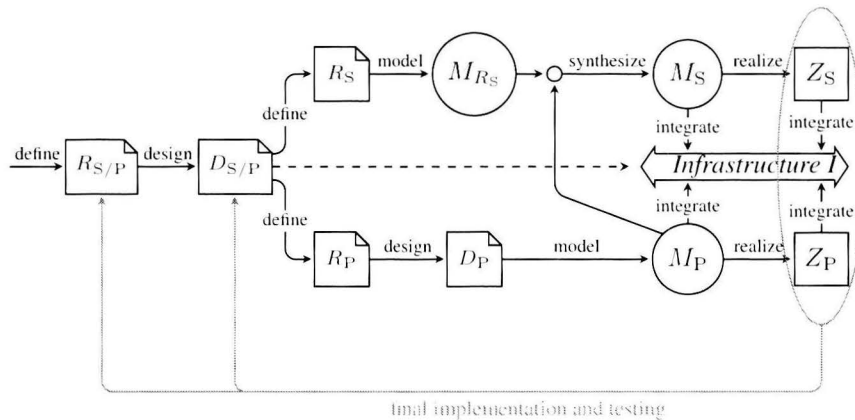


Figure 5.1: Implementation in the synthesis-based engineering framework

In Section 5.1, the differences between a controller and a supervisor are stated and a set-up for implementation of a supervisor is proposed. Then, some implementation aspects are given in Section 5.2. Subsequently, a prototype of this implementation is described in Section 5.3. A description of the validation of the implementation and experiments are described in Section 5.4 and Section 5.5, respectively. Lastly, this chapter ends with evaluation of the supervisor implementation and the applicability in Section 5.6.

5.1 From supervisor to controller

The control problem for a generic dynamical system consists of influencing the behaviour of a system, in order to satisfy given specifications. This is schematically shown in Figure 5.2a. This is achieved by designing an appropriate control unit that uses information from the plant to influence this through the available control mechanism [Bal92b].

In the original supervisory control framework, a supervisor acts as a passive device that tracks events produced by the plant and restricts the behaviour of the plant by disabling the controllable events [Bal92a]. This is schematically shown in Figure 5.2b. However, it is often the case that the plant does not generate all controllable events on its own without being initiated. Normally, simple machines do not start their work unless the start command is given. In this case, it is desirable to have a controller which not only disables controllable events but also initiates the occurrence of particular controllable events [Die02]. Furthermore, supervisory control theory is based on the assumption that the supervisor is always synchronized with the state of the plant, i.e. there is no communication delay. However, in contrast to the synchronous communication used in models, real systems often use asynchronous communication [Bra08].

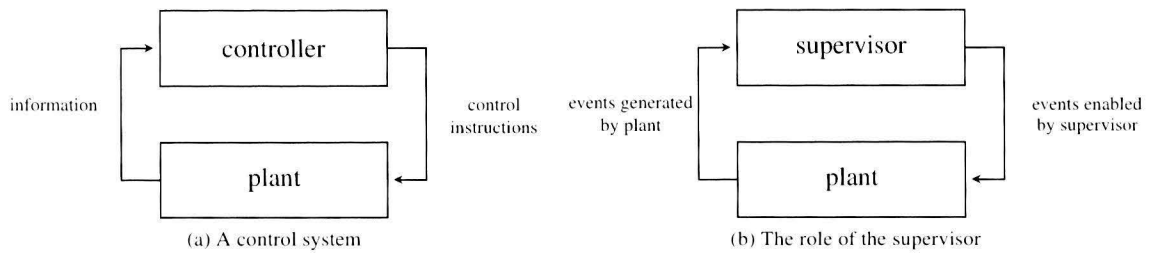


Figure 5.2: The communication between the supervisor S and the uncontrolled plant P

So, a supervisor is not directly a controller, but can be seen as a dictionary of allowed events at each state of the plant. This can be compared with solving a game of chess, where all allowed moves are listed in a lookup table. From any position, the next move can be carried out by searching the lookup table instead of calculating the possible moves [Ber09]. In this section, the implementation of a controller is explained that tracks the state of the plant and sends appropriate control actions back. We refer to this controller as a supervisory controller.

The functionality of a supervisory controller can be roughly divided in two tasks. The supervisory controller needs to track the state of the plant in order to give appropriate feedback to the plant. We call this part of the controller the *state tracker*. Next, the controller is responsible for sending appropriate control actions back to the plant based on the state of the plant. We refer to this part of the supervisory controller as the *control decision maker*. In Figure 5.3, a schematic overview of a supervisory controller is given. In this figure, we can distinguish the plant which represents the components and the low-level resource control, and a supervisory controller (in red). This supervisory controller contains a state tracker which tracks the state, a control decision maker which sends appropriate actions back to the plant and a supervisor which contains all allowed behaviour.

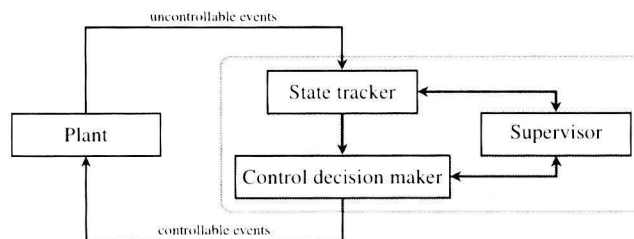


Figure 5.3: A supervisory controller

At some point, the plant will generate an event (e.g. a button is pressed, a sensor is activated etc.). A notification has to be sent to the state tracker, which updates the current state of the supervisor. This is done by looking in the supervisor what the new current state is. Note that that only uncontrollable events and no controllable events are tracked by the state tracker, since the supervisory controller has control over the controllable events. If the state tracker is finished with updating the current state of the supervisor, the

control decision maker has to search for an appropriate control action that can be sent back to the plant (e.g. turn the LED on, turn off the motor etc.). Note that we assume that only controllable events are initiated by the control decision maker. If an appropriate control action is found, this action is carried out and the current state of the supervisor is updated again.

5.2 Implementation aspects

In literature, some information is available about problems that can occur if a controller is derived from a model of a supervisor. These problems are discussed in [Mal03, Mal02, Die02].

5.2.1 Command selection problem

The supervisor synthesized with supervisory control theory is nonblocking according to Definition 2.6. However, there is no guarantee that an implementation of the supervisor is also nonblocking. Even worse, the resulting controller may be blocking due to ‘bad choices’ [Mal03, Mal02, Die02]. This can be illustrated by the example shown in [Mor07].

Example 5.2.1

Consider a system of two machines that can be used for two tasks. One machine can work on task A and the other machine can work on task B. These are controllable events: a supervisor may disable them. The events f_A and f_B model the completion of the task. These are uncontrollable events: the supervisor cannot influence the occurrence of these events. If both machines work at the same time, the system breaks down. In Figure 5.4a, a representation is given of this system. Supervisory control theory can prevent this system from a breakdown, by disabling the controllable events to the state **Down**. The resulting supervisor is given in Figure 5.4b. The desired controller is a realization of the automaton model in Figure 5.4b.

At the initial state **Idle**, a choice has to be made which machine should be started next. However, when a supervisory controller is implemented that has to select one of the signals $start_A$ and $start_B$, we could get into trouble; if the controller always selects $start_A$, the marked state **Task_B** is never reached. This results in violation of the nonblocking property, since the marker state can never be reached (see Figure 5.4c).

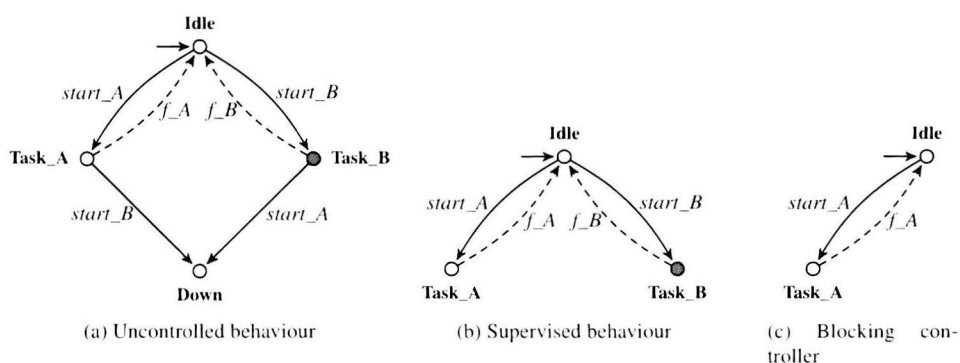


Figure 5.4: Generation of a blocking supervisory controller

⊠

Another problem that could occur when a supervisor is implemented is that the implemented supervisor could end up in livelock, due to a wrong choice of events. Livelock is an endless loop of occurrences of events without gaining any progress. This problem could occur when an infinite sequence of controllable events is possible from the current state. This problem is illustrated by the following example.

Example 5.2.2

An automaton is depicted in Figure 5.5. Consider this automaton as a part of the implemented supervisor. Note that all events of this example are controllable and implemented as control actions. State **2** is marked and can be considered as completion of a certain task. Note that all states of this automaton are nonblocking according to Definition 2.6, since from every state a marker state can be reached. If the current state of the supervisor implementation is state **0**, the supervisory controller has a choice: either event *a* or *b* could be sent. If in this state *a* is always chosen, we have livelock, since marker state **2** is never reached and an infinite sequence of events (*a,b,a,b,...*) is chosen.

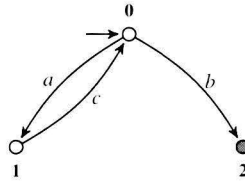


Figure 5.5: An implementation containing livelock

⊠

Thus, an implementation of a supervisor cannot guarantee that a certain marker state is eventually reached. This problem, stated in the previous examples, is called the *command selection problem*. To avoid this problem, [Mal03] proposes checking algorithms that can be used to ensure a nonblocking controller. Three new properties of discrete event systems are introduced. Petra Malik proved that if these properties hold for an automaton, the resulting controller is nonblocking and the *command selection problem* is solved. In this project, discrete-event simulation is used for validating the behaviour of the supervisor. However, simulation cannot prove these properties, which means that the implementation might still be blocking.

In [Mor07], an algorithm is presented that can synthesize deterministic controllers out of the conventional plant and requirement models. However, only centralized controllers are taken into account and therefore not applicable in our case. Furthermore, a synthesis method is developed in [Su09a], that synthesizes centralized supervisors which achieve also a time-optimal performance. One can imagine that an implementation without reachable marker states (Example 5.2.1) or containing livelock (Example 5.2.2) is not time-optimal, since no progress is made. To this end, the synthesis of time-optimal supervisors for real industrial systems is recommended as future research.

5.2.2 Communication problem

The communication problem is another problem related to building controllers from supervisory control models [Mal03]. This problem occurs when the controller sends a control action to the plant, but in the meantime, the state of the plant is changed. This means that a control action is chosen based on an old state of the plant.

The reason why this situation can occur, is that communication between the plant and the controller in the real system is not synchronous. Let us investigate where the time delays are in our closed loop system. We can distinguish three delays:

- **Input delay.** This delay is the time period that is needed from the time instance that an event occurs in the plant until the time instance that the supervisory controller has received this message.
- **Computation delay.** This delay is the time period that is needed from the time instance that the state tracker is updating the current state of the supervisor until the time instance that an appropriate control action is chosen by the control decision maker.

- **Output delay.** This delay is the time period that is needed from the time instance that a message is sent by the control decision maker to the plant until the time instance that the corresponding control action is carried out by the plant.

To investigate the delays further into detail, a model is made in the specification language χ [Bee06] that simulates a closed-loop system with asynchronous communication behaviour. In this χ model, the asynchronous communication is modelled as a buffer between the plant and the supervisor, which delays all messages for a certain time period. Since only the concept of asynchronous communication needs to be modelled, only one buffer is used. The χ model is based on the following assumptions.

Assumption 5.1. Events that are generated by the plant and related messages that are sent to the supervisory controller cannot get lost and cannot overtake each other. The same assumption applies for the communication from the supervisory controller and the plant.

Consider the plant model of a timer in Figure 5.6a, which represents a timer that can be started (*start*) and reset (*reset*). If the timer is expired, the uncontrollable event *timeout* occurs. The controllable events *start* and *reset* are considered as output of the supervisory controller and the uncontrollable event *timeout* as input of the supervisory controller.

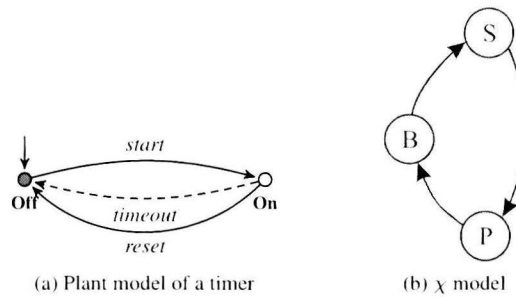


Figure 5.6: Communication delay example

The closed-loop system is modelled in χ as depicted in Figure 5.6b. The timer model is represented by χ process P, while the supervisory controller is represented by χ process S. The communication delay is modelled with a buffer B between the timer P and the supervisory controller S. In this model, messages that are sent from the timer P to the supervisory controller S have to wait for a certain time in the buffer B before they are sent to the supervisory controller S. The χ model is listed in Specification D.1 in Appendix D.1.

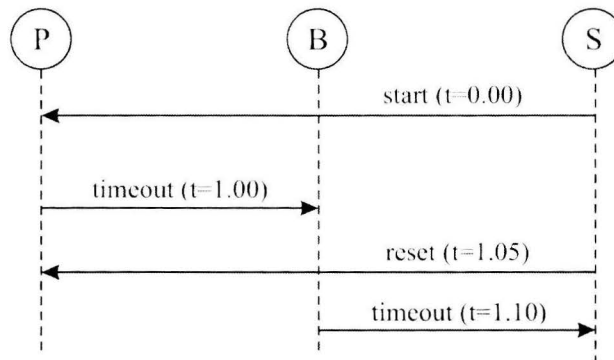


Figure 5.7: Results of simulation

The message sequence chart, depicted in Figure 5.7, shows the behaviour of the χ -model. At $t = 0.00$, the timer is started. It will expire at $t = 1.00$. When the timer is expired, the timer sends a message to the supervisory controller that the event has happened, but due to a communication delay, this is received by the supervisor at $t = 1.10$. In the meantime, a message is sent by the supervisory controller to the plant that the event *reset* takes place at $t = 1.05$ in order to reset the timer. So, the supervisory controller sends a control action to the plant, based on an old state of the plant, since the supervisory controller ‘thinks’ that the timer is not expired and is in state **On**, while the timer is already expired and is in state **Off**.

The sequence of events that is observed by the plant and by the supervisory controller is not captured by the original plant model, since the timer observes the events *start - timeout - reset* and the supervisory controller observes *start - reset - timeout*. It is unclear how both the timer and the supervisory controller will react on these observations. A simple verification approach is suggested in [Mal02] in order to identify a class of plants that are robust with respect to the communication problem. This verification approach can be applied very easily.

Definition 5.2. Let $G = (\mathbf{X}, \Sigma, \xi, \mathbf{x}_0, \mathbf{X}_m)$ be a deterministic automaton. The automaton G is Σ_c - Σ_u -commuting, if for all $\mathbf{x} \in \mathbf{X}$, $\sigma_c \in \Sigma_c$, and $\sigma_u \in \Sigma_u$ such that, if both σ_c and σ_u is an accepting event at a given state $\mathbf{x} \in \mathbf{X}$, we have that $\xi(\mathbf{x}, \sigma_c \sigma_u)$ and $\xi(\mathbf{x}, \sigma_u \sigma_c)$ are both defined and $\xi(\mathbf{x}, \sigma_c \sigma_u) = \xi(\mathbf{x}, \sigma_u \sigma_c)$.

A Σ_c - Σ_u -commuting plant accepts a controllable event σ_c and an uncontrollable event σ_u in any order whenever both are accepted at a certain state. Furthermore, the order does not influence the future behaviour of the system. If a plant is Σ_c - Σ_u -commuting, the following properties are satisfied.

- The plant accepts any message from a supervisory controller that only sends messages based on the current state of the state tracker, even if it is delayed.
- Each supervisory controller, generated from a model containing the plant, accepts any message from the plant, even if it is delayed.
- If all messages are received, i.e. no messages are pending in the network, the state tracker can tell the state of the plant.

We can easily verify that the plant model of the timer is not Σ_c - Σ_u -commuting. The controllable event *reset* and the uncontrollable event *timeout* are accepted in the state **On**. However, the sequence of events *timeout - reset* or *reset - timeout* is not accepted in state **On**.

5.3 Prototype implementation

In order to prove the concept of synthesis-based engineering, a prototype of a supervisory controller with the synthesized supervisors is implemented in the existing control software of the multimover. For flexibility, an implementation of both supervisors is proposed. A schematic overview of the control architecture of the multimover with a supervisory controller is given in Figure 5.8. Note that, in order to implement a supervisory controller, first the existing control software that has the same functionality needs to be removed.

At the bottom of this figure, we can see all components and their resource controllers. Above the resource control, an interface is made which is responsible for sending the correct events from the resource control to the supervisory controller and sending the correct events from the supervisory controller to the resource controllers. This interface makes use of a listener and notifier structure. This is a simple communication paradigm. The resource control of each component can publish messages of a certain topic and can subscribe to a certain topic, which means that they will receive all published messages of that topic. So, the interface is subscribed to all relevant events and will receive them. This interface has to be coded manually, since it is different for every system.

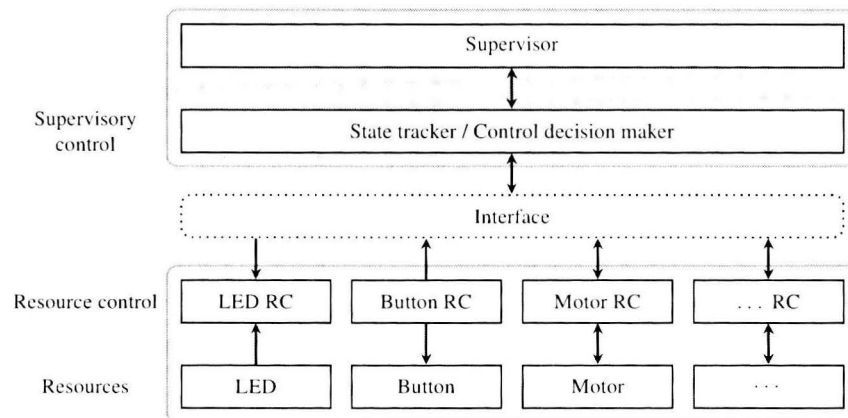


Figure 5.8: Control architecture of the multimover with a supervisory controller

The next layer in Figure 5.8 is an implementation of a supervisory controller, which contains a state tracker and control decision maker. This layer is written in such a way that it is independent of the supervisor model, which supervisory control framework is used and the system itself. A general implementation of a supervisory controller is written in pseudo-code in Algorithm 1. As already told, the functionality of the supervisory controller can be divided into two tasks, namely tracking the state of the system by the state tracker (line 3 to 7) and making appropriate control decisions by the control decision maker (line 9 to 19).

Algorithm 1 Concept of supervisor implementation

```

loop
  // State Tracker
  while len(list) > 0 do
     $E \leftarrow \text{pop}(\text{list})$ 
    5: UpdateSupervisor( $E$ )
     $S \leftarrow 1$ 
  end while
  // Control Decision Maker
  if  $S = 1$  then
    10:  $E \leftarrow \text{ComputeControlAction}$ 
    if  $E \neq 0$  then
      if len(list) = 0 then
        UpdateSupervisor( $E$ )
        ExecuteEvent( $E$ )
    15: end if
    else
       $S \leftarrow 0$ 
    end if
  end if
  20: end loop

```

All (uncontrollable) events that are generated by the plant (e.g. button and sensor signals) are listed in a buffer by the interface, which is a different process that also has access to this buffer. This buffer is emptied by the state tracker by taking and removing the first element of the buffer ($E \leftarrow \text{pop}(\text{list})$) and subsequently, the current state of the supervisor is updated (line 5, UpdateSupervisor(S)). If the list is empty, the state tracker knows the current state of the system. Based on this current state of the supervisor, a control decision can be calculated.

If the current state of the supervisor has changed ($S \leftarrow 1$), the control decision maker has to check if a

control action is possible. This is done by setting a boolean variable S to 1 (line 6), which activates the control decision maker. First, a control decision is computed. If an appropriate control action is found (line 11, $E \neq 0$), the event list has to be checked (line 12), to ensure that the supervisory controller has made an appropriate control action based on the most actual state of the plant. If this the case, the state of the supervisor is updated and the appropriate control action is executed. Note that this implementation does not prevent the execution of a control action based on an old state of the supervisor. The communication problem can still occur.

If no control action is possible (e.g. all controllable events are disabled by the supervisor), there is no need to search for a control action over and over again. So, the boolean variable S is set to 0 (line 17), which means the control decision maker is not executed anymore. If the state of the system changes again due to the occurrence of an uncontrollable event, the control decision maker is activated again.

The next layer in Figure 5.8 is the supervisor itself, which contains the information about the allowed behaviour, according to the requirements. This information can be generated from the model of the supervisor. This is done by a script in Python (see Appendix D.3), that reads the information from a CIF model and stores this information in a lookup table. A lookup table is used for this information, since a lookup table can be used with an efficient indexing operation, which could save in terms of processing time. An explanation of how an automaton is converted to a lookup table is mentioned in Appendix D.2.

The prototype implementation described above is suitable for supervisors of both frameworks, either event-based or state-based. However, there are some differences with respect how the state is tracked and the control decisions are made for both frameworks. These differences are stated below. A summary of the differences between the implementation of the two supervisor types, synthesized with either one of both frameworks is listed in Table 5.1.

5.3.1 Event-based implementation

A supervisor that is synthesized with the event-based framework contains the complete allowed language of the closed-loop system, as mentioned in Section 2.4.3. This is stored in one or more automata. The state is tracked by updating the current states of the automata if an event occurred. An automaton is only updated if the event that has occurred is also in the language of this automaton. If an event occurs that is not allowed by automata, then the model is inadequate, since the state tracker cannot track the state of the system. If this happens, the supervisory controller and all components are switched off.

Control decisions are calculated by searching for controllable events that are allowed by all automata. The first controllable event that is found and allowed by all automata is chosen as the control action.

5.3.2 State-based implementation

A supervisor that is synthesized with the state-based framework uses automata and BDDs to store the state feedback control (SFBC) map in. The automata are used to store the information when each controllable event is allowed by the plant models and the BDDs are used to store the information when each controllable event is allowed by the state-based expressions.

A state-based implementation uses the plant models and event-based requirements to track the state of the system. All automata of the plant models and event-based requirements are updated if an uncontrollable event occurs. If an uncontrollable event occurs that is not allowed by an automaton, the state of the system cannot be tracked and the model of the supervisor is inadequate. If this happens, the supervisory controller and all components are switched off.

Control decisions are calculated by searching for a controllable event that is allowed by all automata *and* its BDD. The first controllable event that is allowed by all automata and its BDD is used as a control action.

Table 5.1: Difference in implementation for supervisors of both frameworks

Implementation	State tracker	Control decision maker
Event-based supervisor	Update supervisor automata	Search for controllable event allowed by all supervisor automata
State-based supervisor	Update automata	Search for controllable event allowed by all automata and its BDD

5.4 Validation of implementation

In the previous section, a concept of implementation of supervisors is discussed. This concept is implemented in the existing control software and validated. The test set-up is depicted in Figure 5.9. First of all, only a state tracker was implemented and tested if the state tracker could update automata according to the uncontrollable events that were generated. After this was validated, the control decision maker was implemented and validated if the right control decisions were chosen at the right moments. A small event-based supervisor was synthesized with two buttons and one LED as the plant, in order to validate the state tracker and the control decision maker.

Subsequently, support for BDDs was implemented to implement supervisors of the state-based framework. Only the control decision maker had to be validated again, since the BDDs are only used by the control decision maker. A small state-based supervisor was synthesized with two buttons and one LED as the plant in order to validate the implementation of the support for BDDs.

After the complete supervisory controller was validated, a script is made that converts the CIF models to lookup tables. This conversion is validated by converting small automata to lookup tables. The lookup tables that were generated, were according to the expectations. After this, both supervisors with two buttons and one LED as the plant, that were implemented manually in the multimover, were generated from CIF models. This was according the expectations, so the complete tool chain and implementation is validated for these testcases.

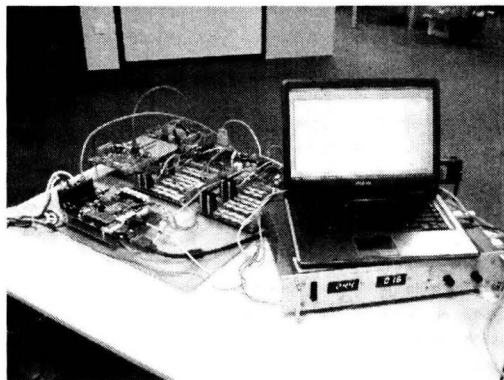


Figure 5.9: A photo impression of the test set-up

5.5 Experiment

After the implementation and the conversion from the CIF-model to C was validated, the supervisors of the multimover were implemented. First, the supervisor, synthesized with the event-based framework, was

implemented. This implementation is tested by pushing buttons and activating and deactivating sensors. With this implementation testing, a lot of situations were tested and validated. All relevant situations were tested exhaustively, in order to validate the error handling, proximity handling and emergency handling. We encountered that the multimover behaved the same as in the model in the supervisor. With these tests, the conclusion is drawn that the event-based implementation and the model of the supervisor works correctly. Subsequently, the supervisor synthesized with the state-based framework was implemented in the supervisory controller. The multimover had the same behaviour as with the event-based model. The conclusion is drawn that the state-based implementation works correctly.

After this, the control software is used on a real vehicle. The same tests were performed as on the test set-up. The outcome of these tests were the same as on the test set-up, which is shown in Figure 5.9. The supervisory controller behaves the same as the model. The conclusion is drawn that the supervisory controller and the supervisor models are correct.

In the previous subsections, two problems are addressed that can occur when a supervisory controller is implemented, derived from a supervisor model. First of all, the command selection problem is addressed in our implementation. Our implementation is verified for the command selection problem by means of simulation. Discrete event simulation is used to check for infinite sequences of controllable events. If an infinite sequence of controllable events is found, the requirement models were adapted. Simulation, however, can only show the presence of these infinite sequences, but cannot prove the absence of them. This means that the supervisory controller could still contain livelock. A recommendation is to check for this property mathematically with a model checker, such that the absence of these infinite sequences can be proven.

Secondly, the communication problem, described in Section 5.2.2 did not occur when the tests were performed. This can be explained by the fact that the calculation of a control action of the multimover is much faster than the reaction speed of the plant. The probability that a control action is computed based on an 'old' state of the plant is almost nil. However, theoretically, the communication problem can still occur and remains a topic of future research.

5.6 Synthesis-based engineering: evaluation

With the concept of implementation described in this chapter, we have performed all steps of the synthesis-based engineering framework. In this section, we discuss the modelling convenience and the applicability of all relevant steps of synthesis-based engineering, namely the supervisor synthesis and supervisor validation, as well as the implementation.

5.6.1 Modelling convenience

Plant models can only be defined by automata. We have experienced that this way of modelling components is quite intuitive and straight-forward. Furthermore, components and their resource controllers can often be reused. As a result, plant models that are made of these components can also be reused, which can reduce time in the product development process.

In the beginning of this project, it appeared that specifying requirements by automata may not always be intuitive. Since requirements can also be modelled in the state-based framework by logical expressions, often this framework is chosen for synthesizing supervisors for industrial systems. The extension defined in [Jac09] has made the modelling of requirements more convenient, since often a lot of state-based expressions of the synthesis tool of Ma-Wonham were needed for a satisfactory supervisory control problem definition. In this project, both a state-based supervisor and an event-based supervisor are synthesized using requirements specified by logical expressions and automata. Further research needs to be done if other control problems of NBG can also be modelled intuitively with automata and logical expressions.

Requirements have to be defined in terms of behaviour, instead of in terms of software code, which is done in traditional engineering. It appeared that modelling requirements with logical expressions and automata is intuitive and easy to understand for engineers that do not have affinity with the control software. This can lead to an easier validation with respect to the original informal specifications.

However, one has to take into account that modelling skills are needed for modelling components by automata and requirements by automata and logical expressions. Time is needed to develop those skills. Furthermore, a manual with modelling guidelines, and modelling tools that can be integrated in the existing software development tools, can enhance the modelling process and speed up further acceptance.

5.6.2 Supervisor synthesis and validation

To address the opportunities of synthesis-based engineering, we compare the software engineering of NBG with and without supervisory control synthesis. In Figure 5.10, the traditional software engineering process and the synthesis-based software engineering process is depicted. Please note that this comparison is only applicable for supervisory control software. In this figure, informal steps are denoted with \uparrow and formal steps with $=$. In traditional software engineering (see Figure 5.10a), documents R are used for specifying the requirements of the supervisory control software. Then, an informal design D of the software is made and subsequently, the realization Z is made.

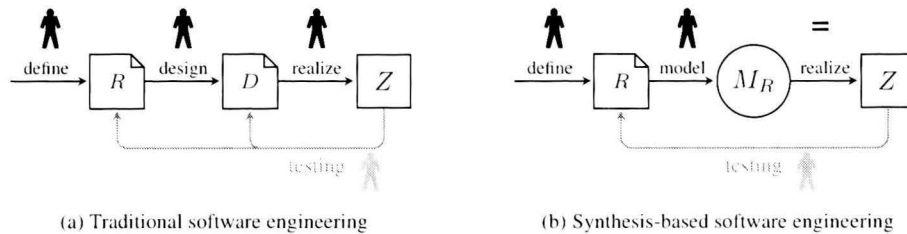


Figure 5.10: Traditional software engineering and synthesis-based software engineering

However, what in fact is done in traditional software engineering is that an engineering problem (the document with requirements R) is translated informally into another engineering problem (the document with a software design D). Subsequently, the last engineering problem is solved by realizing the software design. As a consequence of informally designing and realizing software, the realization needs to be tested against the requirements R and the design D .

With synthesis-based engineering (see Figure 5.10b), the manual design of a certain part of the software is eliminated. Now, the informal document of requirements R is translated into a formal model of the requirements M_R . Synthesis-based engineering allows us to generate control software that is correct with respect to the formal models of the requirements. As a consequence, the realization Z does not need to be tested against the formal model of the requirements M_R . Note that this is only correct if the realization Z is derived correctly out of the model of the supervisor.

Both frameworks are used in order to synthesize a supervisor for the case study. It appeared that the state-based framework is slightly more convenient for synthesizing supervisors, since the control problem does not need to be partitioned in order to avoid the state-space explosion, which can happen when supervisors are synthesized in the event-based framework. However, distributed supervisors of the event-based framework are likely to solve control problems for systems with even larger state spaces, since the state-based framework is essentially a centralized approach.

Next, the supervisor synthesis has the advantage that the synthesized supervisors can be simulated immediately. As a result, the engineer can get feedback immediately and the design-validate-redesign loop is

shortened. Furthermore, the use of models allows application of model-based techniques, used for thorough system analysis and testing, which help to get a better system overview. As a consequence, the use of models can possibly reduce the cost of the production of expensive prototypes.

Furthermore, synthesis-based engineering offers advantages for products that are evolving over time. In general, if the requirements of a system change, the formal requirements are likely to be adapted more easily than a part of a software design. Subsequently, a new supervisor can be synthesized immediately, without losing the consistency between the requirements and the realization, which is often a challenge in traditional engineering. The multimover is an example of such an evolving product, since most theme parks have their own specifications of their theme park vehicle.

5.6.3 Implementation

In this project, a prototype implementation is developed that can be used at NBG for many other applications. It supports supervisors synthesized with either one of both frameworks. This is done in order to give the engineer freedom in choosing which framework is more suitable for the control problem. The prototype implementation performs satisfactorily and evaluates changes in the system's state fast enough. As a result, this concept supervisor implementation is suitable for implementation in other systems.

A difference that is encountered between the implementation of the event-based supervisor and the state-based supervisor is the memory size needed for the controller. The event-based supervisor needs much more memory than the state-based supervisor. This can be explained by the fact that an event-based supervisor contains the complete closed-loop language of a system, as stated in Section 2.4.3, while a state-based supervisor only needs the state of the system in order to calculate a control action. However, the implementation of the event-based supervisor makes more use of lookup tables, which results in a significant faster evaluation of the supervisors compared to the implementation of the state-based supervisors.

Synthesis-based engineering is applicable for control problems that coordinate components. The safety issue of the multimover is an example of such a control problem, since safety is assured by coordination of all relevant components. However, the implementation takes a nondeterministic choice which control action is executed, if more are possible. As a result, the implementation can guarantee that something will certainly not happen, but cannot guarantee that something will happen eventually, since the implementation can still contain livelock, due to a bad choice of control actions. Therefore, the implementation of optimal supervisors in terms of time could be a next step and is suggested as further research.

Furthermore, this prototype implementation is suitable for model-based integration and testing, introduced in [Bra08]. Figure 5.11 shows possible combinations of model-based integration. Figure 5.11a shows the integration of the supervisory controller in the modelling environment. With this set-up, the implementation of the supervisor in the control software can be tested more thoroughly. This set-up is suitable for evaluating the choices that are made by the supervisory controller if more than one control action is possible and which sequences of control actions are produced at each state. Furthermore, with this set-up, livelock can be encountered in an early stage of the product development process. If the real plant is integrated in the modelling environment (see Figure 5.11b), the interactions with the real plant can be simulated with the model of the supervisor. Note that this set-up also allows that only a part of the real plant is integrated.

This section concludes this chapter about the implementation of supervisors. In this chapter, a concept of implementation of a supervisor in the existing control environment of the multimover is described. To prove the concept, an implementation of a supervisory controller is made, which is independent of the used framework. Pitfalls of this concept of implementation are addressed and the conclusion is drawn that the supervisory controller works correctly. In the next chapter, the conclusions and suggestions for further research are given.

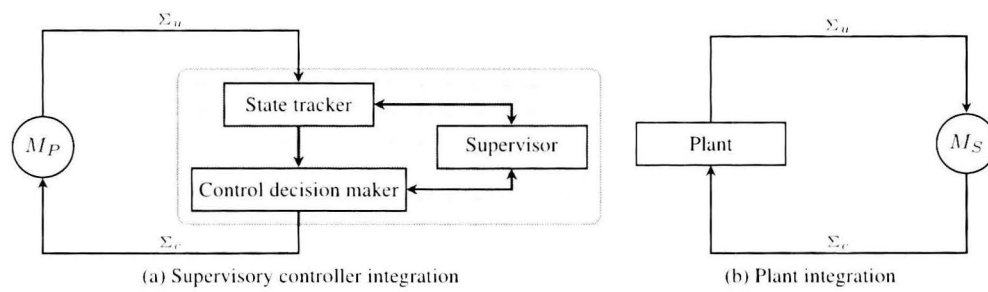


Figure 5.11: Model-based integration and testing

Chapter 6

Conclusions and suggestions for further research

In the previous chapters, all steps to integrate supervisory control synthesis in the engineering process of NBG are discussed. In this chapter, the conclusions and suggestions for further research are presented. The conclusions of the previous chapters are presented in Section 6.1. This chapter ends with recommendations for further research in Section 6.2.

6.1 Conclusions

The objective of this project is to test the applicability of supervisory control synthesis to the product development process of NBG. To this end, a real product of NBG is chosen as a case study. This case study is the multimover, an automated guided vehicle that is used in the amusement park business. For this study, a supervisor for the multimover is synthesized that assures safety, which includes anticipating on emergency and error signals and an accurate proximity handling.

Two main frameworks exist that can be used in order to synthesize supervisors, namely the event-based framework of Ramadge and Wonham [Ram87, Won84] and the state-based framework of Ma and Wonham [Ma05]. Both frameworks are used to synthesize a supervisor for the multimover. Furthermore, the synthesis aspects and modelling aspects of both frameworks have been discussed.

The synthesis of supervisors with the event-based framework often suffers from calculation complexity. As a consequence, the supervisor synthesis of the event-based framework may result in a so-called state-space explosion. Nevertheless, the event-based framework can be used for synthesis of distributed or hierarchical supervisors. Two distributed supervisors have been synthesized for the control problem of the multimover with the event-based framework, using automaton abstraction [Su08a]. The state-based framework is more efficient with respect to centralized supervisor synthesis. However, a disadvantage of the state-based framework might be that only centralized supervisors can be synthesized with this framework. A centralized supervisor has been synthesized for the control problem of the multimover with the state-based framework.

The state-based framework is often preferred for synthesizing supervisors for industrial systems, since this framework is more convenient with respect to modelling the requirements. The state-based framework allows to model requirements by logical expressions *and* finite state machines, while for the event-based framework only finite state machines can be used. In this project, an algorithm is developed that automatically converts basic logical expressions to finite state machines. With this conversion, the user can specify the requirements with logical expressions *and* finite state machines and synthesize a supervisor with the event-based framework. An experiment has been performed with the model of the multimover and an

event-based distributed supervisor has been synthesized with requirements specified by logical expressions and finite state machines. The event-based supervisor synthesis did not suffer from calculation complexity.

The supervisors that have been synthesized using both frameworks have been implemented on a real embedded platform of the multimover. To this end, a state tracker, that tracks the state of the multimover, and a control decision maker which sends appropriate actions have been implemented. The prototype implementation is developed in such a way that supervisors, synthesized with either one of both frameworks, can be implemented easily. The drawbacks of this prototype implementation have been addressed. While the models are based on the assumptions that communication is synchronous, no problems were encountered due to asynchronous communication between the components of the multimover and the supervisor implementation. However, the supervisor implementation could still contain blocking behaviour due to a wrong choice of control actions [Mal03].

The prototype implementation has been validated on the real hardware platform and the conclusion is drawn that the implementation controls the multimover satisfactorily, based on implementation testing. The implementation showed the same behaviour as the models. Additionally, both frameworks are suitable for implementation. Finally, a suitable concept of implementation for supervisory controllers has been proven. With the prototype implementation of the supervisor in the control software, we have completed the synthesis-based engineering process. Subsequently, we state below the most relevant findings about the usage of formal models and supervisory control synthesis in the product development process.

Formal models are a key element in the synthesis-based engineering process. These formal models provide a structured and systematic approach to specify component and system behaviour with more consistency and less ambiguity than documents, because the model semantics precisely defines what a certain modelling construct means [Bra08]. By working with formal models in an early stage of the product development process, the engineers are forced to clarify all aspects of the system in an early stage of the product development process. Clarity contributes to a good design and correct control software. Furthermore, modelling the uncontrolled system by finite state machines and modelling the requirements by finite state machines and logical expressions is intuitive. However, modelling skills need to be developed to be able to model control systems and time is needed to develop those skills.

The automatic synthesis of a supervisor changes the software development process from designing and debugging controller code into designing and debugging *requirements*, assuming correct plant models. Since these requirements are modelled formally, we do not need to test the model of the supervisor against the requirements, since it is mathematically correct by construction. Thus, the engineers can focus on validating the system, not on verifying the software design. Subsequently, the requirements of a system can change over time, due to customer demands. As a consequence, in traditional engineering, all changes have to be made in the software design informally, and this is difficult to do without introducing errors or inconsistencies. With supervisory control theory, only plant models and requirement models have to be adapted and a new supervisor can be synthesized, which is correct by construction. This means that the system is *evolvable*, i.e. able to withstand changes.

In addition, the synthesized supervisors can be simulated immediately, which results in a short feedback loop in the development process. Furthermore, the usage of models allows the application of model-based techniques, such as simulation and formal verification, which can detect errors in an early stage of the system development process. As a result, the costs to develop expensive prototypes can possibly be reduced. Furthermore, since the desired behaviour is specified in models instead of in the software code, engineers can have a better understanding of the control software, which can lead to an easier validation of the resulting control software with respect to the original informal specifications.

We end this section by considering how the Key Performance Indicators (KPI) of Chapter 1 might be affected by synthesis-based engineering. While the functionality is increasing ($F \uparrow$), we conclude that the quality can possibly increase, due to the fact that the software can be mathematically correct with respect to the models of the requirements ($Q \uparrow$). Additionally, the time-to-market might decrease, since every change in functionality needs only a small change in the models ($T \downarrow$). Finally, the product costs can possibly be reduced ($C \downarrow$), since simulation allows the engineers to detect errors in an early stage of the product development process, which leads to development of less prototypes.

6.2 Suggestions for further research

In order to synthesize a supervisor, models have to be defined of the uncontrolled system and its control requirements. In this project, the uncontrolled system is modelled by finite state machines and the control requirements by logical expressions and finite state machines. Since we were able to model the supervisory control problem satisfactorily, the conclusion can be drawn that the existing modelling environment is sufficient for modelling the supervisory control problem of the multimover. However, more case studies need to be performed to test the applicability of the existing modelling environment to other industrial control problems. To give an example, one can think of specifying control requirements with temporal logic [Seo07].

A concept of a supervisory controller is developed that tracks the state of a system and determines an appropriate control action according to the state of the system. However, it could be the case that an infinite sequence of control actions is chosen or that a marker state is never reached and as a result, the system is blocking, which is not desired. The properties of [Mal03] prevent a supervisory controller from doing this. Therefore, model checkers could be used to check for these properties. Synthesizing supervisors that are optimal with respect to time could probably also solve this problem, since supervisors which contain livelock or non-reachable marker states are not time-optimal. Therefore, the synthesis of time-optimal supervisors for real industrial systems is suggested as future research.

Finally, the models that are used for supervisor synthesis assume that the communication between the uncontrolled system and the supervisor is synchronous. However, in contrast to the synchronous communication used in models, real systems often use asynchronous communication. As a result, the supervisor implementation could send a control action to the uncontrolled system that is based on a wrong state of the system. The prototype implementation that is developed in this project did not show this behaviour. However, this phenomenon could still occur. Therefore, a suggestion for further research is to investigate how the communication problem can be solved or avoided.

Bibliography

- [Bal92a] S. Balemi. Communication delays in connections of input/output discrete event processes. In *Decision and Control, 1992., Proceedings of the 31st IEEE Conference on*, pages 3374–3379 vol.4, 1992.
- [Bal92b] S. Balemi. *Control of discrete event systems: theory and application*. PhD thesis, Swiss federal institute of technology Zurich, 1992.
- [Bee06] D.A van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Syntax and consistent equation semantics of hybrid chi. 68(1-2):129, 2006.
- [Bee07] D.A. van Beek, M.A. Reniers, R.R.H. Schiffelers, and J.E. Rooda. Foundations of a compositional interchange format for hybrid systems. In *Hybrid Systems: Computation and Control*, pages 587–600. 2007.
- [Bee08] D.A. van Beek, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Concrete syntax and semantics of the compositional interchange format for hybrid systems. In *Proc. 17th IFAC World Congress, Seoul, Korea*, 2008.
- [Ber09] E.L.G. Bertens, R. Fabel, D.A. van Beek, M. Petreczky, and J.E. Rooda. Supervisory control synthesis for exception handling in printers. In *Proceedings Philips Conference on Applications of Control Technology 2009*, 2009. Paper available at <http://se.wtb.tue.nl/sewiki/projects/twins/start>, 2009. Philips Applied Technologies.
- [Bra08] N.C.W.M. Braspenning, J.E. Rooda, J.C.M. Bacten, and J.M. van de Mortel-Fronczak. *Model-based integration and testing of high-tech multi-disciplinary systems*. PhD thesis. Eindhoven University of Technology, 2008.
- [Bro03] B. Broekman and E. Notenboom. *Testing Embedded Software*. Addison-Wesley, 1st edition, 2003.
- [Cai08] K. Cai. Supervisor localization: A top-down approach to distributed control of discrete-event systems. MSc thesis, Graduate Department of Electrical and Computer Engineering, University of Toronto, 2008.
- [Cas07] C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2nd edition, November 2007.
- [Die02] P. Dietrich, R. Malik, W.M. Wonham, and B.A. Brandin. Implementation considerations in supervisory control. In *Synthesis and Control of Discrete Event Systems*, pages 185–201. Kluwer Academic Publishers, 2002.
- [Hop93] T. Hoppe and P. Meseguer. VVT terminology: A proposal. *IEEE Expert: Intelligent Systems and Their Applications*, 8(3):48–55, 1993.

- [Jac09] K.G.M. Jacobs, J. Markovski, D.A. van Beek, J.E. Rooda, and L.J.A.M. Somers. Specifying state-based supervisory control requirements. SE Report 2009-06, Eindhoven University of Technology, Systems Engineering Group, Department of Mechanical Engineering, Eindhoven, The Netherlands, 2009.
- [Led05] R.J. Leduc, M. Lawford, and W.M. Wonham. Hierarchical interface-based supervisory control-part II: parallel case. *Automatic Control, IEEE Transactions on*, 50(9):1336–1348, 2005.
- [Ma05] C. Ma and W.M. Wonham. *Nonblocking Supervisory Control of State Tree Structures*. Springer, 1st edition, August 2005.
- [Ma06] C. Ma and W.M. Wonham. Nonblocking supervisory control of state tree structures. *Automatic Control, IEEE Transactions on*, 51(5):782–793, 2006.
- [Mal02] P. Malik. Generating controllers from discrete-event models. In *Proceedings of MOVEP'2002*, 2002.
- [Mal03] P. Malik, H. Hagen, O. Mayer, and W. Buttner. *From supervisory control to nonblocking controllers for discrete event systems*. PhD thesis, University of Kaiserslautern, 2003.
- [Man85] Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming, Volume 1*. Addison-Wesley Professional, 1985.
- [Mor07] A. Morgenstern and K. Schneider. Synthesizing deterministic controllers in supervisory control. *Informatics in Control, Automation and Robotics II*, page 95, 2007.
- [Pat89] R.J. Patton, P.M. Frank, and R.N. Clarke, editors. *Fault diagnosis in dynamic systems: theory and application*. Prentice-Hall, Inc., 1989.
- [Ram87] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control and Optimization*, 25(1):206–230, 1987.
- [Roo86] P. Rook. Controlling software projects. *Software Engineering Journal*, 1(1):7–16, 1986.
- [Sch09] R.R.H. Schiffelers, R.J.M. Theunissen, D.A. van Beek, and J.E. Rooda. Model-based engineering of supervisory controller using cif. *3rd IFAC Conference on Analysis and Design of Hybrid Systems*, 2009. Submitted for review.
- [Seo07] Kiam Tian Seow. Integrating temporal logic as a State-Based specification language for Discrete-Event control design in finite automata. *Automation Science and Engineering, IEEE Transactions on*, 4(3):451–464, 2007.
- [Su08a] R. Su, J.H. van Schuppen, and J.E. Rooda. Model abstraction of nondeterministic finite state automata in supervisor synthesis. SE Report 2008-03, Eindhoven University of Technology, Systems Engineering Group, Department of Mechanical Engineering, Eindhoven, The Netherlands, 2008.
- [Su08b] R. Su, J.H. van Schuppen, J.E. Rooda, and A.T. Hofkamp. Nonconflict check by using sequential automaton abstractions. SE Report 2008-010, Eindhoven University of Technology, Systems Engineering Group, Department of Mechanical Engineering, Eindhoven, The Netherlands, 2008.
- [Su09a] R. Su, J.E. Rooda, and J.H. van Schuppen. The synthesis of time optimal supervisors by using heaps-of-pieces. SE Report 2009-08, Eindhoven University of Technology, Systems Engineering Group, Department of Mechanical Engineering, Eindhoven, The Netherlands, 2009.
- [Su09b] R. Su, J.H. van Schuppen, and J.E. Rooda. Aggregative synthesis of distributed supervisors based on automaton abstraction. SE Report 2009-01, Eindhoven University of Technology, Systems Engineering Group, Department of Mechanical Engineering, Eindhoven, The Netherlands, 2009.

- [Su09c] R. Su, J.H. van Schuppen, and J.E. Rooda. Coordinated distributed supervisory control. SE Report 2009-02, Eindhoven University of Technology, Systems Engineering Group, Department of Mechanical Engineering, Eindhoven, The Netherlands, 2009.
- [The08] R.J.M. Theunissen, R.R.H. Schiffelers, D.A. van Beek, and J.E. Rooda. Supervisory control synthesis for a patient support system. SE Report 2008-08, Eindhoven University of Technology, Systems Engineering Group, Department of Mechanical Engineering, Eindhoven, The Netherlands, 2008.
- [Won84] W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. In *Decision and Control, 1984. The 23rd IEEE Conference on*, volume 23, pages 1073–1080, 1984.
- [Won08] W.M. Wonham. Supervisory control of discrete-event systems. Technical report, University of Toronto, Toronto, 2008.

Appendix A

Formal models

In this chapter, all used formal models are presented that are used for synthesizing a supervisor for the chosen case study, the multimover. In the first section, all used plant models for synthesizing a supervisor are presented.

A.1 Plant models

In this section, all plant models are presented. The alphabets of all plant models are disjoint. For clarity, all state names and event names are prefixed with an abbreviation of the component name. For a full list of used event names and state names, see Appendix A.3 and Appendix A.4.

A.1.1 Interface buttons

An interface button is represented by a small automaton with two uncontrollable events: the event that represents the button being pressed (*press*) and the event that represents the button being released (*release*). The user interface of the multimover contains three buttons: a reset, forward and backward button. In Figure A.1, the automaton representing the forward button is depicted.

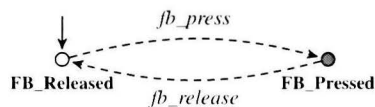


Figure A.1: Automaton of the forward button

A.1.2 Interface LEDs

The user interface of the multimover contains three LEDs. Each LED is modelled by one automaton. The automaton representing the forward LED is given in Figure A.2. Note that the initial states and marker states of the LEDs can differ.

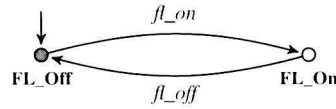


Figure A.2: Automaton of the forward LED

A.1.3 Bumper switch

This automaton represents the sensor mounted on the bumper of the multimover that can detect physical contact with an object. The automaton of a bumper switch has the same structure as an automaton of a button. See Figure A.3.

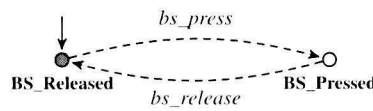


Figure A.3: Automaton of the bumper switch

A.1.4 Battery

This automaton represents the sensor that measures the battery level of the battery. If the battery level is below a certain limit, an uncontrollable event *ba_empty* is sent. If the vehicle is charged, *ba_ok* is sent. The automaton representing the battery is depicted in Figure A.4.

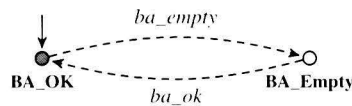


Figure A.4: Automaton of a battery

A.1.5 Proximity sensors

The multimover contains four proximity sensors, two at the front and two at the back of the multimover. On each side, we have a long proximity sensor and a short proximity sensor. If the multimover is riding forward and the short proximity sensor on the forward side is activated, the multimover should stop immediately. If only the long proximity sensor is activated, the multimover should drive at a safe speed. If the proximity sensors at the back of the multimover are activated while it is riding in the forward direction, nothing should happen. The proximity sensors are modelled like other sensors on the vehicle. In Figure A.5, the automaton of the short proximity front sensor is depicted.

A.1.6 Ride Control

In Figure A.6, the automaton of Ride Control is depicted. Ride Control can send two uncontrollable events: *rc_stop* which denotes a ‘general stop’ command and *rc_start* which denotes a ‘general start’ command. This is modelled with a selfloop in each state, since they can occur in any arbitrary order.

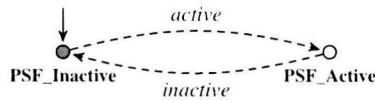


Figure A.5: Automaton of a proximity sensor

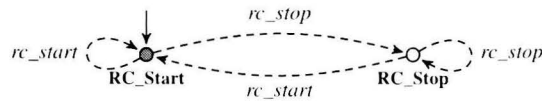


Figure A.6: Automaton of Ride Control

A.1.7 Drive motor

This automaton represents the drive motor of the multimover (see Figure A.7). The controllable events *dm_enable_fw* and *dm_enable_bw* represent the drive motor being powered on with initial speed in the forward direction or backward direction, respectively. From the state **On**, the maximum speed of the drive motor can be determined. This is modelled by the selfloop with controllable events at state **On**. If the drive motor does not behave like desired due to a hardware error, the uncontrollable event *dm_error* will be declared, which turns off the drive motor immediately, since the motor is not reliable anymore. From state **On**, it is possible to stop the drive motor with the controllable event *dm_stop*. The motor is switched off when the motor has stopped completely. This is modelled by the uncontrollable event *dm_disable* at the state **Stopping** and brings us back to state **Off**.

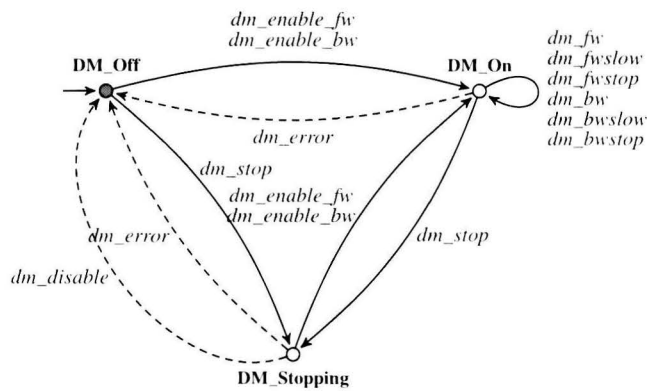


Figure A.7: Automaton of the drive motor

A.1.8 Steer motor

The model of the steer motor contains the motor and track sensor that controls the steering action of the multimover. The automaton model of the steer model is depicted in Figure A.8. The only discrete modelled behaviour is switching on (*sm_enable*) and switching off (*sm_disable*) the steer motor. If a hardware error occurs, the uncontrollable event *sm_error* takes place which switches off the steer motor. Note that this uncontrollable event can also occur at state **Off**, namely when the steer motor is still slowing down.

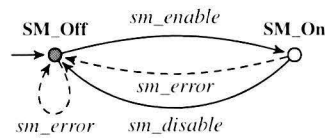


Figure A.8: Automaton of the steer motor

A.1.9 Scene program handler

The Scene Program Handler (see Figure A.9) reads the scene programs provided by the customer and sends certain commands to the rotation device, drive motor, steer motor and audio player. Since only starting (*sh_enable*) and stopping (*sh_disable*) the reading of the scene program is relevant for our supervisor, only these events are modelled. Because a scene program could contain a command that the multimover should start driving in the opposite direction, the uncontrollable event *sh_chdir* is modelled. If the scene program file contains a parse error, the multimover should stop moving and enter the emergency mode. If a parse error is read, the uncontrollable event *sh_error* will occur.

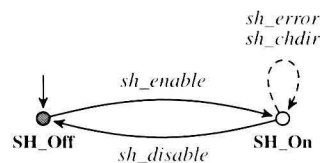


Figure A.9: Automaton of the scene program handler

A.1.10 Multimover

The automaton depicted in Figure A.10 consists of three internal controllable events that are used to specify the state of the multimover. The desired behaviour of the multimover can roughly be divided in three states and this plant model represents these three states:

- *MM_Active* In this state, the multimover is active and operational.
- *MM_Emergency* The multimover ends up in this state after an emergency happened. If this state is active, all motors are powered off and the multimover must be reset to be deployed into the ride again.
- *MM_Reset* If this state is active, the system is reset and waiting for being deployed into the ride.

A.2 Requirement models

In this section, all requirement models are presented. For each module, all logical expressions and automata are explained that are used to specify the intended behaviour of the multimover.

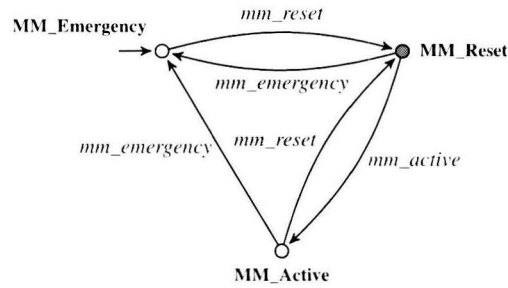


Figure A.10: Automaton of the multimover

A.2.1 LED module

Logical expressions

The ResetLED may only be switched off if the status of the multimover is active or reset.

$$\rightarrow \{ rl_off \} \Rightarrow (\mathbf{MM_Active} \downarrow \vee \mathbf{MM_Reset} \downarrow)$$

The ResetLED may only be switched on if the status of the multimover is emergency.

$$\rightarrow \{ rl_on \} \Rightarrow \mathbf{MM_Emergency} \downarrow$$

The ForwardLED may only be switched on if the status of the multimover is reset.

$$\rightarrow \{ fl_on \} \Rightarrow \mathbf{MM_Reset} \downarrow$$

The ForwardLED may only be switched off if the status of the multimover is active or emergency.

$$\rightarrow \{ fl_off \} \Rightarrow (\mathbf{MM_Active} \downarrow \vee \mathbf{MM_Emergency} \downarrow)$$

The BackwardLED may only be switched on if the status of the multimover is reset.

$$\rightarrow \{ bl_on \} \Rightarrow \mathbf{MM_Reset} \downarrow$$

The BackwardLED may only be switched off if the status of the multimover is active or emergency.

$$\rightarrow \{ bl_off \} \Rightarrow (\mathbf{MM_Active} \downarrow \vee \mathbf{MM_Emergency} \downarrow)$$

A.2.2 Motor module

Logical expressions

The Scene program handler may only be switched off only if the status of the multimover is reset or emergency.

$$\rightarrow \{ sh_disable \} \Rightarrow (\mathbf{MM_Reset} \downarrow \vee \mathbf{MM_Emergency} \downarrow)$$

The Drive Motor may only be stopped if the status of the multimover is reset or emergency and the Scene program Handler is off.

$$\rightarrow \{ dm_stop \} \Rightarrow ((\mathbf{MM_Reset} \downarrow \vee \mathbf{MM_Emergency} \downarrow) \wedge \mathbf{SH_Off} \downarrow)$$

The Steer Motor may only be switched off if the status of the multimover is reset or emergency and the Drive Motor is off.

$$\rightarrow \{ sm_disable \} \Rightarrow ((\mathbf{MM_Reset} \downarrow \vee \mathbf{MM_Emergency} \downarrow) \wedge \mathbf{DM_Off} \downarrow)$$

The Steer motor may only be switched on if the status of the multimover is Active.

$$\rightarrow \{ sm_enable \} \Rightarrow \mathbf{MM_Active} \downarrow$$

The Drive Motor may only be switched on if the status of the multimover is Active and the Steer Motor is on.

$$\rightarrow \{ dm_enable_fw, dm_enable_bw \} \Rightarrow (\mathbf{MM_Active} \downarrow \wedge \mathbf{SM_On} \downarrow)$$

The Scene Program Handler may only be switched on if the status of the multimover is Active, the Steer Motor is on and the Drive Motor is on.

$$\rightarrow \{ sh_enable_on \} \Rightarrow (\mathbf{MM_Active} \downarrow \wedge \mathbf{SM_On} \downarrow \wedge \mathbf{DM_On} \downarrow)$$

The Drive Motor may only execute another drive command if the multimover is Active.

$$\rightarrow \{ dm_enable_fw, dm_enable_bw, dm_fw, dm_fwslow, dm_fwstop, dm_bw, dm_bwslow, dm_bwstop \} \Rightarrow \mathbf{MM_Active} \downarrow$$

Automaton

The automaton depicted in Figure A.11 specifies the relationship between the scene program handler and the drive motor. If the scene program handler receives a command to change the direction sh_chdir , the active state of the drive motor is changed.

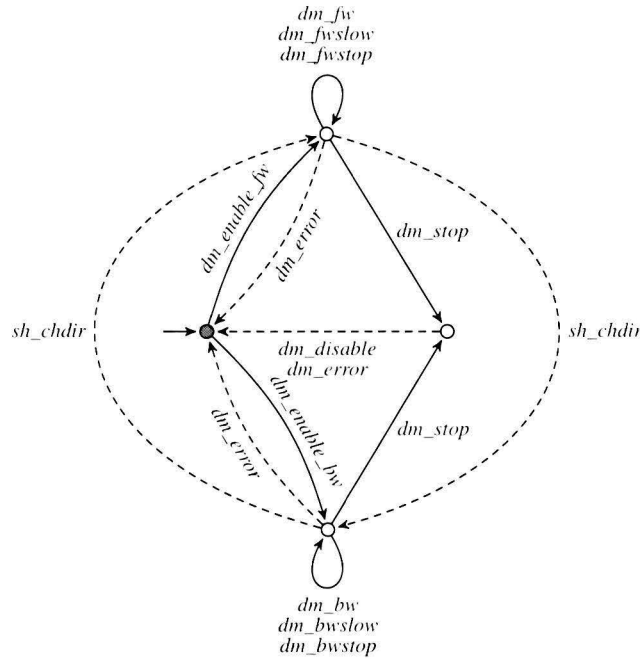


Figure A.11: Requirement of the motor module

A.2.3 Button module

Logical expressions

The multimover may only switch to Active if the forward button or the backward button (not both) is pressed and the resetbutton is not pressed.

$$\rightarrow \{ mm_active \} \Rightarrow (((\mathbf{FB_Pressed} \downarrow \wedge \mathbf{BB_Released} \downarrow) \vee (\mathbf{BB_Pressed} \downarrow \wedge \mathbf{FB_Released} \downarrow)) \wedge \neg \mathbf{RB_Pressed} \downarrow)$$

The multimover may only switch to Reset if the reset button is pressed.

$$\rightarrow \{ mm_reset \} \Rightarrow \mathbf{RB_Pressed} \downarrow$$

Automata

The automata depicted in Figures A.12a and A.12b determine the occurrence of the events dm_enable_fw and dm_enable_bw . Both events are only allowed if the corresponding interface button is pressed (fb_press or bb_press) and the multimover has become active (mm_active). Note that these requirements is the occurrence of an event under condition of a sequence of events and is therefore specified by an automaton.

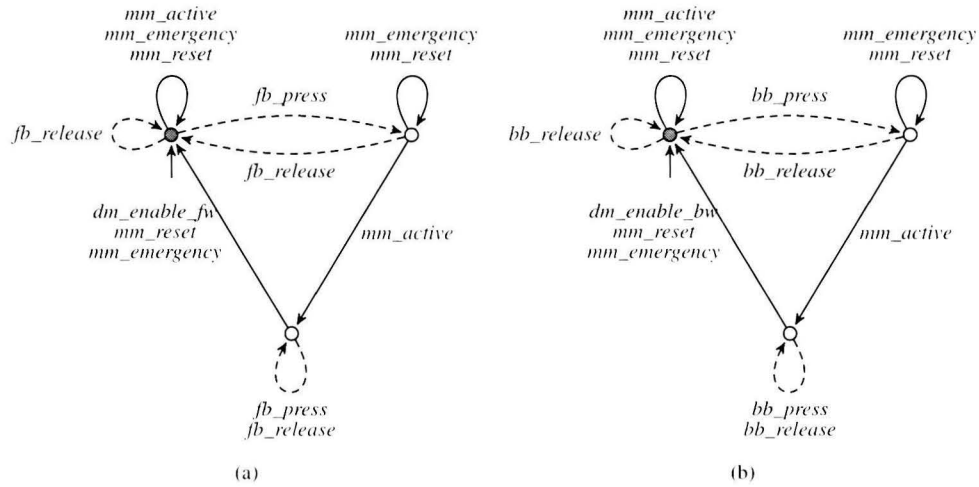


Figure A.12: Requirements of the button module

A.2.4 Emergency module

Logical expressions

The multimover may only switch to reset and active only if the bumper switch is not active and the battery is not empty.

$$\rightarrow \{ mm_reset, mm_active \} \Rightarrow (\mathbf{BS_Released} \downarrow \wedge \mathbf{BA_OK} \downarrow)$$

Automata

The automaton specified in Figure A.13 determines the occurrence of the event $mm_emergency$. This event is only allowed after an 'emergency event' occurred, e.g. the steer motor sending an error (sm_error), the drive motor sending an error (dm_error), the scene program handler sending an error (sh_error), the battery becoming empty (ba_empty) or the bumper switch being pressed (bs_press).

A.2.5 Proximity module

Logical expressions

The multimover must stop driving in the forward direction only if Ride Control is in the status Stop or the short proximity sensor in the forward direction is active.

$$\rightarrow \{ dm_fwstop \} \Rightarrow (\mathbf{RC_Stop} \downarrow \vee \mathbf{PSF_Active} \downarrow)$$

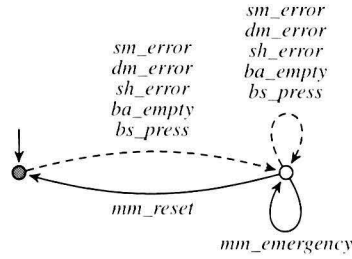


Figure A.13: Requirement of the emergency module

The multimover must stop driving in the backward direction only if Ride Control is in the status Stop or the short proximity sensor in the backward direction is active.

$\rightarrow \{ dm_bwstop \} \Rightarrow (\mathbf{RC_Stop} \downarrow \vee \mathbf{PSB_Active} \downarrow)$

The multimover must continue driving in the forward direction only if Ride Control is in the status Start and the short proximity sensor in the backward direction is inactive.

$\rightarrow \{ dm_fwslow, dm_fw \} \Rightarrow (\mathbf{RC_Start} \downarrow \wedge \mathbf{PSF_Inactive} \downarrow)$

The multimover must continue driving in the backward direction only if Ride Control is in the status Start and the short proximity sensor in the backward direction is inactive.

$\rightarrow \{ dm_bwslow, dm_bw \} \Rightarrow (\mathbf{RC_Start} \downarrow \wedge \mathbf{PSB_Inactive} \downarrow)$

The multimover must slow down in the forward direction only if the Front Long Proximity Sensor is active.

$\rightarrow \{ dm_fwslow \} \Rightarrow \mathbf{PLF_Active} \downarrow$

The multimover must drive at regular speed in the forward direction only if the Front Long Proximity Sensor is inactive.

$\rightarrow \{ dm_fw \} \Rightarrow \mathbf{PLF_Inactive} \downarrow$

The multimover must slow down in the backward direction only if the Backward Long Proximity Sensor is active.

$\rightarrow \{ dm_bwslow \} \Rightarrow \mathbf{PLB_Active} \downarrow$

The multimover must drive at regular speed in the backward direction only if the Backward long Proximity Sensor is inactive.

$\rightarrow \{ dm_bw \} \Rightarrow \mathbf{PLB_Inactive} \downarrow$

Automata

The proximity module contains one requirement specified by an automaton. Since this automaton is too large to depict here, only a description is given. This requirement specifies the occurrence of the events dm_fw , dm_fwslow , dm_fwstop , dm_bw , dm_bwslow , dm_bwstop and dm_stop . Each of these events are not allowed to take place twice without the occurrence of another event in between.

A.3 Event list

A list of all used events is given in Table A.2. In this table, the controllability of each event is stated (Uncontrollable / Controllable), together with a short description of the representation of each event.

Table A.1: List of events.

Event	U/C	Description
<i>mm_active</i>	C	The multimover switches to state MM_Active .
<i>mm_reset</i>	C	The multimover switches to state MM_Reset .
<i>mm_emergency</i>	C	The multimover switches to state MM_Emergency .
<i>bb_press</i>	U	The backward button is being pressed.
<i>bb_release</i>	U	The backward button is being released.
<i>fb_press</i>	U	The forward button is being pressed.
<i>fb_release</i>	U	The forward button is being released.
<i>rb_press</i>	U	The reset button is being pressed.
<i>rb_release</i>	U	The reset button is being released.
<i>bl_on</i>	C	The backward LED is being switched on.
<i>bl_off</i>	C	The backward LED is being switched off.
<i>fl_on</i>	C	The forward LED is being switched on.
<i>fl_off</i>	C	The forward LED is being switched off.
<i>rl_on</i>	C	The reset LED is being switched on.
<i>rl_off</i>	C	The reset LED is being switched off.
<i>dm_enable_fw</i>	C	The drive motor is being switched on in the forward direction.
<i>dm_enable_bw</i>	C	The drive motor is being switched on in the backward direction.
<i>dm_fw</i>	C	The maximum speed of the drive motor in the forward direction is set to 0.5 <i>m/s</i> .
<i>dm_fwslow</i>	C	The maximum speed of the drive motor in the forward direction is set to 0.2 <i>m/s</i> .
<i>dm_fwstop</i>	C	The maximum speed of the drive motor in the forward direction is set to 0.0 <i>m/s</i> .
<i>dm_bw</i>	C	The maximum speed of the drive motor in the backward direction is set to 0.5 <i>m/s</i> .
<i>dm_bwslow</i>	C	The maximum speed of the drive motor in the backward direction is set to 0.2 <i>m/s</i> .
<i>dm_bwstop</i>	C	The maximum speed of the drive motor in the backward direction is set to 0.0 <i>m/s</i> .
<i>dm_stop</i>	C	The drive motor is decelerating to a speed of 0.0 <i>m/s</i> .
<i>dm_disable</i>	U	The drive motor has no speed and is being switched off.
<i>dm_error</i>	U	The drive motor is broken and is being switched off.
<i>sm_enable</i>	C	The steer motor is being switched on.
<i>sm_disable</i>	C	The steer motor is being switched off.
<i>sm_error</i>	U	The steer motor is broken and is being switched off.
<i>sh_enable</i>	C	The scene program handler is being started with reading the scene program.
<i>sh_disable</i>	C	The scene program handler is being stopped with reading the scene program.
<i>sh_error</i>	U	The scene program handler has found a parse error.
<i>sh_chdir</i>	U	The scene program handler has received the command to change direction.
<i>rc_start</i>	U	The command 'start' of Ride Control is received.
<i>rc_stop</i>	U	The command 'stop' of Ride Control is received.
<i>bs_press</i>	U	The bumper switch has become active.
<i>bs_release</i>	U	The bumper switch has become inactive.
<i>psf_active</i>	U	The short proximity sensor on the forward side has become active.
<i>psf_inactive</i>	U	The short proximity sensor on the forward side has become inactive.
<i>plf_active</i>	U	The long proximity sensor on the forward side has become active.
<i>plf_inactive</i>	U	The long proximity sensor on the forward side has become inactive.

Continued on next page

Table A.1: List of events (*continued*).

Event	U/C	Description
<i>psb_active</i>	U	The short proximity sensor on the backward side has become active.
<i>psb_inactive</i>	U	The short proximity sensor on the backward side has become inactive.
<i>plb_active</i>	U	The long proximity sensor on the backward side has become active.
<i>plb_inactive</i>	U	The long proximity sensor on the backward side has become inactive.
<i>ba_empty</i>	U	The battery level has become too low.
<i>ba_ok</i>	U	The battery level has become sufficient.

A.4 State list

Table A.2: List of states.

State	Initial	Marked	Description
MM_Active			The multimover is in state 'Active'.
MM_Reset		✓	The multimover is in state 'Reset'.
MM_Emergency	✓		The multimover is in state 'Emergency'.
BB_Pressed			The backward button is pressed.
BB_Released	✓	✓	The backward button is released.
FB_Pressed			The forward button is pressed.
FB_Released	✓	✓	The forward button is released.
RB_Pressed			The reset button is pressed.
RB_Released	✓	✓	The reset button is released.
BL_On		✓	The backward LED is switched on.
BL_Off	✓		The backward LED is switched off.
FL_On		✓	The forward LED is switched on.
FL_Off	✓		The forward LED is switched off.
RL_On	✓		The reset LED is switched on.
RL_Off		✓	The reset LED is switched off.
DM_Off	✓	✓	The drive motor is switched on in the forward direction.
DM_On			The drive motor is switched on in the backward direction.
DM_Stopping			The drive motor is decelerating to a speed of 0.0 <i>m/s</i> .
SM_On			The steer motor is switched on.
SM_Off	✓	✓	The steer motor is switched off.
SH_On			The scene program handler is switched on and reading the scene program.
SH_Off	✓	✓	The scene program handler is switched off.
RC_Start	✓	✓	The last received command of Ride Control is 'start'.
RC_Stop			The last received command of Ride Control is 'stop'.
BS_Pressed			The bumper switch is pressed.
BS_Released	✓	✓	The bumper switch is released.
PSF_Active			The short proximity sensor on the forward side is active.
PSF_Inactive	✓	✓	The short proximity sensor on the forward side is inactive.
PLF_Active			The long proximity sensor on the forward side is active.
PLF_Inactive	✓	✓	The long proximity sensor on the forward side is inactive.
PSB_Active			The short proximity sensor on the backward side is active.
PSB_Inactive	✓	✓	The short proximity sensor on the backward side is inactive.
PLB_Active			The long proximity sensor on the backward side is active.

Continued on next page

Table A.2: List of states (*continued*).

State	Initial	Marked	Description
PLB_Inactive	✓	✓	The long proximity sensor on the backward side is inactive.
BA_Empty			The battery level is too low.
BA_OK	✓	✓	The battery level is sufficient.

Appendix B

Logic expression converter

In this chapter, the source code is listed that converts the type 1 and type 2 expressions of the synthesis tool of Ma-Wonham to automata, which is used in Chapter 4. The source code uses functions of the Supervisor Synthesis Package¹.

B.1 Source code

The source code consists of four files, which contain a class structure (data.py), an expression parser (parser.py), the frontend functions (frontend.py) and other functions (func.py).

B.1.1 data.py

Specification B.1: Python script of used data structure

```
"""
    The source code of the logic expression converter.
"""

5 class Type1Requirement(object):
    """
        This class represents a type 1 requirement.
        @ param state_set: Set of states for which the state is
    10 state_text: String of states
    """
    def __init__(self, state_set):
        self.state_set = state_set

    15 def __str__(self):
        state_set_text = "(" + ", ".join(self.state_set) + ")"
        return "Type1: " + state_set_text

class Type2Requirement(object):
    """
    20 This class represents a type 2 requirement.
    @ param state_set: Set of states.
    @ param event: Event that is not a word.
    @ param state_text: String of states
    25 @ param event_text: String of event
    """
    def __init__(self, state_set, event):
    30 self.state_set = state_set
        self.event = event
```

¹Downloadable at <http://dev.se.wtb.tue.nl/projects/chi-tooling/downloads>


```

def __str__(self):
    state_set_text = "{" + ", ".join(self.state_set) + "}"
35     return "Type2: " + state_set_text + ", " + self.event

```

B.1.2 parser.py

Specification B.2: Python script of parser

```

from aut_expr import data

import ply.lex as lex
5 import ply.yacc as yacc

# --S--

# List of tokens: name, "data" is a data requirement.
10 tokens = (
    'LPAREN',
    'RPAREN',
    'LCURLY',
    'RCURLY',
15 'COMMA',
    'ID',
)

# --E--

20 t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LCURLY = r'\{'
t_RCURLY = r'\}'
t_COMMA = r','
25 t_ID = r'[a-zA-Z_][a-zA-Z_0-9]*'

# --P--

def t_newline(t):
    r'\n+'
30     t.lexer.lineno += len(t.value)

# --I--

t_ignore = '\t'

35

# --R--

def t_error(t):
    raise ValueError("Illegal character '%s'" % t.value[0])

40 def p_spec1(p):
    'spec : StateSet'
    p[0] = data.Type1Requirement(p[1])

def p_spec2(p):
45     'spec : LPAREN StateSet COMMA ID RPAREN'
    p[0] = data.Type2Requirement(p[2],p[4])

def p_StateSet_1(p):
    'StateSet : LCURLY States RCURLY'
50     p[0] = set(p[2])

def p_States_1(p):
    'States : ID'
    p[0] = [p[1]]

55 def p_States_2(p):
    'States : States COMMA ID'
    p[0] = p[1] + [p[3]]

60

# --E--

def p_error(p):
    raise ValueError("Syntax error in input!")

65

def parse(data):
    lexer = lex.lex()

```

```

lexer.input(data)

70  yacc.yacc(debug=1)
    p=yacc.parse(data)

    return p

```

B.1.3 frontend.py

Specification B.3: Python script of used frontend functions

```

from aut_expr import parser, data, func

def make_type1_specification(aut_fnames, logical_spec, req_fname):
    """
    5  Compute a specification automaton according to a type 1 logic specification.

    Input: aut_fnames: Name separated list of automata files.
         logical_spec: Logical specification in LTL format.
         req_fname: File name for writing the resulting automata.

    10  Output: type1spec: Type 1 specification in LTL format.
         state_map: Dict.

    Input: req_fname: File name for writing the resulting automata.
         state_map: Dict.

    15  """
    coll = collection.Collection()

    aut_list = load_automata(coll, aut_fnames, False, False)

    20  type1spec = parser.parse(logical_spec)
    if not isinstance(type1spec, data.Type1Requirement):
        raise ValueError("Error with parsing")

    25  aut_list, state_tuple = func.sort_automata(aut_list, type1spec.state_set)

    result, state_map = func.make_product_with_mapping(aut_list)

    illegal_state = state_map[state_tuple]
    30  result.remove_state(illegal_state)

    save_automaton(result, "Requirement saved in %s", req_fname)

def make_type2_specification(aut_fnames, aut_fname, logical_spec, req_fname):
    """
    35  Compute a specification automaton according to a type 2 logic specification.

    Input: aut_fnames: Name separated list of automata files.
         aut_fname: File name for writing the resulting automata.
         logical_spec: Logical specification in LTL format.
         req_fname: File name for writing the resulting automata.

    40  Output: type2spec: Type 2 specification in LTL format.
         state_map: Dict.

    Input: aut_fnames: Name separated list of automata files.
         aut_fname: File name for writing the resulting automata.
         logical_spec: Logical specification in LTL format.
         req_fname: File name for writing the resulting automata.

    45  """
    coll = collection.Collection()

    aut_list = load_automata(coll, aut_fnames, False, False)

    50  type2spec = parser.parse(logical_spec)
    if not isinstance(type2spec, data.Type2Requirement):
        raise ValueError("Error with parsing")

    55  aut_list, state_tuple = func.sort_automata(aut_list, type2spec.state_set)

    result, state_map = func.make_product_with_mapping(aut_list)
    state = state_map[state_tuple]

    60  aut = load_automaton(coll, aut_fname, False, False)
    event = func.search_event(aut, type2spec.event)

    65

```

```

illegal_edges=[]

# If event belongs to alphabet,
# event should be disabled in predicate state.
70 if event in result.alphabet:
    edges = list(state.get_outgoing(event))
    if len(edges) > 0:
        for edge in edges:
75             result.remove_edge(edge.pred, edge.succ, edge.label)
    else:
        raise ValueError("Event %r not outgoing edge of stateset" % event)

# If event does not belong to alphabet, the event set is generated
# if it is, of all states.
80 if event in result.alphabet:
    else:
        result.add_event_set(set([event]))
        is_standardized = result.is_standardized()
        for s in result.get_states():
85             if s is not state:
                 if not (is_standardized and s is result.initial):
                     result.add_edge_data(s, s, event)

save_automaton(result, "Requirement saved in %s", req_fname)

```

B.1.4 func.py

Specification B.4: Python script of used functions

```

"""
7  This module contains the functions used in the logic expression converter.
"""

5  from automata import product, common

def sort_automata(aut_list, state_name_list):
    """
    10  Sorts the automata in aut_list according to the state_name_list.
    The state_name_list is a list of state names. The automata
    are sorted according to the state names.

    """
    15  # Sort the automata according to the state names.
    # Sort the automata according to the state names.

    # Sort the automata according to the state names.
    20  # Sort the automata according to the state names.
    # Sort the automata according to the state names.

    checked_names = set([])
    25  sorted_aut_list = []
    state_tuple = []

    for state_name in state_name_list:
        for aut in aut_list:
            for aut_state, aut_state_name in aut.state_names.iteritems():
30                 if aut_state_name == state_name:
                     if state_name in checked_names:
                         raise ValueError("Automata contain same state "\
                                     "name %r" % aut_state_name)
                     checked_names.add(aut_state_name)
                     sorted_aut_list.append(aut)
                     state_tuple.append(aut.get_state(aut_state))
35

    state_tuple = tuple(state_tuple)

    40  if not len(aut_list) == len(sorted_aut_list):
        raise ValueError("Not all automata needed!")

    if aut not in sorted_aut_list:
45         raise ValueError("Multiple states belong to one automaton")

    return sorted_aut_list, state_tuple

```

```

50 def make_product_with_mapping(aut_list):
    """
    Makes product and a corresponding mapping of all original automaton states
    to a state of the resulting product.

    @param aut_list: list of Automata
    @type aut_list: List of Automata

    @return: tuple of a Automaton and the resulting state map.
    @rtype: (Automaton, Dict of (States, ...)) to a state in the product)
    """
60     complete_state_map = {}

    if len(aut_list) == 1:
        result = aut_list[0]
        for i in result.state_names:
65             complete_state_map[(result.get_state(i),)] = result.get_state(i)
        return result, complete_state_map

    common.print_line("Must do %d product computations." % (len(aut_list) - 1))
    result = aut_list[0]
70     for idx, aut in enumerate(aut_list[1:]):
        oldresult = result
        result, state_map = product.product_map(result, aut)
        msg = "Product #%d done: %d states, %d transitions" \
              % (idx + 1, result.get_num_states(), result.get_num_edges())
75         common.print_line(msg)
        if idx == 0:
            complete_state_map = state_map
        else:
80             old_map = complete_state_map
            reverse_old_map = product.reverse_statemap(complete_state_map)
            reverse_new_map = product.reverse_statemap(state_map)
            complete_state_map={}
            for k, (c,b) in reverse_new_map.iteritems():
                complete_state_map[reverse_old_map[c]+(b,)] = k
85
    return result, complete_state_map

def search_event(aut, eventname):
    """
90     Get all the events of the aut alphabet, that is,
    """
    """
    @param aut: Automaton whose event names are looked for.
    @type aut: Automaton

95     @param eventname: Name of the event to look for.
    @type eventname: String

    @return: List of automata by event
    @rtype: List of Automata
    """
100
    for event in aut.alphabet:
        if event.name == eventname:
            return event
105
    raise ValueError ("Event %r not found in automaton" % eventname)

```

Appendix C

Supervisor synthesis

In this chapter, the script file of Specification C.1 is listed that is used for converting the state-based expressions of the multimover to automata. Furthermore, all supervisors of the event-based framework are synthesized in this script. The script that is used for synthesizing a supervisor with the state-based framework is listed in Specifications C.2 and C.3.

C.1 Event-based supervisor synthesis

Specification C.1: Batch file of event-based supervisor synthesis

```
from automata import frontend

#####
### The Batch file of the BPF
5 #####

### The list of the
frontend.make_get_size('MM.cfg') # The MM
frontend.make_get_size('FB.cfg') # The FB
10 frontend.make_get_size('BB.cfg') # The BB
frontend.make_get_size('RB.cfg') # The RB
frontend.make_get_size('PSF.cfg') # The PSF
frontend.make_get_size('PLP.cfg') # The PLP
frontend.make_get_size('PSB.cfg') # The PSB
15 frontend.make_get_size('PLB.cfg') # The PLB
frontend.make_get_size('BS.cfg') # The BS
frontend.make_get_size('BA.cfg') # The BA
frontend.make_get_size('RC.cfg') # The RC
frontend.make_get_size('FL.cfg') # The FL
20 frontend.make_get_size('BL.cfg') # The BL
frontend.make_get_size('RL.cfg') # The RL
frontend.make_get_size('DM.cfg') # The DM
frontend.make_get_size('SM.cfg') # The SM
frontend.make_get_size('SH.cfg') # The SH

25

### The list of the
frontend.make_get_size('REQ_MM.cfg')
frontend.make_get_size('REQ_BM1.cfg')
frontend.make_get_size('REQ_BM2.cfg')
30 frontend.make_get_size('REQ_EM.cfg')
frontend.make_get_size('REQ_PM.cfg')

#####
### The list of the
35 #####

### The list of the
frontend.make_type2_specification('MM.cfg', 'RL.cfg', '{MM_Emergency}, rl_off)', 'SpecLM1.cfg')
frontend.make_type2_specification('MM.cfg', 'RL.cfg', '{MM_Reset}, rl_on)', 'SpecLM2.cfg')
40 frontend.make_type2_specification('MM.cfg', 'RL.cfg', '{MM_Active}, rl_on)', 'SpecLM3.cfg')
frontend.make_type2_specification('MM.cfg', 'FL.cfg', '{MM_Emergency}, fl_on)', 'SpecLM4.cfg')
```

```

frontend.make_type2_specification('MM.cfg','FL.cfg','(MM_Active),fl_on'),'SpecLM5.cfg')
frontend.make_type2_specification('MM.cfg','FL.cfg','(MM_Reset),fl_off'),'SpecLM6.cfg')
frontend.make_type2_specification('MM.cfg','BL.cfg','(MM_Emergency),bl_on'),'SpecLM7.cfg')
45 frontend.make_type2_specification('MM.cfg','BL.cfg','(MM_Active),bl_on'),'SpecLM8.cfg')
frontend.make_type2_specification('MM.cfg','BL.cfg','(MM_Reset),bl_off'),'SpecLM9.cfg')

frontend.make_product('SpecLM1.cfg, SpecLM2.cfg, SpecLM3.cfg, SpecLM4.cfg, SpecLM5.cfg, ←
SpecLM6.cfg, SpecLM7.cfg, SpecLM8.cfg, SpecLM9.cfg', 'SpecLM.cfg')
frontend.make_dot('SpecLM.cfg','SpecLM.dot')
50

### SpecM - specifications (33) =
frontend.make_type2_specification('MM.cfg','SH.cfg','(MM_Active),sh_disable'),'SpecMM1.cfg' ←
)
frontend.make_type2_specification('MM.cfg','DM.cfg','(MM_Active),dm_stop'),'SpecMM2.cfg')
frontend.make_type2_specification('SH.cfg','DM.cfg','(SH_On),dm_stop'),'SpecMM3.cfg')
55 frontend.make_type2_specification('MM.cfg','SM.cfg','(MM_Active),sm_disable'),'SpecMM4.cfg' ←
)
frontend.make_type2_specification('DM.cfg','SM.cfg','(DM_On),sm_disable'),'SpecMM5.cfg')
frontend.make_type2_specification('DM.cfg','SM.cfg','(DM_Stopping),sm_disable'),'SpecMM6. ←
cfg')
frontend.make_type2_specification('MM.cfg','SM.cfg','(MM_Emergency),sm_enable'),'SpecMM7. ←
cfg')
frontend.make_type2_specification('MM.cfg','SM.cfg','(MM_Reset),sm_enable'),'SpecMM8.cfg')
60 frontend.make_type2_specification('MM.cfg','DM.cfg','(MM_Emergency),dm_enable_fw'),'SpecMM9 ←
.cfg')
frontend.make_type2_specification('MM.cfg','DM.cfg','(MM_Emergency),dm_enable_bw'),' ←
SpecMM10.cfg')
frontend.make_type2_specification('MM.cfg','DM.cfg','(MM_Reset),dm_enable_fw'),'SpecMM11. ←
cfg')
frontend.make_type2_specification('MM.cfg','DM.cfg','(MM_Reset),dm_enable_bw'),'SpecMM12. ←
cfg')
frontend.make_type2_specification('SM.cfg','DM.cfg','(SM_Off),dm_enable_fw'),'SpecMM13.cfg' ←
)
65 frontend.make_type2_specification('SM.cfg','DM.cfg','(SM_Off),dm_enable_bw'),'SpecMM14.cfg' ←
)
frontend.make_type2_specification('MM.cfg','SH.cfg','(MM_Emergency),sh_enable'),'SpecMM15. ←
cfg')
frontend.make_type2_specification('MM.cfg','SH.cfg','(MM_Reset),sh_enable'),'SpecMM16.cfg')
frontend.make_type2_specification('SM.cfg','SH.cfg','(SM_Off),sh_enable'),'SpecMM17.cfg')
frontend.make_type2_specification('DM.cfg','SH.cfg','(DM_Off),sh_enable'),'SpecMM18.cfg')
70 frontend.make_type2_specification('DM.cfg','SH.cfg','(DM_Stopping),sh_enable'),'SpecMM19. ←
cfg')
frontend.make_type2_specification('MM.cfg','DM.cfg','(MM_Emergency),dm_fw'),'SpecMM20.cfg')
frontend.make_type2_specification('MM.cfg','DM.cfg','(MM_Emergency),dm_fwslow'),'SpecMM21. ←
cfg')
frontend.make_type2_specification('MM.cfg','DM.cfg','(MM_Emergency),dm_fwstop'),'SpecMM22. ←
cfg')
frontend.make_type2_specification('MM.cfg','DM.cfg','(MM_Emergency),dm_bw'),'SpecMM23.cfg')
75 frontend.make_type2_specification('MM.cfg','DM.cfg','(MM_Emergency),dm_bwslow'),'SpecMM24. ←
cfg')
frontend.make_type2_specification('MM.cfg','DM.cfg','(MM_Emergency),dm_bwstop'),'SpecMM25. ←
cfg')
frontend.make_type2_specification('MM.cfg','DM.cfg','(MM_Reset),dm_fw'),'SpecMM26.cfg')
frontend.make_type2_specification('MM.cfg','DM.cfg','(MM_Reset),dm_fwslow'),'SpecMM27.cfg')
frontend.make_type2_specification('MM.cfg','DM.cfg','(MM_Reset),dm_fwstop'),'SpecMM28.cfg')
80 frontend.make_type2_specification('MM.cfg','DM.cfg','(MM_Reset),dm_bw'),'SpecMM29.cfg')
frontend.make_type2_specification('MM.cfg','DM.cfg','(MM_Reset),dm_bwslow'),'SpecMM30.cfg')
frontend.make_type2_specification('MM.cfg','DM.cfg','(MM_Reset),dm_bwstop'),'SpecMM31.cfg')
frontend.make_type2_specification('SM.cfg','MM.cfg','(SM_On),mm_active'),'SpecMM32.cfg')
frontend.make_type2_specification('DM.cfg','MM.cfg','(DM_On),mm_active'),'SpecMM33.cfg')
85 frontend.make_type2_specification('DM.cfg','MM.cfg','(DM_Stopping),mm_active'),'SpecMM34. ←
cfg')
frontend.make_type2_specification('SH.cfg','MM.cfg','(SH_On),mm_active'),'SpecMM35.cfg')

frontend.make_product('SpecMM1.cfg, SpecMM2.cfg, SpecMM3.cfg, SpecMM4.cfg, SpecMM5.cfg, ←
SpecMM6.cfg, SpecMM7.cfg, SpecMM8.cfg, SpecMM9.cfg, SpecMM10.cfg, SpecMM11.cfg, ←
SpecMM12.cfg, SpecMM13.cfg, SpecMM14.cfg, SpecMM15.cfg, SpecMM16.cfg, SpecMM17.cfg, ←
SpecMM18.cfg, SpecMM19.cfg, SpecMM20.cfg, SpecMM21.cfg, SpecMM22.cfg, SpecMM23.cfg, ←
SpecMM24.cfg, SpecMM25.cfg, SpecMM26.cfg, SpecMM27.cfg, SpecMM28.cfg, SpecMM29.cfg, ←
SpecMM30.cfg, SpecMM31.cfg, SpecMM32.cfg, SpecMM33.cfg, SpecMM34.cfg, SpecMM35.cfg', ' ←
SpecMM.cfg')
90

### SpecB - specifications (3) =
frontend.make_type2_specification('FB.cfg, BB.cfg','MM.cfg','(FB_Released,BB_Released),' ←
mm_active'),'SpecBM1.cfg')
frontend.make_type2_specification('BB.cfg, FB.cfg','MM.cfg','(BB_Pressed,FB_Pressed),' ←
mm_active'),'SpecBM2.cfg')
frontend.make_type2_specification('RB.cfg','MM.cfg','(RB_Pressed),mm_active'),'SpecBM3.cfg' ←
)

```

```

frontend.make_type2_specification('RB.cfg', 'MM.cfg', '({RB_Released}, mm_reset)', 'SpecBM4.cfg' ←
)
95 frontend.make_product('SpecBM1.cfg, SpecBM2.cfg, SpecBM3.cfg, SpecBM4.cfg', 'SpecBM.cfg')

*** Emergency start plant le
frontend.make_type2_specification('BS.cfg', 'MM.cfg', '({BS_Pressed}, mm_reset)', 'SpecEM1.cfg')
100 frontend.make_type2_specification('BS.cfg', 'MM.cfg', '({BS_Pressed}, mm_active)', 'SpecEM2.cfg' ←
)
frontend.make_type2_specification('BA.cfg', 'MM.cfg', '({BA_Empty}, mm_reset)', 'SpecEM3.cfg')
frontend.make_type2_specification('BA.cfg', 'MM.cfg', '({BA_Empty}, mm_active)', 'SpecEM4.cfg')

frontend.make_product('SpecEM1.cfg, SpecEM2.cfg, SpecEM3.cfg, SpecEM4.cfg', 'SpecEM.cfg')
105

*** Emergency start dm plant le
frontend.make_type2_specification('RC.cfg, PSF.cfg', 'DM.cfg', '({RC_Start, PSF_Inactive}, ←
dm_fwstop)', 'SpecPM1.cfg')
frontend.make_type2_specification('PSB.cfg, RC.cfg', 'DM.cfg', '({PSB_Inactive, RC_Start}, ←
dm_bwstop)', 'SpecPM2.cfg')
frontend.make_type2_specification('RC.cfg', 'DM.cfg', '({RC_Stop}, dm_fwslow)', 'SpecPM3.cfg')
110 frontend.make_type2_specification('RC.cfg', 'DM.cfg', '({RC_Stop}, dm_fw)', 'SpecPM4.cfg')
frontend.make_type2_specification('PSF.cfg', 'DM.cfg', '({PSF_Active}, dm_fwslow)', 'SpecPM5.cfg ←
')
frontend.make_type2_specification('PSF.cfg', 'DM.cfg', '({PSF_Active}, dm_fw)', 'SpecPM6.cfg')
frontend.make_type2_specification('RC.cfg', 'DM.cfg', '({RC_Stop}, dm_bwslow)', 'SpecPM7.cfg')
frontend.make_type2_specification('RC.cfg', 'DM.cfg', '({RC_Stop}, dm_bw)', 'SpecPM8.cfg')
115 frontend.make_type2_specification('PSB.cfg', 'DM.cfg', '({PSB_Active}, dm_bwslow)', 'SpecPM9.cfg ←
')
frontend.make_type2_specification('PSB.cfg', 'DM.cfg', '({PSB_Active}, dm_bw)', 'SpecPM10.cfg')
frontend.make_type2_specification('PLF.cfg', 'DM.cfg', '({PLF_Inactive}, dm_fwslow)', 'SpecPM11. ←
cfg')
frontend.make_type2_specification('PLF.cfg', 'DM.cfg', '({PLF_Active}, dm_fw)', 'SpecPM12.cfg')
frontend.make_type2_specification('PLB.cfg', 'DM.cfg', '({PLB_Inactive}, dm_bwslow)', 'SpecPM13. ←
cfg')
120 frontend.make_type2_specification('PLB.cfg', 'DM.cfg', '({PLB_Active}, dm_bw)', 'SpecPM14.cfg')

frontend.make_product('SpecPM1.cfg, SpecPM2.cfg, SpecPM3.cfg, SpecPM4.cfg, SpecPM5.cfg, ←
SpecPM6.cfg, SpecPM7.cfg, SpecPM8.cfg, SpecPM9.cfg, SpecPM10.cfg, SpecPM11.cfg, ←
SpecPM12.cfg, SpecPM13.cfg, SpecPM14.cfg', 'SpecPM.cfg')

*****
125 *** Emergency start plant le
*** Emergency start dm plant le
*****

*** Emergency start plant le
130 frontend.make_product('FL.cfg, BL.cfg, RL.cfg, MM.cfg', 'PlantLM.cfg')
frontend.make_supervisor('PlantLM.cfg', 'SpecLM.cfg', 'SupervisorC_LM.cfg')

*** Emergency start dm plant le
frontend.make_product('MM.cfg, DM.cfg, SM.cfg, SH.cfg', 'PlantMM.cfg')
135 frontend.make_product('SpecMM.cfg, REQ_MM.cfg', 'SpecMM.cfg')
frontend.make_supervisor('PlantMM.cfg', 'SpecMM.cfg', 'SupervisorC_MM.cfg')

*** Emergency start plant le
frontend.make_product('FB.cfg, BB.cfg, RB.cfg, MM.cfg, DM.cfg', 'PlantBM.cfg')
140 frontend.make_product('SpecBM.cfg, REQ_BM1.cfg, REQ_BM2.cfg', 'SpecBM.cfg')
frontend.make_supervisor('PlantBM.cfg', 'SpecBM.cfg', 'SupervisorC_BM.cfg')

*** Emergency start plant le
frontend.make_product('BS.cfg, BA.cfg, DM.cfg, SM.cfg, SH.cfg, MM.cfg', 'PlantEM.cfg')
145 frontend.make_product('SpecEM.cfg, REQ_EM.cfg', 'SpecEM.cfg')
frontend.make_supervisor('PlantEM.cfg', 'SpecEM.cfg', 'SupervisorC_EM.cfg')

*** Emergency start plant le
frontend.make_product('DM.cfg, PSF.cfg, PLF.cfg, PSB.cfg, PLB.cfg, RC.cfg', 'PlantPM.cfg')
150 frontend.make_product('SpecPM.cfg, REQ_PM.cfg', 'SpecPM.cfg')
frontend.make_supervisor('PlantPM.cfg', 'SpecPM.cfg', 'SupervisorC_PM.cfg')

frontend.make_nonconflicting_check('SupervisorLM.cfg, SupervisorMM.cfg, SupervisorBM.cfg, ←
SupervisorEM.cfg, SupervisorPM.cfg')

155 *****
*** Emergency start plant le
*** Emergency start dm plant le
*****

160 *** Emergency start plant le
frontend.make_product('FL.cfg, BL.cfg, RL.cfg, MM.cfg', 'PlantLM.cfg')
frontend.make_supervisor('PlantLM.cfg', 'SpecLM.cfg', 'SupervisorAl_LM.cfg')

```



```

frontend.make_sequential_abstraction('PlantLM.cfg, SupervisorAl_LM.cfg', 'tau, mm_active, ←
mm_emergency, mm_reset', 'SupervisorAl_LM_abstraction.cfg')

165 *** Defining abstract models
frontend.make_product('SupervisorAl_LM_abstraction.cfg, DM.cfg, SM.cfg, SH.cfg', 'PlantMM.cfg ←
')
frontend.make_product('SpecMM.cfg, REQ_MM.cfg', 'SpecMM.cfg')
frontend.make_supervisor('PlantMM.cfg', 'SpecMM.cfg', 'SupervisorAl_MM.cfg')
frontend.make_sequential_abstraction('PlantMM.cfg, SupervisorAl_MM.cfg', 'tau, mm_active, ←
mm_emergency, mm_reset, dm_stop, dm_enable_fw, dm_enable_bw, dm_fw, dm_fwslow, ←
dm_fwstop, dm_bw, dm_bwslow, dm_bwstop, dm_error, sm_error, sh_error', ' ←
SupervisorAl_MM_abstraction.cfg')

170 *** Defining abstract models
frontend.make_product('SupervisorAl_MM_abstraction.cfg, RB.cfg, BB.cfg, FB.cfg', 'PlantBM.cfg ←
')
frontend.make_product('SpecBM.cfg, REQ_BM1.cfg, REQ_BM2.cfg', 'SpecBM.cfg')
frontend.make_supervisor('PlantBM.cfg', 'SpecBM.cfg', 'SupervisorAl_BM.cfg')
175 frontend.make_sequential_abstraction('PlantBM.cfg, SupervisorAl_BM.cfg', 'tau, mm_active, ←
mm_emergency, mm_reset, dm_stop, dm_fw, dm_fwslow, dm_fwstop, dm_bw, dm_bwslow, ←
dm_bwstop, dm_error, sm_error, sh_error', 'SupervisorAl_BM_abstraction.cfg')

*** Defining abstract models
frontend.make_product('SupervisorAl_BM_abstraction.cfg, BS.cfg, BA.cfg', 'PlantEM.cfg')
frontend.make_product('SpecEM.cfg, REQ_EM.cfg', 'SpecEM.cfg')
180 frontend.make_supervisor('PlantEM.cfg', 'SpecEM.cfg', 'SupervisorAl_EM.cfg')
frontend.make_sequential_abstraction('PlantEM.cfg, SupervisorAl_EM.cfg', 'tau, dm_stop, dm_bw ←
, dm_bwslow, dm_bwstop, dm_fw, dm_fwslow, dm_fwstop', 'SupervisorAl_EM_abstraction.cfg')

*** Defining abstract models
frontend.make_product('SupervisorAl_EM_abstraction.cfg, RC.cfg, PSF.cfg, PSB.cfg, PLF.cfg, ←
PLB.cfg', 'PlantPM.cfg')
185 frontend.make_product('SpecPM.cfg, REQ_PM.cfg', 'SpecPM.cfg')
frontend.make_supervisor('PlantPM.cfg', 'SpecPM.cfg', 'SupervisorAl_PM.cfg')

*****
190 *****

*** Defining abstract models
frontend.make_product('DM.cfg, RC.cfg, PSF.cfg, PSB.cfg, PLF.cfg, PLB.cfg', 'PlantPM.cfg')
195 frontend.make_product('SpecPM.cfg, REQ_PM.cfg', 'SpecPM.cfg')
frontend.make_supervisor('PlantPM.cfg', 'SpecPM.cfg', 'SupervisorA2_PM.cfg')
frontend.make_sequential_abstraction('PlantPM.cfg, SupervisorA2_PM.cfg', 'tau, dm_stop, ←
dm_disable, dm_enable_bw, dm_enable_fw, dm_error, dm_bw, dm_bwstop, dm_bwslow, dm_fw, ←
dm_fwslow, dm_fwstop', 'SupervisorA2_PM_abstraction.cfg')

*** Defining abstract models
200 frontend.make_product('SupervisorA2_PM_abstraction.cfg, MM.cfg, SM.cfg, SH.cfg', 'PlantMM.cfg ←
')
frontend.make_product('SpecMM.cfg, REQ_MM.cfg', 'SpecMM.cfg')
frontend.make_supervisor('PlantMM.cfg', 'SpecMM.cfg', 'SupervisorA2_MM.cfg')
frontend.make_sequential_abstraction('PlantMM.cfg, SupervisorA2_MM.cfg', 'tau, mm_active, ←
mm_emergency, mm_reset, dm_enable_fw, dm_enable_bw, dm_error, sm_error, sh_error', ' ←
SupervisorA2_MM_abstraction.cfg')

205 *** Defining abstract models
frontend.make_product('SupervisorA2_MM_abstraction.cfg, BS.cfg, BA.cfg', 'PlantEM.cfg')
frontend.make_product('SpecEM.cfg, REQ_EM.cfg', 'SpecEM.cfg')
frontend.make_supervisor('PlantEM.cfg', 'SpecEM.cfg', 'SupervisorA2_EM.cfg')
frontend.make_sequential_abstraction('PlantEM.cfg, SupervisorA2_EM.cfg', 'tau, mm_active, ←
mm_emergency, mm_reset, dm_enable_fw, dm_enable_bw', 'SupervisorA2_EM_abstraction.cfg')

210 *** Defining abstract models
frontend.make_product('SupervisorA2_EM_abstraction.cfg, RB.cfg, BB.cfg, FB.cfg', 'PlantBM.cfg ←
')
frontend.make_product('SpecBM.cfg, REQ_BM1.cfg, REQ_BM2.cfg', 'SpecBM.cfg')
frontend.make_supervisor('PlantBM.cfg', 'SpecBM.cfg', 'SupervisorA2_BM.cfg')
215 frontend.make_sequential_abstraction('PlantBM.cfg, SupervisorA2_BM.cfg', 'tau, mm_active, ←
mm_emergency, mm_reset', 'SupervisorA2_BM_abstraction.cfg')

*** Defining abstract models
frontend.make_product('SupervisorA2_BM_abstraction.cfg, FL.cfg, BL.cfg, RL.cfg', 'PlantLM.cfg ←
')
frontend.make_supervisor('PlantLM.cfg', 'SpecLM.cfg', 'SupervisorA2_LM.cfg')

```

C.2 State-based supervisor synthesis

Specification C.2: Batch file of state-based supervisor synthesis

```

root = plant
{
  plant = AND {MM, FB, BB, RB, BS, BA, PSF, PLF, PSB, PLB, RC, FL, BL, RL, DM, SM, SH, MotorM3 ↔
    , ButtonM3, ButtonM4, EmergencyM3, ProximityM}

  5 MM = OR { MM_Emergency, MM_Reset, MM_Active }
  FB = OR { FB_Released, FB_Pressed }
  BB = OR { BB_Released, BB_Pressed }
  RB = OR { RB_Released, RB_Pressed }
  PSF = OR { PSF_Inactive, PSF_Active }
  10 PLF = OR { PLF_Inactive, PLF_Active }
  PSB = OR { PSB_Inactive, PSB_Active }
  PLB = OR { PLB_Inactive, PLB_Active }
  BS = OR { BS_Released, BS_Pressed }
  BA = OR { BA_OK, BA_Empty }
  15 RC = OR { RC_Start, RC_Stop }
  FL = OR { FL_Off, FL_On }
  BL = OR { BL_Off, BL_On }
  RL = OR { RL_Off, RL_On }
  DM = OR { DM_Off, DM_On, DM_Stopping }
  20 SM = OR { SM_Off, SM_On }
  SH = OR { SH_Off, SH_On }
  MotorM3 = OR {MotorM30, MotorM31, MotorM32, MotorM33}
  ButtonM3 = OR {ButtonM30, ButtonM31, ButtonM32}
  ButtonM4 = OR {ButtonM40, ButtonM41, ButtonM42}
  25 EmergencyM3 = OR {EmergencyM30, EmergencyM31}
  ProximityM = OR {ProximityM0, ProximityM1, ProximityM2, ProximityM3, ProximityM4, ↔
    ProximityM5, ProximityM6}
}

FB
30 {}
{fb_release, fb_press}
{
  [FB_Released fb_press FB_Pressed]
  [FB_Pressed fb_release FB_Released]
  35 }

BB
{}
{bb_release, bb_press}
40 {
  [BB_Released bb_press BB_Pressed]
  [BB_Pressed bb_release BB_Released]
}

RB
45 {}
{rb_release, rb_press}
{
  [RB_Released rb_press RB_Pressed]
  50 [RB_Pressed rb_release RB_Released]
}

FL
{fl_on, fl_off}
55 {}
{
  [FL_Off fl_on FL_On]
  [FL_On fl_off FL_Off]
}

60 BL
{bl_on, bl_off}
{}
{
  65 [BL_Off bl_on BL_On]
  [BL_On bl_off BL_Off]
}

70 RL
{rl_on, rl_off}
{}

```

```

75 {
  [RL_Off rl_on RL_On]
  [RL_On rl_off RL_Off]
}

BS
80 {}
{bs_release, bs_press}
{
  [BS_Released bs_press BS_Pressed]
  [BS_Pressed bs_release BS_Released]
85 }

BA
{}
{ba_empty, ba_ok}
90 {
  [BA_OK ba_empty BA_Empty]
  [BA_Empty ba_ok BA_OK]
}

95 PSF
{}
{psf_active, psf_inactive}
{
  [PSF_Inactive psf_active PSF_Active]
100 [PSF_Active psf_inactive PSF_Inactive]
}

PLF
{}
105 {plf_active, plf_inactive}
{
  [PLF_Inactive plf_active PLF_Active]
  [PLF_Active plf_inactive PLF_Inactive]
}

110 PSB
{}
{psb_active, psb_inactive}
{
  [PSB_Inactive psb_active PSB_Active]
115 [PSB_Active psb_inactive PSB_Inactive]
}

PLB
120 {}
{plb_active, plb_inactive}
{
  [PLB_Inactive plb_active PLB_Active]
  [PLB_Active plb_inactive PLB_Inactive]
125 }

RC
{}
{rc_start, rc_stop}
130 {
  [RC_Stop rc_stop RC_Stop]
  [RC_Stop rc_start RC_Start]
  [RC_Start rc_stop RC_Stop]
  [RC_Start rc_start RC_Start]
135 }

DM
{dm_enable_fw, dm_enable_bw, dm_fw, dm_fwslow, dm_fwstop, dm_bw, dm_bwslow, dm_bwstop, ←
  dm_stop}
{dm_error, dm_disable}
140 {
  [DM_Off dm_enable_fw DM_On]
  [DM_Off dm_enable_bw DM_On]
  [DM_On dm_error DM_Off]
  [DM_Stopping dm_error DM_Off]
145 [DM_Stopping dm_disable DM_Off]
  [DM_Stopping dm_enable_fw DM_On]
  [DM_Stopping dm_enable_bw DM_On]
  [DM_On dm_stop DM_Stopping]
  [DM_Off dm_stop DM_Stopping]
150 }

[DM_On dm_fw DM_On]
[DM_On dm_fwslow DM_On]

```

```

[DM_On dm_fwstop DM_On]
[DM_On dm_bw DM_On]
155 [DM_On dm_bwslow DM_On]
[DM_On dm_bwstop DM_On]
}

SM
160 {sm_enable, sm_disable}
{sm_error}
{
[SM_Off sm_enable SM_On]
[SM_On sm_disable SM_Off]
165 [SM_On sm_error SM_Off]
[SM_Off sm_error SM_Off]
}

SH
170 {sh_enable, sh_disable}
{sh_chdir, sh_error}
{
[SH_Off sh_enable SH_On]
[SH_On sh_disable SH_Off]
175 [SH_On sh_error SH_On]
[SH_On sh_chdir SH_On]
}

MM
180 {mm_active, mm_emergency, mm_reset}
{}
{
[MM_Emergency mm_reset MM_Reset]
[MM_Reset mm_emergency MM_Emergency]
185 [MM_Reset mm_active MM_Active]
[MM_Active mm_reset MM_Reset]
[MM_Active mm_emergency MM_Emergency]
}

190 MotorM3
{dm_fw, dm_fwslow, dm_fwstop, dm_bw, dm_bwslow, dm_bwstop, dm_stop, dm_enable_fw,
dm_enable_bw}
{sh_chdir, dm_disable, dm_error}
{
[MotorM30 dm_enable_fw MotorM31]
195 [MotorM31 dm_fw MotorM31]
[MotorM31 dm_fwslow MotorM31]
[MotorM31 dm_fwstop MotorM31]
[MotorM30 dm_enable_bw MotorM32]
[MotorM32 dm_bw MotorM32]
200 [MotorM32 dm_bwslow MotorM32]
[MotorM32 dm_bwstop MotorM32]
[MotorM32 dm_stop MotorM33]
[MotorM31 dm_stop MotorM33]
[MotorM33 dm_disable MotorM30]
205 [MotorM30 sh_chdir MotorM30]
[MotorM33 sh_chdir MotorM33]
[MotorM31 sh_chdir MotorM32]
[MotorM32 sh_chdir MotorM31]
[MotorM31 dm_error MotorM30]
210 [MotorM32 dm_error MotorM30]
[MotorM33 dm_error MotorM30]
}

ButtonM3
215 {mm_active, mm_emergency, mm_reset, dm_enable_fw}
{fb_press, fb_release}
{
[ButtonM30 fb_release ButtonM30]
[ButtonM30 fb_press ButtonM31]
220 [ButtonM31 fb_release ButtonM30]
[ButtonM30 mm_active ButtonM30]
[ButtonM30 mm_reset ButtonM30]
[ButtonM30 mm_emergency ButtonM30]
[ButtonM31 mm_reset ButtonM31]
225 [ButtonM31 mm_emergency ButtonM31]

[ButtonM31 mm_active ButtonM32]

[ButtonM32 fb_press ButtonM32]
230 [ButtonM32 fb_release ButtonM32]

```

```

[ButtonM32 dm_enable_fw ButtonM30]
[ButtonM32 mm_reset ButtonM30]
[ButtonM32 mm_emergency ButtonM30]
235 }

ButtonM4
{mm_active, mm_emergency, mm_reset, dm_enable_bw}
{bb_press, bb_release}
240 {
[ButtonM40 bb_release ButtonM40]
[ButtonM40 bb_press ButtonM41]
[ButtonM41 bb_release ButtonM40]
[ButtonM40 mm_active ButtonM40]
245 [ButtonM40 mm_reset ButtonM40]
[ButtonM40 mm_emergency ButtonM40]
[ButtonM41 mm_reset ButtonM41]
[ButtonM41 mm_emergency ButtonM41]

250 [ButtonM41 mm_active ButtonM42]

[ButtonM42 bb_press ButtonM42]
[ButtonM42 bb_release ButtonM42]

255 [ButtonM42 dm_enable_bw ButtonM40]
[ButtonM42 mm_reset ButtonM40]
[ButtonM42 mm_emergency ButtonM40]
}

260 EmergencyM3
{mm_emergency, mm_reset}
{sm_error, dm_error, sh_error, ba_empty, bs_press}
{
[EmergencyM30 mm_reset EmergencyM30]
[EmergencyM30 sm_error EmergencyM31]
265 [EmergencyM30 dm_error EmergencyM31]
[EmergencyM30 sh_error EmergencyM31]
[EmergencyM30 ba_empty EmergencyM31]
[EmergencyM30 bs_press EmergencyM31]
270 [EmergencyM31 sm_error EmergencyM31]
[EmergencyM31 dm_error EmergencyM31]
[EmergencyM31 sh_error EmergencyM31]
[EmergencyM31 ba_empty EmergencyM31]
[EmergencyM31 bs_press EmergencyM31]
275 [EmergencyM31 mm_emergency EmergencyM31]
[EmergencyM31 mm_reset EmergencyM30]
}

ProximityM
280 {dm_stop, dm_fw, dm_fwslow, dm_fwstop, dm_bw, dm_bwslow, dm_bwstop}
{}
{
[ProximityM0 dm_stop ProximityM0]
[ProximityM1 dm_stop ProximityM0]
285 [ProximityM2 dm_stop ProximityM0]
[ProximityM3 dm_stop ProximityM0]
[ProximityM4 dm_stop ProximityM0]
[ProximityM5 dm_stop ProximityM0]
[ProximityM6 dm_stop ProximityM0]

290 [ProximityM0 dm_fw ProximityM1]
[ProximityM2 dm_fw ProximityM1]
[ProximityM3 dm_fw ProximityM1]
[ProximityM4 dm_fw ProximityM1]
295 [ProximityM5 dm_fw ProximityM1]
[ProximityM6 dm_fw ProximityM1]

[ProximityM0 dm_fwslow ProximityM2]
[ProximityM1 dm_fwslow ProximityM2]
300 [ProximityM3 dm_fwslow ProximityM2]
[ProximityM4 dm_fwslow ProximityM2]
[ProximityM5 dm_fwslow ProximityM2]
[ProximityM6 dm_fwslow ProximityM2]

305 [ProximityM0 dm_fwstop ProximityM3]
[ProximityM1 dm_fwstop ProximityM3]
[ProximityM2 dm_fwstop ProximityM3]
[ProximityM4 dm_fwstop ProximityM3]
[ProximityM5 dm_fwstop ProximityM3]
310 [ProximityM6 dm_fwstop ProximityM3]

```

```

[ProximityM0 dm_bw ProximityM4]
[ProximityM1 dm_bw ProximityM4]
[ProximityM2 dm_bw ProximityM4]
315 [ProximityM3 dm_bw ProximityM4]
[ProximityM5 dm_bw ProximityM4]
[ProximityM6 dm_bw ProximityM4]

[ProximityM0 dm_bwslow ProximityM5]
320 [ProximityM1 dm_bwslow ProximityM5]
[ProximityM2 dm_bwslow ProximityM5]
[ProximityM3 dm_bwslow ProximityM5]
[ProximityM4 dm_bwslow ProximityM5]
325 [ProximityM6 dm_bwslow ProximityM5]

[ProximityM0 dm_bwstop ProximityM6]
[ProximityM1 dm_bwstop ProximityM6]
[ProximityM2 dm_bwstop ProximityM6]
[ProximityM3 dm_bwstop ProximityM6]
330 [ProximityM4 dm_bwstop ProximityM6]
[ProximityM5 dm_bwstop ProximityM6]
)

335 {FB_Released BB_Released RB_Released FL_Off BL_Off RL_Off BS_Released BA_OK PSF_Inactive ↔
    PLP_Inactive PSB_Inactive PLB_Inactive RC_Start DM_Off SM_Off SH_Off MM_Emergency ↔
    MotorM30 ButtonM30 ButtonM40 EmergencyM30 ProximityM0}
{ { FB_Released BB_Released RB_Released FL_On BL_On RL_Off BS_Released BA_OK PSF_Inactive ↔
    PLP_Inactive PSB_Inactive PLB_Inactive RC_Start DM_Off SM_Off SH_Off MM_Reset MotorM30 ↔
    ButtonM30 ButtonM40 EmergencyM30 ProximityM0} }
{MotorM3, ButtonM3, ButtonM4, EmergencyM3, ProximityM}

```

Specification C.3: Batch file of used logical expressions of state-based supervisor synthesis

```

*****
# File: radis.f
*****

5 # The rl_off flag can only be switched off if the status of the FullDrive is active or emergency.
--> rl_off ==> ( MM_Active | MM_Reset )

# The rl_on flag can only be switched on if the status of the FullDrive is emergency.
--> rl_on ==> MM_Emergency

10 # The fl_on flag can only be switched on if the status of the FullDrive is active or emergency.
--> fl_on ==> MM_Reset

# The fl_off flag can only be switched off if the status of the FullDrive is active or emergency.
15 --> fl_off ==> ( MM_Active | MM_Emergency )

# The bl_on flag can only be switched on if the status of the FullDrive is active or emergency.
--> bl_on ==> MM_Reset

20 # The bl_off flag can only be switched off if the status of the FullDrive is active or emergency.
--> bl_off ==> ( MM_Active | MM_Emergency )

*****
# File: Dupes.f
*****

# The sh_disable flag can only be switched off if the status of the FullDrive is active or emergency.
--> sh_disable ==> ( MM_Reset | MM_Emergency )

30 # The dm_stop flag can only be switched off if the status of the FullDrive is active or emergency and the prime flag is off.
--> dm_stop ==> ( ( MM_Reset | MM_Emergency ) & SH_Off )

# The sm_disable flag can only be switched off if the status of the FullDrive is active or emergency and the prime flag is off.
35 --> sm_disable ==> ( ( MM_Reset | MM_Emergency ) & DM_Off )

# The sm_enable flag can only be switched on if the status of the FullDrive is active.
--> sm_enable ==> ( MM_Active )

40 # The rlve Motor flag can only be switched on if the status of the FullDrive is active and the Prime Motor is on.

```

```

--> ( dm_enable_fw, dm_enable_bw ) ==> ( MM_Active & SM_On )

# The Write Stop of bridge may only be switched on if the status of the controller is ==
# Active, the Write Mode is on and the write buffer is on.
--> sh_enable ==> ( MM_Active & SM_On & DM_On )
45

# The Write Vector may only be active when the status of the controller is Active.
--> ( dm_fw, dm_fwslow, dm_fwstop, dm_bw, dm_bwslow, dm_bwstop ) ==> MM_Active

# The Multiplexer may only become active if the status vector, Write Vector and Write Program are
# enabled for the MM.
50 --> mm_active ==> ( SM_Off & DM_Off & SH_Off )

#####
# Unit: MM. u #
#####

55 # The Multiplexer may only switch to active if the forward button or the backward button (but not
# both) is pressed and the controller is not pressed.
--> mm_active ==> ( ( ( FB_Pressed & BB_Released ) | ( BB_Pressed & FB_Released ) ) & ~ (
  RB_Pressed ) )

# The Multiplexer may only switch to reset if the reset button is pressed.
60 --> mm_reset ==> RB_Pressed

#####
# Unit: MM. u #
#####

65 # The Multiplexer may only switch to reset and active only if the reset button is pressed ==
# and the status is not active.
( --> mm_reset, mm_active ) ==> ( BS_Released & BA_OK )

70 #####
# Priority Mode = #
#####

75 # The Multiplexer may only switch to forward if the forward button is pressed and the status is not active.
--> dm_fwstop ==> ( RC_Stop | PSP_Active )

# The Multiplexer may only switch to backward if the backward button is pressed and the status is not active.
--> dm_bwstop ==> ( RC_Stop | PSB_Active )
80

# The Multiplexer may only switch to forward if the forward button is pressed and the status is not active.
# The status is not active and the status is not active.
( --> dm_fwslow, dm_fw ) ==> ( RC_Start & PSP_Inactive )

# The Multiplexer may only switch to backward if the backward button is pressed and the status is not active.
# The status is not active and the status is not active.
85 ( --> dm_bwslow, dm_bw ) ==> ( RC_Start & PSB_Inactive )

# The Multiplexer may only switch to forward if the forward button is pressed and the status is not active.
--> dm_fwslow ==> PLF_Active

90 # The Multiplexer may only switch to backward if the backward button is pressed and the status is not active.
--> dm_fw ==> PLF_Inactive

# The Multiplexer may only switch to forward if the forward button is pressed and the status is not active.
--> dm_bwslow ==> PLB_Active

95 # The Multiplexer may only switch to backward if the backward button is pressed and the status is not active.
--> dm_bw ==> PLB_Inactive

```

Appendix D

Implementation

D.1 Communication delay example

In this section, the χ -model is presented that simulates a communication delay between a supervisor and a component. Process P is a model of the plant model of a timer, depicted in Figure 5.6a and consists of modes **Off** and **On**. Process B is a model of a buffer, modelled as a conveyor, which delays all incoming events from process S for 0.1 time unit before sending it to process P . Process S is a model of a supervisory controller that sends appropriate control actions (*start* and *reset*) to the timer. Simulation of this χ -model results in the message sequence chart of Figure 5.7.

Specification D.1: χ model of a communication delay example

```

// Specification D.1:  $\chi$  model of a communication delay example
//
// Process P: timer
// Process B: buffer
// Process S: supervisor

5  from standardlib import *

// Process P: timer
10 proc P(chan a?, b!: string) =
  | mode Off = ( a?x; ( x = "timer" -> skip; On )
    )
    , On = ( a?x; x = "reset" -> skip; Off
      | time >= 1.00 and time <= 1.01 -> b!"timeout"; Off
    )
    , var x: string = ""
15 :: Off
  ||

// Process B: buffer
20 proc B(chan a?, b!: string, val t: real) =
  | var ys: [(string,real)] = [], x: string = ""
  :: * ( a?x; ys:= ys ++ [ (x,time + t) ]
    ; len(ys) > 0
    -> ( a?x; ys:= ys ++ [ (x,time + t) ]
      | delay hd(ys).1 - time; b!!hd(ys).0; ys:= tl(ys)
    )
  )
25 ||

// Process S: supervisor
30 proc S(chan a!, b?: string) =
  | mode S0 = ( time >= 0.00 and time <= 0.01 -> skip
    ; a!"timer"
    ; S1
  )
35 , S1 = ( time >= 1.05 and time <= 1.06 -> skip
    ; a!"reset"
    ; S0
  )
  | b?y
40 ; S0

```



```

    )
    , var y: string = ""
    :: S0
  ||
45
  // Complete model: ////////////////////////////////////////////////////////////////////
  model M() =
  || chan a,b,c: string
  :: SM(a,b) || B(b,c,0.1) || S(a,c)
50 ||

```

D.2 Explanation lookup tables

In this section, an explanation is given how an automaton is converted to a lookup table. Consider Specification D.2, which contains a lookup table of the plant model of the drive motor (see Figure A.7). Lines 1-13 specifies the alphabet of the automaton. Furthermore, the boolean variable after each event name indicates if this event is controllable (TRUE) or uncontrollable (FALSE).

Then, line 15-27 specifies the transition structure of the automaton of the drive motor. Each row in this lookup table corresponds with the same row of the alphabet list of line 1-13 and each column corresponds with the originating state of this particular event in the automaton. Note that the automaton of the drive motor contains three states, which corresponds with the number of columns. The value of each index corresponds with the destination state of this transition with this particular event in this particular originating state. If no transition is possible with a particular event from a particular state, 0xFFFF is listed.

Specification D.2: Example of an automaton specified with a lookup table

```

static const TAutomatonEvent      SupervisorDataDMAlphabet[] = {
  {SUPERVISOR_DATA_EVENT_DM_BW, TRUE},
  {SUPERVISOR_DATA_EVENT_DM_STOP, TRUE},
  {SUPERVISOR_DATA_EVENT_DM_FWSTOP, TRUE},
5  {SUPERVISOR_DATA_EVENT_DM_ENABLE_BW, TRUE},
  {SUPERVISOR_DATA_EVENT_DM_BWSLOW, TRUE},
  {SUPERVISOR_DATA_EVENT_DM_FWSLOW, TRUE},
  {SUPERVISOR_DATA_EVENT_DM_BWSTOP, TRUE},
  {SUPERVISOR_DATA_EVENT_DM_ENABLE_FW, TRUE},
10 {SUPERVISOR_DATA_EVENT_DM_DISABLE, FALSE},
  {SUPERVISOR_DATA_EVENT_DM_ERROR, FALSE},
  {SUPERVISOR_DATA_EVENT_DM_FW, TRUE},
};

15 static const uns16              SupervisorDataDMTransitionTable[] = {
  0xFFFF, 1,      0xFFFF,
  2,      2,      0xFFFF,
  0xFFFF, 1,      0xFFFF,
  1,      0xFFFF, 1,
20  0xFFFF, 1,      0xFFFF,
  0xFFFF, 1,      0xFFFF,
  0xFFFF, 1,      0xFFFF,
  1,      0xFFFF, 1,
  0xFFFF, 0xFFFF, 0,
25  0xFFFF, 0,      0,
  0xFFFF, 1,      0xFFFF,
};

```

D.3 CIF to C conversion

In Specification D.3, the source code is listed that converts CIF automata to lookup tables in C. Furthermore, the BDD information is also converted to a BDD structure in C. The source code of this conversion takes one input argument, which is the location of the CIF-model containing the supervisor. The source code is compatible with CIF revision 5501.

Specification D.3: Python script of CIF to C conversion

```

"""
Automatic conversion of CIF to C code. Generated by Sphenix's routine. Input will
find back the file structure in C.
"""
5 import os
import sys

#####
# Read [1] file.
10 #

# Make sure the translated version is used (as if the Python path).
CHINETICS_VERSION = '5501'
PY_VERSIONS = ['2.3', '2.4', '2.5', '2.6']
15 site_packages = None
for py_ver in PY_VERSIONS:
    path = "/opt/se/chi/chinetics/trunk-rev%s/lib/python%s/site-packages" % (
        CHINETICS_VERSION, py_ver)
    if os.path.isdir(path):
        site_packages = path
20 break
assert site_packages is not None

if not site_packages.startswith("%SITE") and site_packages not in sys.path:
    # Add it to version, path's initial path.
25 sys.path = [site_packages] + sys.path

from chinetics.core import exceptions
from chinetics.languages.common.hybrid import expression

30 from chinetics.languages.cif.xml import cif_xml_reader
from chinetics.languages.cif.core_cif import cifaction, cifterm, \
    core_cif_tree, cifmodel

try:
35 set
except NameError:
    from sets import Set as set, ImmutableSet as frozenset

def expr2cpp(expr):
40 if isinstance(expr, expression.And):
    return '%s && %s' % ( expr2cpp(expr.left_child),
        expr2cpp(expr.right_child) )

    elif isinstance(expr, expression.Or):
45 return '%s || %s' % ( expr2cpp(expr.left_child),
        expr2cpp(expr.right_child) )

    elif isinstance(expr, expression.Not):
        return '!(%s)' % expr2cpp(expr.child)
50 elif isinstance(expr, (expression.Variable, expression.MinusedVariable)):
        return 'currentState[ %s ]' % expr.name

    elif isinstance(expr, expression.In):
55 strvar = expr2cpp(expr.left_child)
    strelem = [expr2cpp(e) for e in expr.right_child.items]
    strin = ' || '.join(['%s == %s' % (strvar, e) for e in strelem])
    return '%s' % strin

60 elif isinstance(expr, expression.Literal):
    exprstr = str(expr)
    if isinstance(expr.value, int):
        return '%d' % expr.value
    if exprstr == "false":
        return '%s' % exprstr
65 elif exprstr == "true":
        return '%s' % exprstr
    else:
        return '%s' % expr

70 elif isinstance(expr, expression.Minus):
        return '-%s' % expr2cpp(expr.child)

    elif isinstance(expr, expression.Equal):
75 return '%s == %s' % ( expr2cpp(expr.left_child),
        expr2cpp(expr.right_child) )

    elif isinstance(expr, expression.Conditional):

```

```

cond_expr = ''
80 for alt in expr.alts:
    if cond_expr == '':
        cond_expr = expr2cpp(alt)
    else:
        cond_expr = cond_expr % expr2cpp(alt)
85 return cond_expr % 'CONDEXPRERROR'

elif isinstance(expr, expression.ConditionalAlternative):
    invert = False
    guard_expr = expr.guard
90 while isinstance(guard_expr, expression.Not):
    invert = not invert
    guard_expr = guard_expr.child
    if invert:
        return '%s ? %s : %s' % (expr2cpp(expr.guard),
95                               expr2cpp(expr.value))
    else:
        return '%s ? %s : %s' % (expr2cpp(expr.guard),
                               expr2cpp(expr.value))
elif isinstance(expr, (expression.Tuple, expression.Array, expression.Set)):
100 if isinstance(expr, expression.Set):
    if len(expr.items) == 0:
        return '%d, emptystring%s' % (len(expr.items), ', '.join([expr2cpp(i) for i in
        in expr.items]))
    else:
        return '%d, %s' % (len(expr.items), ', '.join([expr2cpp(i) for i in expr.
        items]))
105 if isinstance(expr, expression.Array):
    return '%d, %s' % (len(expr.items), ', '.join([expr2cpp(i) for i in expr.items]))
    else:
        return '%s' % ', '.join([expr2cpp(i) for i in expr.items])
elif isinstance(expr, expression.Call):
110 assert isinstance(expr.funcexpr, expression.Function)
    assert expr.funcexpr.friendly_name == 'eval_bdd'
    assert len(expr.args) == 2
    return 'evaluateBdd(EVENTNAME)'
else:
115 raise ValueError('Unexpected expression: %s' % expr)
return s

reader = cif_xml_reader.CifXmlReader()
spec = reader.read_cif_xml_file(sys.argv[1], [])
120 auts = core_cif_tree.AtomicAutCollector().collect(spec)

/* ***** */
/* ***** */
/* ***** */
125 SDC = open ('SupervisorData.c', 'w')
SDC.write('// *INDENT-OFF*\n')
SDC.write('#include <stddef.h>\n')
SDC.write ('#include "Typedefs.h"\n')
130 SDC.write ('#include "SupervisorData.h"\n')
SDC.write ('#include "Debug.h"\n')

autlocdict=[]
statedict = {}
135 for aut in auts:
    if aut.name != 'bdd':
        *
        SDC.write ('\n')
140 SDC.write ('static const TAutomatonEvent SupervisorData')
SDC.write (aut.name)
SDC.write ('Alphabet[] = {\n')
    alphabetdict = {}
    k = 0
145 for lbl in aut.labels:
        alphabetdict[lbl.name] = k
        k = k + 1
        SDC.write ('    {SUPERVISOR_DATA_EVENT_'
SDC.write (lbl.name.upper())
150 SDC.write (', ')
        if lbl.controllable.value:
            SDC.write ('TRUE')
        else:
            SDC.write ('FALSE')
155 SDC.write (',)\n')
SDC.write (');\n')

```

```

SDC.write ('\n')
SDC.write ('static const uns16                               SupervisorData')
SDC.write (aut.name)
160 SDC.write ('TransitionTable[] = {\n')
    # Make 'action' dictionary
    locdict = {}
    i = 0
    for loc in aut.locations:
165         locdict[loc.name]=i
            i = i + 1
            statedict[loc.name]= aut.name
    autlocdict.append(locdict)
    TransitionTable = []
170 for j in range(len(aut.locations)*len(aut.labels)):
        TransitionTable.append('0xFFFF')
    for stt, edgs in aut.edges.iteritems():
        for edg in edgs:
            TransitionTable[alphabetdict[edg.action.label]*len(aut.locations)+locdict[
175             edg.sourceLocation.name]]=locdict[edg.targetLocation.name]
    for i in TransitionTable:
        SDC.write (str(i))
        SDC.write (' ')
    SDC.write ('\n')
    SDC.write (');\n')
180
# End of data of AUT1=1
SDC.write ('const TAutomatonConfig                               SupervisorDataAutomatonConfig[ ←
    SUPERVISOR_DATA_NR_AUTOMATA] = {\n')
for aut in auts:
    if aut.name != 'bdd':
185         locdict = {}
            i = 0
            for loc in aut.locations:
                locdict[loc.name]=i
                i = i + 1
190         SDC.write('    {\n')
            SDC.write('        SupervisorData')
            SDC.write(aut.name)
            SDC.write('Alphabet,\n')
            SDC.write('        sizeof(SupervisorData')
195         SDC.write(aut.name)
            SDC.write('Alphabet) / sizeof(TAutomatonEvent),\n')
            SDC.write('        SupervisorData')
            SDC.write(aut.name)
            SDC.write('TransitionTable,\n')
200         SDC.write('        sizeof(SupervisorData')
            SDC.write(aut.name)
            SDC.write('TransitionTable) / sizeof(uns16),\n')
            SDC.write('        ')
            SDC.write(str(locdict[aut.initLocation.name]))
205         SDC.write(');\n')
            SDC.write('        (sizeof(SupervisorData')
            SDC.write(aut.name)
            SDC.write('TransitionTable) / sizeof(uns16)) /\n')
            SDC.write('        (sizeof(SupervisorData')
210         SDC.write(aut.name)
            SDC.write('Alphabet) / sizeof(TAutomatonEvent))\n')
            SDC.write('        },\n')
SDC.write(');\n')
autalph = set()
215 for aut in auts:
    if aut.name != 'bdd':
        for lbl in aut.labels:
            autalph.add(lbl)
SDC.write('const TAutomatonEvent                               SupervisorDataAlphabet[ ←
    SUPERVISOR_DATA_ALPHABET_SIZE] = {\n')
220
for x in autalph:
    SDC.write('    {SUPERVISOR_DATA_EVENT_')
    SDC.write(x.name.upper())
    SDC.write(' ', ' ')
225     if x.controllable.value:
        SDC.write('TRUE')
    else:
        SDC.write('FALSE')
    SDC.write(');\n')
230 SDC.write(');\n')

#####
# End of data of AUT1=1

```



```

        SDC.write('\n')
315     SDC.write('const TBDDNode SupervisorData')
        SDC.write(variable.replace('bdddata_', ''))
        SDC.write('Nodes[] = {\n')
        SDC.write('    {NULL, 0, 0, 0},\n')
        SDC.write('    {NULL, 0, 0, 0},\n')
320     for k,l in enumerate(TrueEdge):
        print k
        SDC.write('    (')
        SDC.write('SupervisorData')
        SDC.write(variable.replace('bdddata_', ''))
325     SDC.write('Node')
        SDC.write(str(k+2))
        SDC.write('TrueStates')
        SDC.write(', sizeof(')
        SDC.write('SupervisorData')
        SDC.write(variable.replace('bdddata_', ''))
330     SDC.write('Node')
        SDC.write(str(k+2))
        SDC.write('TrueStates')
        SDC.write(')/sizeof(TBDDTrueState), ')
335     SDC.write(l)
        SDC.write(', ')
        SDC.write(FalseEdge[0])
        FalseEdge.pop(0)
        SDC.write(');\n')
340     SDC.write(');\n')
        SDC.write('\n')

no_BDDs = len(EventName)
if no_BDDs is not 0:
345     SDC.write('const TBDDConfig SupervisorDataBDD[SUPERVISOR_DATA_NR_BDDS] = {\n')
        for i,j in enumerate(EventName):
            SDC.write('    {SUPERVISOR_DATA_EVENT_'
                SDC.write(j.upper())
                SDC.write(', ')
350            SDC.write(InitialNode[0])
                InitialNode.pop(0)
                SDC.write(', SupervisorData')
                SDC.write(j)
                SDC.write('Nodes, sizeof(SupervisorData')
355            SDC.write(j)
                SDC.write('Nodes) / sizeof(TBDDNode)},\n')
            SDC.write(');\n')
        SDC.write('\n')
        SDC.write('// *INDENT-ON*\n')
360     SDC.close

print 'SupervisorData.c written succesfully'

#####
365 * * * * *
= * * * * *
=

SDH = open ('SupervisorData.h', 'w')

370 SDH.write('#ifndef SUPERVISORDATA_H\n')
        SDH.write('#define SUPERVISORDATA_H\n')
        SDH.write('\n')
        SDH.write('#ifdef __cplusplus\n')
        SDH.write('// *INDENT-OPF*\n')
375 SDH.write('extern "C" {\n')
        SDH.write('#endif\n')
        SDH.write('\n')
        SDH.write('#include "Typedefs.h"\n')
        SDH.write('#include "Automaton.h"\n')
380 SDH.write('#include "BDD.h"\n')
        SDH.write('\n')
        SDH.write('#define SUPERVISOR_DATA_NR_AUTOMATA      ')
        SDH.write(str(len(auts)))
        SDH.write('\n')
385 SDH.write('#define SUPERVISOR_DATA_NR_BDDS      ')
        SDH.write(str(no_BDDs))
        SDH.write('\n')
        SDH.write('#define SUPERVISOR_DATA_ALPHABET_SIZE      ')
        SDH.write(str(len(autalph)))
390 SDH.write('\n')
        i = 0
        for x in autalph:
            SDH.write('#define SUPERVISOR_DATA_EVENT_'

```

```

SDH.write(x.name.upper())
395 SDH.write(' ')
SDH.write(str(i))
SDH.write('\n')
i = i + 1
SDH.write('#define SUPERVISOR_DATA_NO_EVENT          255\n')
400 SDH.write('\n')
SDH.write('extern const TAutomatonConfig              SupervisorDataAutomatonConfig[ ↔
SUPERVISOR_DATA_NR_AUTOMATA];\n')
SDH.write('extern const TAutomatonEvent              SupervisorDataAlphabet[ ↔
SUPERVISOR_DATA_ALPHABET_SIZE];\n')
if no_BDDs is not 0:
SDH.write('extern const TBDDConfig                    SupervisorDataBDD[SUPERVISOR_DATA_NR_BDDS ↔
];\n')
405 SDH.write('#ifdef __cplusplus\n')
SDH.write(')\n')
SDH.write('// *INDENT-ON*\n')
SDH.write('#endif')
SDH.write('\n')
410 SDH.write('#endif // SUPERVISORDATA_H\n')
SDH.close

print 'SupervisorData.h written succesfully'

```
