Supervisory control in health care systems

Supervisory Control in Health Care Systems

R.J.M. Theunissen

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Reproduction: Print Service Ede

A catalogue record is available from the Eindhoven University of Technology Library

ISBN: 978-90-386-3789-1

The research described in this thesis was carried out as part of the DARWIN project at Philips Healthcare under the responsibilities of the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economics Affairs under the BSIK program.

Supervisory Control in Health Care Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op maandag 2 maart 2015 om 16.00 uur

door

Rolf Jozef Maria Theunissen

geboren te Nuland

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr. L.P.H. de Goey
1e promotor:	prof.dr.ir. J.E. Rooda
2e promotor:	prof.dr.ir. J.H. van Schuppen
copromotor:	dr.ir. D.A. van Beek
leden:	prof.dr.ir. A.A. Basten
	Prof.Dr.-Ing. S. Engell (Technische Universität Dortmund)
	prof.dr. J.C. van De Pol (UT)
adviseur:	dr.ir. M.J.A.M. van Helvoort (Philips Healthcare)

Preface

This thesis is the final results of my Ph.D. research at the Systems Engineering Group of the Mechanical Engineering Department at the Eindhoven University of Technology.

My Ph.D. research has been performed as part of the DARWIN research project. The DARWIN project was a collaboration between: The Embedded Systems Institute[1] (ESI), Philips Healthcare MRI, Philips Research, Eindhoven University of Technology and four other dutch universities. All cases used in the project were based on real-life industrial problems of Philips Healthcare. This offered me the interesting challenge to use tools and apply techniques that are developed within academic research on industrial problems. Moreover, the problems that I encountered while applying the techniques were input to further develop these academic tools and techniques. This offered me a nice balance between industrial relevance and academic research.

I would like to thank all the persons who contributed to this thesis. In particular I thank my supervisors, Bert van Beek and professor Koos Rooda, for the supervision of my Ph.D. research work and for all their time and efforts. Without their constant support and advice, this thesis would not have appeared in its current form. I also would like to thank my second promotor, professor Jan van Schuppen, for his valuable comments on my thesis. I also thank the members of my committee, for reviewing the manuscript of this thesis and giving many valuable comments.

Next, I would like to thank people from Philips Healthcare for posing challenging industrial problems, sharing their expertise, coaching and supporting this research. In particular, John van der Koijk for providing the patient support case, and the experimental setup to test our generated supervisors; and Geran Peeren for providing the patient communication case and for showing us the importance of intuitive, user-friendly specifications combined with interactive simulation and visualization to support rapid iterative development of control requirements.

The DARWIN project brought together an interesting mix of people from different backgrounds to form a team. I enjoyed being part of that team and would like to thank all DARWIN project members, not only for their cooperation and fruitful discussions, but also for all the fun we had during social events. Special thanks to our DARWIN

[1] Currently integrated into TNO as Embedded Systems Innovation by TNO

coaches: Pierre van den Laar, David Watts, Teade Punter and Pierre America for the coordination of the project, their support and valuable advices.

I also thank my colleagues and the students at the Systems Engineering Group of the Eindhoven University of Technology for the pleasant working atmosphere. Especially, I thank Dennis Hendriks, Albert Hofkamp and Ramon Schiffelers for providing the required tooling for supervisory control synthesis and the associated tool-chain for simulation and visualization of synthesized controllers in combination with models of the uncontrolled system. I would like to thank, Jason Markovski, Mihaly Petreczky, Michel Reniers and Rong Su, for their help with the theoretical aspects of, and discussions about supervisory control theory. Furthermore, I would like the thank all the students of the Systems Engineering Group with who I had discussions about supervisory control, these discussions provided much insight into the supervisory control domain and problems that occur while using supervisory control theory.

Finally, I want to thank my family and friends for their interest and support during these years. A special word of gratitude goes to my girlfriend Maaike for her love and understanding during the past years.

Rolf Theunissen
January 2015

Summary

Control systems can be divided into multiple layers: regulative control, supervisory control and user interfacing. Regulative control assures that a system reaches the desired position or the desired state in the desired way. These controllers are typically designed in the continuous-time domain. This is the domain of classical control theory. Supervisory control assures that a system correctly performs its function by determining and executing allowed sequences of tasks on (in)dependent resources. These controllers are typically designed in the discrete-event domain. User interfacing is provided to interact with the users of the system. A user of the system can be a human operator, but also another system. This thesis focuses on the design of controllers for the supervisory control layer.

In the traditional approach to supervisory controller design, behavioral requirements are informally specified by domain experts, and software is coded by software experts. This leads to long development cycles, and to code and requirements that are difficult to develop, debug, maintain, and adapt. Observed erroneous behavior of the system under test can be caused by, among others, ambiguous or inconsistent control requirements, miscommunication between domain engineer and software coder, and errors in the control code. This can be especially problematic when the functionality of existing products evolves over time, such as in the case of Magnetic Resonance Imaging (MRI) scanners, that can be maintained at the level of new, state of the art, systems by periodic upgrades over a period of approximately ten years.

To address these issues, this thesis proposes the use of existing supervisory control theory for formal *specification* of the control requirements and the uncontrolled plant, and *generation* of the control code by means of supervisory control synthesis. By combining supervisory control synthesis with the well-known principles of model-based engineering, 'synthesis-based engineering' is obtained.

Using supervisory control theory has the following advantages: The engineering process changes from implementing and debugging the control code, to designing and debugging the behavioral requirements. The models of the uncontrolled system and the control requirements are unambiguous, leaving no room for different interpretations. The synthesized supervisors are suitable for code generation, which makes the design relatively independent of the implementation technology. Finally, changes in the control requirements can be realized quickly without introducing

errors. As a result, the development time of the supervisors decreases while the quality increases.

Model-based analysis techniques can be used to get early feedback on the system under development. The control requirements can be validated by means of interactive simulation and visualization of the synthesized supervisor together with the plant model. The use of supervisory control synthesis means that changes in the requirements, resulting from these validation steps, can be incorporated in the supervisor quickly.

The synthesis-based engineering design process is applied to two cases in the Philips Healthcare MRI scanner patient environment, namely the patient support table, and the patient communication system. Different kinds of supervisory control theory are applied: event-based supervisory control, state-based supervisory control using automata, and state-based supervisory control using automata with variables, the so-called 'extended automata'.

In event-based supervisory control, the plant and the control requirements are both modeled by automata. The patient support table case has shown that the evolvability of the control system can be improved by dividing the plant model and control requirements into small, mostly independent, specifications. The plant models are divided into models for the actuators, sensors, and for structural restrictions. The control requirements are divided into requirements for the individual components and for requirements defining their interaction. This division is facilitated by splitting of events, such as stop events, into a number of independent sub-events, and by introducing internal events to allow modeling of various modes of operation. In this way, model errors can more easily be detected, and attributed to specific parts of the specification. The improved evolvability is demonstrated by means of an actual redesign of the control requirements to incorporate a user change request.

The generated real-time implementation was extensively tested on an actual patient support system: several operating procedures, that are used in practice, were carried out. In addition, attempts were made to generate erroneous behavior by means of very rapid pressing of buttons and switches, and by intentionally giving illegal commands. In all cases, the control system reacted as desired. The system was also operated and tested by Philips employees, but no errors were found.

State-based supervisory control is introduced as an extension of event-based supervisory control: in addition to automata, plants and control requirements can also be specified using state exclusion predicates, and state-event exclusion predicates. Experience at Philips Healthcare has shown that state-based control requirements are intuitive for both domain experts and software experts, since they closely match the view of the systems in terms of states, transitions between states, and restrictions on allowable states and state-transitions. To allow a straightforward comparison of event-based and state-based supervisory control specifications, the same patient support table has been modeled in detail using event-based and state-based specifications. A comparison shows that the combination of state-based and event-based requirements leads to far more intuitive specifications than are possible using event-based specifications alone. In addition, where event-based supervisory control in

general allows only a single initial state, the state-based supervisory control synthesis algorithm of [44] allows an arbitrary number of initial states. This has proved to be essential for actual real-time control of the patient support system, because it allows activation of the controller in any initial state of the physical system. Finally, the state-based control requirements that are defined in this thesis not only facilitate the exclusion of unsafe behavior by means of state-event exclusion and state exclusion (safety), but also the inclusion of required safe behavior by means of state-event inclusion (liveness).

In existing 'extended automata'-based supervisory control, automata are extended with variables, guards and updates. The patient communication system application illustrates how the use of variables is essential for intuitive modeling of various modes of operation of the system. To support systematic, modular specification of the patient communication system for supervisory control synthesis, observers that record sequences of events in terms of states are shown to be essential. An advantage of this approach is that it facilitates state-based output. That is, the definition of the values of output variables as a function of the state of the control system.

The applications discussed in this thesis show that synthesis-based engineering has major advantages compared to conventional supervisory control system design. As a result, Philips Healthcare has started to investigate the use of supervisory control synthesis for all major components of MRI scanners. Regarding the tool chain, also considerable progress is made: a new supervisory control tool chain, based on the tool chain that is proposed in this thesis, is currently implemented in the Systems Engineering Group.

Samenvatting

Besturingssystemen kunnen worden onderverdeeld in meerdere lagen: regelaars, besturingen, en gebruikersinterface. Regelaars zorgen ervoor dat het systeem de gewenste positie of toestand op een gewenste manier bereikt. Regelaars worden typisch ontwikkeld in het continue-tijd domein. Dit is het domein van de klassieke regeltechniek. Besturingen zorgen ervoor dat het systeem op een correcte manier zijn functie uitvoert door het bepalen en uitvoeren van een reeks van taken op (on)afhankelijke resources. De gebruikersinterface verzorgt de interactie met de gebruiker met het systeem. De gebruiker kan een mens zijn, maar ook een ander systeem. Dit proefschrift concentreerd zich op het ontwerp van besturingen.

In de traditionele aanpak van het ontwikkelen van besturingen worden gedragseisen informeel gespecificeerd door domeinexperts en wordt software gecodeerd door software-experts. Dit leidt tot lange ontwikkelcycli, en tot software en eisen die moeilijk zijn te ontwikkelen, debuggen, onderhouden, en aan te passen. Foutief gedrag in de testfase kan zijn oorzaak hebben in, onder andere, ambigue of inconsistente besturingseisen, miscommunicatie tussen domeinexpert en software-expert, en fouten in de besturings-software. Dit kan extra problematisch zijn als de functionaliteit van bestaande producten evolueert in de tijd, zoals in het geval van MRI scanners, die op het niveau van nieuwe, state of the art, systemen worden gebracht door periodieke upgrades over een periode van ongeveer tien jaren.

Om het ontwerp van programmatuur voor machines aan te pakken, introduceert dit proefschrift de formele *specificatie* van besturingseisen en de *generatie* van besturingen aan de hand van de bestaande theorie van synthese van besturingen. De combinatie van synthese van besturingen met de principes van modelgebaseerde ontwikkeling leidt tot 'synthese-gebaseerde ontwikkeling'.

Het gebruik van de theorie van synthese van besturingen heeft de volgende voordelen: Het ontwikkelproces verandert van het implementeren en debuggen van besturingen naar het ontwikkelen en debuggen van de besturingseisen. De modellen van het onbestuurde systeem en de besturingseisen zijn ondubbelzinnig, en laten geen ruimte voor verschillende interpretaties. De gesynthetiseerde besturingen zijn geschikt voor het genereren van implementatiesoftware, wat het ontwerp relatief onafhankelijk maakt van de implementatietechnologie. Tenslotte kunnen wijzigingen in de besturingseisen relatief snel en zonder fouten worden gerealiseerd. Dit zal

leiden tot een afname van de ontwikkeltijd van besturingen, terwijl de kwaliteit zal toenemen.

Modelgebaseerde analysetechnieken kunnen worden gebruikt om snel feedback te krijgen op het in ontwikkeling zijnde systeem. De besturingseisen kunnen worden gevalideerd door middel van interactieve simulatie en visualisatie van de gesynthetiseerde besturing samen met het model van het onbestuurde systeem. Het gebruik van synthese van besturingen betekent dat wijzigingen in de besturingseisen, als gevolg van de modelgebaseerde validatie, snel kunnen worden gerealiseerd.

Het synthese-gebaseerde ontwerpproces is toegepast op twee gevallen op het gebied van de patiënt-ondersteuning voor MRI-scanners bij Philips Healthcare, namelijk de patiënt-tafel en het patiëntcommunicatie-systeem. Verschillende vormen van de theorie van synthese van besturingen worden toegepast: event-gebaseerde synthese, toestand-gebaseerde synthese met behulp van automaten en toestand-gebaseerde synthese met behulp van automaten en variabelen, de zogenaamde 'uitgebreide automaten'.

In event-gebaseerde synthese van besturingen worden zowel het onbestuurde systeem als de besturingseisen gemodelleerd met automaten. De case van de patiënt-tafel heeft laten zien dat de evolueerbaarheid van de besturing verbeterd kan worden door het model van het onbestuurde systeem en de besturingseisen op te delen in kleine, grotendeels onafhankelijke, specificaties. De modellen van het onbestuurde systeem worden opgedeeld in modellen voor de actuatoren, voor de sensoren, en voor de structurele beperkingen. De besturingseisen worden opgedeeld in de eisen voor de individuele componenten en voor de eisen die hun interactie bepalen. Deze opdeling wordt mogelijk gemaakt door de verdeling van events, zoals stop-events, in een aantal onafhankelijke sub-events, en door de introductie van interne events om het modelleren van verschillende werkingsmodi mogelijk te maken. Op deze wijze kunnen modelleerfouten eenvoudiger worden gedetecteerd, en worden toegewezen aan specifieke delen van de specificatie. De verbeterde evolueerbaarheid wordt aangetoond met behulp van een herontwerp van de besturingseisen om een daadwerkelijke wijziging van een gebruikerseis te realiseren.

De gegenereerde real-time implementatie is uitgebreid getest op een daadwerkelijke patiënt-tafel: diverse gebruiksprocedures uit de praktijk zijn uitgevoerd. Daarnaast is er getracht foutief gedrag op te roepen door drukknoppen en schakelaars zeer snel te activeren, en door opzettelijk foutieve opdrachten te geven. In alle gevallen reageerde de besturing zoals gewenst. Het systeem is ook bediend en getest door Philips personeel, waarbij evenmin fouten zijn gevonden.

Toestand-gebaseerde synthese van besturingen is geïntroduceerd als een extensie van event-gebaseerde synthese van besturingen. Deze extensie maakt het mogelijk om het onbestuurde systeem en de besturingseisen, behalve met automaten, ook met behulp van toestand-uitsluitingspredicaten en toestand-event-uitsluitingspredicaten te specificeren. Ervaring bij Philips Healthcare heeft aangetoond dat toestandsgebaseerde besturingseisen intuïtief zijn voor zowel domeinexperts als software-experts, aangezien zij sterk overeenkomen met de wijze van beschouwen van systemen in termen van toestanden, overgangen tussen toestanden, en beperkingen op toe-

gestane toestanden en toestandsovergangen. Om een rechtstreekse vergelijking tussen event-gebaseerde en toestand-gebaseerde specificaties mogelijk te maken, is dezelfde patiënt-tafel in detail gemodelleerd met zowel event-gebaseerde als toestand-gebaseerde specificaties. Een vergelijking toont aan dat de combinatie van toestand-gebaseerde en event-gebaseerde specificaties tot aanzienlijk intuïtievere specificaties leidt dan mogelijk zijn met uitsluitend event-gebaseerde specificaties. Bovendien, waar event-gebaseerde specificaties in het algemeen slechts een enkele begintoestand toestaan, staat het toestand-gebaseerde synthese algoritme van [44] een willekeurig aantal begintoestanden toe. Dit is essentieel gebleken voor de daadwerkelijke real-time besturing van de patiënt-tafel, omdat het de activering van de besturing mogelijk maakt in iedere willekeurige begintoestand van het fysieke systeem. Tenslotte maken de toestand-gebaseerde specificaties die zijn gedefinieerd in dit proefschrift het mogelijk om niet alleen het onveilige gedrag uit te sluiten met behulp van toestand-event-uitsluiting en toestand-uitsluiting, maar ook om het gewenste veilige gedrag te vereisen met behulp van toestand-event-inclusie.

In synthese van besturingen met behulp van 'uitgebreide automaten' worden de automaten uitgebreid met variabelen, condities en updates. De patiëntcommunicatietoepassing laat zien hoe essentieel het gebruik van variabelen is voor het intuïtief modelleren van de verschillende werkingsmodi van het systeem. Toestandwaarnemers worden geïntroduceerd om het systematisch en modulair ontwerpen van besturings-specificaties te ondersteunen. Deze toestandwaarnamers leggen sequenties van events vast in termen van toestanden. Een voordeel van deze aanpak is dat het toestand-gebaseerde output mogelijk maakt. Hiermee kan de waarde van outputvariabelen worden gedefinieerd als functie van de toestand van de besturing.

De behandelde toepassingen laten zien dat het synthese-gebaseerde ontwerpproces essentiële voordelen biedt ten opzichte van het conventionele ontwerpproces van besturingen. Als gevolg hiervan is Philips Healthcare begonnen met onderzoek naar het toepassen van synthese van besturingen voor alle hoofdonderdelen van MRI scanners. Ten aanzien van de ondersteunende gereedschappen wordt ook aanzienlijke vooruitgang geboekt: een nieuwe ontwikkelomgeving voor de synthese van besturingen, gebaseerd op de ontwikkelomgeving die is voorgesteld in dit proefschrift, is in ontwikkeling in de Systems Engineering Groep van de TU/e.

Contents

Chapter 1

Introduction

A current trend in embedded systems is that, due to market demands and increased competition, the numbers of functions in a system is increasing. At the same time, the time-to-market of new functions should be decreased. As a result, functions are added to (new generations of) embedded systems at an accelerating rate. For instance, around 1980 the first mobile phones could be used for making phone calls only. Around 1993, the first data features such as sending of text messages were added to the phone. Currently, mobile phones run advanced operating systems to which new features can be added every day, by means of so called 'apps'.

Adding extra functions to an embedded system increases its complexity. Whilst the quality of the systems should meet high quality constraints. To be able to release a new generation of an embedded system in time, many systems are not designed from scratch. Functions are added to previous generations of the system. The system evolves during multiple generations of the system. The investments in a system are accumulating over years, therefore, replacing it becomes more expensive every year.

As a result, developers must handle the increased complexity, and understand the system thoroughly to add the new features. To assist the development, the system should be designed such that new features and implementation technologies can be incorporated easily. In other words, the system should not be designed as a one time product; instead, the system should be designed for evolvability as well. One way to accomplish this is to use development methods that support the evolvability of a system.

In academia new methods are developed. Some of these methods could support the evolvability. However, theoretical soundness of a method does not mean the method is applicable in an industrial setting. For that, the method should scale for industrial size problems, the method should be intuitive for the engineers, and there must be industrial quality tools supporting the method, preferably with professional support. Furthermore, the method should be embedded into the development process

of the company, including its impact on service and maintenance of the embedded systems. Furthermore, engineers must be trained to use the new methods.

One of the methods that is developed in academia is Supervisory Control Theory (SCT) or Supervisory Control Synthesis (SCS). This method is used to synthesize discrete event controllers, based on models of the uncontrolled system and models of control requirements. In this thesis, it is investigated how Supervisory Control Theory can contribute to improved evolvability of MRI scanner control systems. This research is executed within the scope of the DARWIN project.

The following section introduces the DARWIN project. Section 1.2 introduces the supervisory control domain; it explains what a supervisory controller is, and how this controller relates to the other parts of the system. Section 1.3 further details the goal of this PhD project. Section 1.4 list some related work. Finally, Section 1.5 gives the outline of this thesis.

1.1 The DARWIN project

The research described in this thesis is part of the research project called DARWIN. The main objective of the DARWIN project was to investigate 'specific methods, techniques, and patterns to improve the evolvability of product families within industrial constraints and while maintaining other qualities'. The evolvability of a system is the ability of the system to respond effectively to change. Two main types of changes are considered, first, changes in the required functionality of the system, and second, changes in implementation technologies.

The DARWIN project was carried out using the Industry-as-Laboratory paradigm [55], in which the researchers closely work together with the industrial practitioners. Hence, the researchers experience not only the challenges that must be overcome to achieve the industrial goals, but also the context in which the practitioners operate. In addition, the researchers have easy access to industrial experts, their state-of-practice, and large amounts of industrial data. The DARWIN project was a collaboration between: The Embedded Systems Institute[1] (ESI), Philips Healthcare MRI, Philips Research and five dutch universities; University of Twente, Delft University of Technology, Eindhoven University of Technology, VU University Amsterdam, and University of Groningen.

These project members had different responsibilities in the DARWIN project. The Embedded Systems Institute was responsible for the management of the project, had the overall technical lead, and had to capture and consolidate the knowledge generated in the project. Philips Healthcare MRI was the carrying industrial partner, providing access to technical and business experts and to its repositories containing large amounts of industrial data and software. Furthermore, Philips Healthcare MRI provided feedback on the appropriateness of proposed methods. Philips Research and the universities were the solution providers wanting to develop and prove methods that solve industrial problems. For that, they cooperated with industry to search for

[1] Currently integrated into TNO as Embedded Systems Innovation by TNO

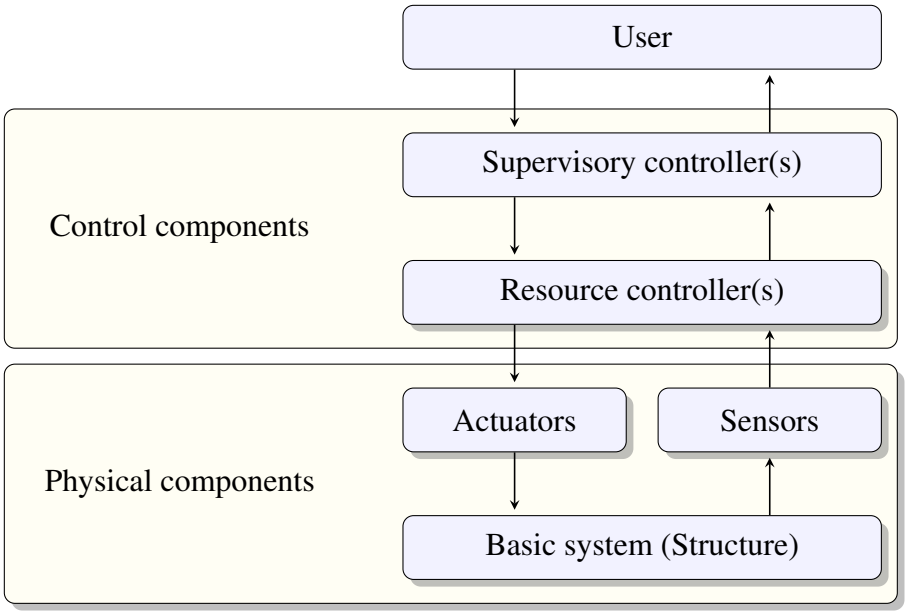


Figure 1.1: System view.

relevant problems, to investigate their root causes and their contexts, and finally, to develop, try out, and validate methods to solve them. For a complete overview of project and its results, the reader is referred to [76].

The work in this thesis has been carried out at the Eindhoven University of Technology with the Systems Engineering group. The main objective of this project was to improve the evolvability of systems by applying supervisory control theory in the development of supervisory controllers. Mainly by applying the theory with two different cases within the MRI patient environment.

1.2 Supervisory control

Complex systems do not operate without some method of control. A (controlled) system can be divided into two parts, namely physical components (hardware) and control components, see Figure 1.1. The physical components provide the means of the system, that is what the system can do. The control components employ those means to fulfill the system's functions, that is what the system should do. The users of the system interacts with the control components to define which tasks must be performed by the system.

The physical components typically consist of a basic system (i.e., structure), sensors and actuators [33]. The basic system provides the basic structure of the system. The structure can be, among others, mechanical, electrical, thermal and/or hydraulic. The sensors measure a physical quantity of the basic system. For instance, position, voltage, temperature or pressure. The measured quantity is converted to a signal that is sent to the control components. The control components send control signals to the actuators. The actuators act directly on the basic structure,

having effects on the physical quantities of the basic system. The interactions between the physical components result in the so-called *uncontrolled behavior* of the machine. The combination of all physical components is the system under control, the uncontrolled system or *plant*.

The control components interact with the physical components by reading the signals from the sensors, and by sending signals to the actuators. Based on the information read from the sensors and the user inputs, instructions are sent to the actuators to fulfill the desired function of the system. This results in the *controlled behavior* of the system. The combination of all physical components together with all control components is the *controlled system*. The behavior of the controlled system should be such that the system fulfills its functions, i.e., the system meets its predefined requirements.

The control system can be divided into multiple levels [6, 8, 33]:

1. *Resource control* (also know as regulative or low-level control) assures that a system reaches the desired position or the desired state in the desired way. In other words, it provides predefined functions or resources that the higher levels of control can use. Generally, regulative control acts on small parts of the system with much detailed information about this part of the system. Regulative control is typically the domain of the classical control theory, in which continuous time driven systems are considered. PID controllers are commonly used in this domain.

2. *Supervisory control* (also known as logic control) assures that a system correctly performs its function by determining and executing allowed sequence of tasks on (in)dependent resources. In other words, it uses the resources provided by the resource controller to execute defined tasks in the correct order. Supervisory control includes coordination of the individual components of the system, and the sequencing and scheduling of tasks. Furthermore, it must prevent the system from entering dangerous states, by that, it ensures that the system stays safe. Finally, supervisory control can optimize the system, for instance for optimal speed or minimal energy consumption. Generally, supervisory control acts on the global system with abstract information about the different parts in the system. Furthermore, supervisory control is in general less time critical then resource control.

3. *User interfacing* is provided to interact with the users of the system. A user of the system can be a human operator, but also another system. The user interface provides information about the current state of the system. Furthermore, it allows the user to request the execution or abortion of tasks. In this sense, the user of the system can be regarded as yet another (more abstract) controller of the system. This is especially true if the user of the system is another system.

At each level of control, the system is regarded at a different abstraction level. Higher levels of control regard the system on a more abstraction level then the lower levels do. In the higher levels many details about the lower level control are hidden. In

general, the number of components on a higher level are less than the number of components in the lower levels. For each level of control different methods are used to develop the controllers. Typically, the resource controller is in a continuous and time driven domain. The supervisory controller is in a discrete and event driven domain.

This thesis focuses on the design of the supervisory control level. At this control level, many systems can be regarded as Discrete Event Systems (DES) [13]. A discrete event system is a system that has discrete states, and the system switches between these states instantly by means of events. For instance, a motor can be in the states 'moving' or 'stopped', the event 'stop' indicates that the motor switches from the state 'moving' to 'stopped'. A sensor can be in the states 'on' or 'off', the event 'off' indicates that the sensor switches from the state 'on' to the state 'off'. Discrete event systems can be modeled by means of automata, i.e., finite state machines.

1.3 Goal of the project

Although Supervisory Control Theory was initiated by Ramadge and Wonham long ago [56, 80, 57], and many scientific advances have been made over the years (see Section 7.1 for a short overview), there are still no signs of significant industrial adoption of supervisory control synthesis in industry. The aim of this thesis is first, to show the potential of supervisory control synthesis in industry, second, to clarify some of the reasons for the lacking industrial adoption, and third, to provide solutions for improved industrial adoption of supervisory control synthesis. To achieve this, several flavours of supervisory control synthesis are applied to representative cases in industrial supervisory controller design in High Tech Healthcare systems, where safety is paramount. The complete supervisory controller design chain is taken into account, from analysis, modeling and synthesis, to validation and real-time implementation. Results are analysed, shortcomings of current methods, tools and design techniques are identified, and improvements are proposed and partially implemented.

1.4 Related work on SCS applications

Without claiming completeness, previous applications of SCS include: a rapid thermal multiprocessor, see [5], mobile robots, see [41, 35], passenger land-transport systems, see [60], a water bath boiler, see [49], under-load tap-changing transformers, see [52], a cluster tool for wafer processing [64], automated manufacturing and assembly systems: [9, 40, 10, 38, 26, 34, 17, 14, 53, 16, 58, 42, 54, 25, 61].

To the best of our knowledge, the application domain of MRI scanners is new. In addition, none of the above cited papers consider the problem of evolvability. In contrast, improving evolvability is one of the main tasks in the application considered in this chapter.

1.5 Outline

This thesis is organized as follows. Chapter 2 describes different engineering methods. Chapter 3 presents the supervisory control theory. Chapter 4 gives an overview of the MRI scanner, and the patient environment. Chapters 5 to 7 describe applications of supervisory control theory to a patient support system and a patient communication system. Finally, Chapter 8 presents some concluding remarks.

Chapter 2

Engineering of supervisors

Overview The aim of this chapter is to give on overview of the development process for control system design in industry, and to indicate how supervisory control synthesis can be incorporated into an industrial controller design process.

Contribution A contribution of the chapter is, that it clarifies how supervisory control synthesis cannot be introduced in existing controller design processes in isolation. During the four year DARWIN research project at Philips, described in this thesis, feedback from industrial users and developers led to the insight that introduction of supervisory control synthesis in industry should be embedded in an integrated model-based development framework [11], which is referred to as 'synthesis-based engineering'. Such an integrated, synthesis-based engineering process has been shown to be feasible, by executing all steps, from control requirement and plant model design, via synthesis, simulation-based validation and visualization, to real-time implementation and testing. As a side effect, this research project has also accelerated the development of the integrated CIF toolset. At the beginning of the research project, only a number of isolated tools were available. During the project, major steps have been taken by the CIF development team towards an integrated CIF tool set for synthesis-based engineering . This took place in close interaction with this research project, and has led, via many iterations, to various improvements in the toolchain, and in the models for the three applications discussed in the three Chapters 5 - 7.

Outline Section 2.1 explains the traditional engineering process, focusing on the design of supervisory controllers and the challenges that are faced in evolving systems. In Section 2.2 the model-based engineering process [11] is described. The model-based engineering process extends the traditional engineering process by adding (formal) models. This allows for many model-based analysis techniques to be applied. Finally, Section 2.3 introduces the synthesis-based engineering process. In

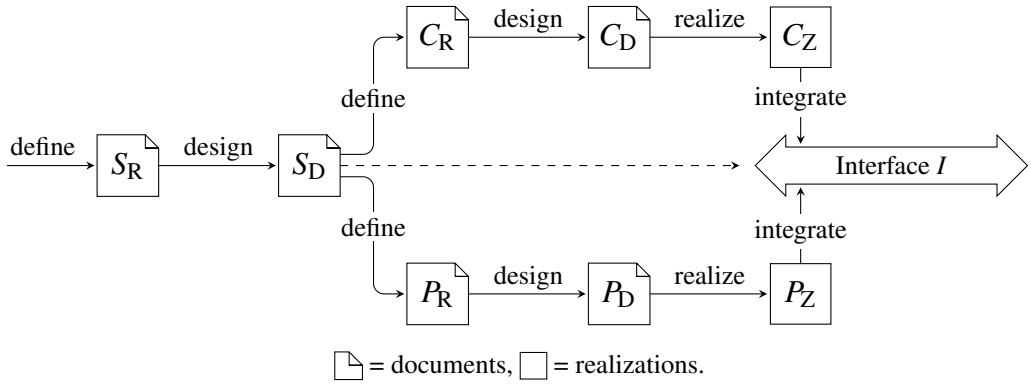


Figure 2.1: Traditional engineering process [11]

this engineering process, the supervisory controller can be synthesized (generated), which results in reduced development effort.

2.1 Traditional engineering

The traditional engineering process is in use by developers for a long time to develop, among others, supervisory controllers. This section describes this engineering process, with a focus on supervisory control. Figure 2.1 shows a graphical representation of the traditional engineering process. The arrows depict the different engineering phases, where the boxes depict the different representations of the (sub)systems. The wide arrow represents the interface I by which subsystems are integrated. The capital S refers the system, and the capitals P and C refer to the subsystems plant and supervisory controller, respectively. The subscripts R, D and Z refer to the representations requirements, designs and implementations, respectively. The following paragraphs will explain the process in some more detail.

In the initial phase, the requirements of the system are defined. System requirements S_R define the functionality that a system should provide. Typically, the requirements also include constraints on for instance costs, performance and/or safety that should be satisfied by the system as well.

In the next phase, a design of the system is made. System design S_D specifies how the system should be built to satisfy the requirements. The design specifies, for example, the architecture, the decomposition of the system, the internal behavior, and the technologies used. Furthermore, it specifies the interaction and interfaces between the components of the system. For supervisory control design, the system is decomposed into the components plant P and supervisory controller C. These two components are connected via interface I which is also defined in the design. Note that the plant consists of all physical components together with the resource controllers.

The decomposition in the design defines the requirements for the components P_R and C_R. The higher level requirements and design decisions are translated into the corresponding requirements for the plant and supervisor. The corresponding

design of the plant P_D and design of the controller C_D specify how the components satisfy their requirements. The components are realized according to the design, resulting in realizations of the components P_Z and C_Z. Realizations are the real components, e.g., mechanics, electronics or software, of the system. They should satisfy the requirements as defined for each component. Traditionally, supervisory controllers are developed by coding them manually, based on the defined control requirements. While the uncontrolled system is developed independently by another group of engineers.

After the realizations of the plant and the supervisor are completed, the realizations are first tested separately to ensure that the subsystems meet their requirements and design. Thereafter, the two subsystems are integrated by means of interface I to build the complete system. The interface can consist of, for example, signal cables, in electronics, or communication networks, in software. The interface connects the individual components, thereby it establishes the component interaction, according to the design S_D. When the subsystems are integrated, the integrated system can be tested to ensure that it conforms to intended system design S_D and satisfies the systems requirements S_R. Finally, in the acceptance test it is validated whether or not the engineered system meets customer demands.

So far, the engineering process is described as a 'sequential' process. There are no feedback loops from certain phases to earlier phases, and a phase only starts when the previous phase has been finished. The real system engineering process has a more incremental and iterative nature, involving multiple versions of the requirements, designs, and realizations, and having feedback loops between the phases. These feedback loops occur for instance when design and/or implementation errors must be fixed, or when the system requirements change as a result of evolution. In the latter case, there is a feedback loop from the realization phase back to the system definition phase.

Furthermore, in practice, the different phases are executed in parallel. For instance the design phase stared before all requirements are defined and completely understood. This is often necessary due to the time-to-market pressure to deliver a system in time. Moreover, often the requirements only become clear after a (prototype) system has been built.

2.1.1 Verification and validation

An important part of the engineering process is checking quality of the system. It must be justified to the customer that the system includes the expected functions and that these functions are performed correctly. To this end, verification and validation activities are performed during the development of the system. Verification is the process of checking whether a component or the system corresponds to the design X_D and requirements X_R, i.e., the component or system is built right. Validation is the process of checking whether a component or system fulfills its intended purpose, i.e., the right component or system is built. Multiple activities can be preformed to check the quality of the system under development in different phases.

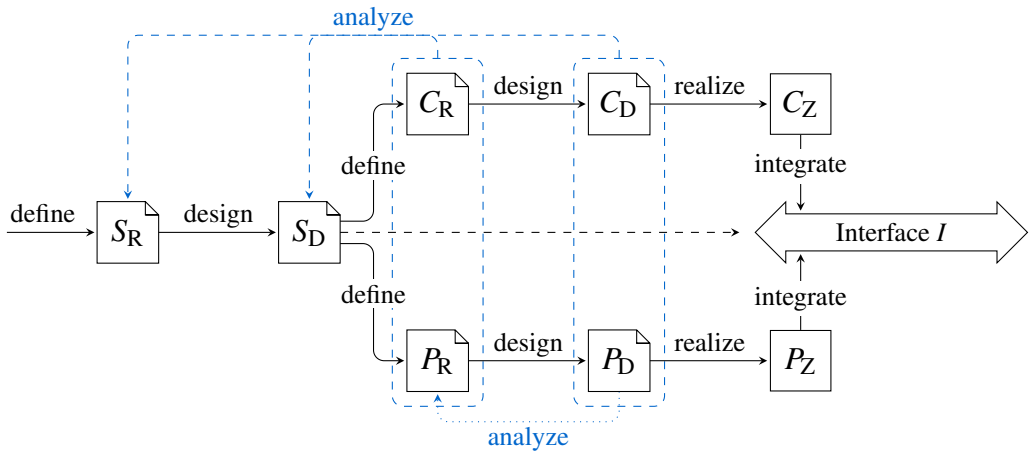


Figure 2.2: Requirement and design analysis [11]

Requirement and design analysis

Requirement and design analysis is applied to check the consistency between requirements and designs on the component level and those on the system level. It is analyzed whether the requirements and designs of the components together implement the system design and requirements. To analyze these relations in the traditional engineering process, engineers should create a 'mental' model of the integrated components. The behavior of this mental model is checked against the intended behavior of the system for consistency. This requires significant knowledge of the components as well as clear system overview. This activity is depicted by the dashed arrows in Figure 2.2, in which the representations not involved in this activity are grayed out.

An example of requirement analysis is *requirement traceability* [30]. Requirement traceability is about understanding how system level requirements are related to component level requirements, and vice versa. It should be possible to trace back the origin of each requirement and every change which was made to this requirement. This is especially important in evolving systems, in which the main effort of the engineering process is to adjust the system and components to satisfy new or changed requirements.

Component and system testing

A common way of assessing the quality of components or systems is testing. By testing the component or system failures can be detected, so that errors in the requirements, the design and the implementation of the component or system can be uncovered and corrected. If no failures are found, the confidence that the component or system is correct increases. Although, testing cannot guarantee the absence of errors. The component and system test activities are depicted in Figure 2.3. Note that testing both verification and validation of the system, for instance, component testing is often a instance of verification, where acceptance testing is often a form of

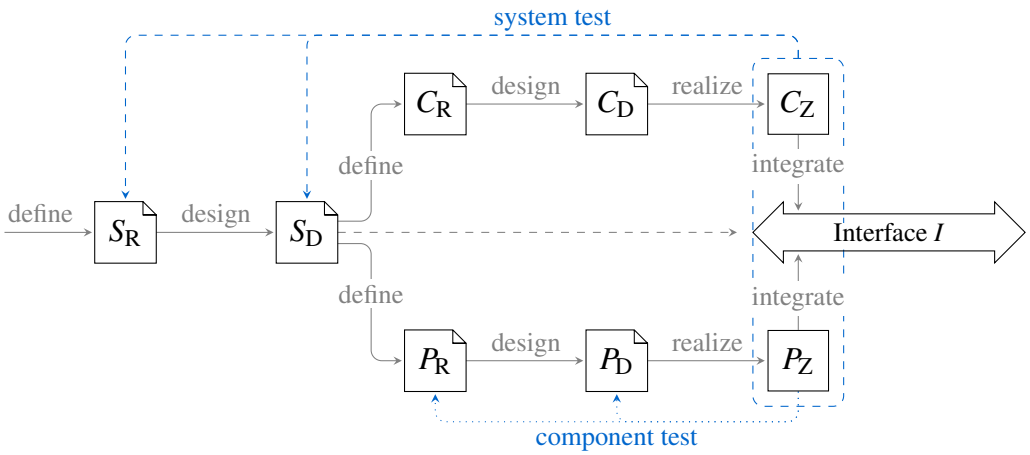


Figure 2.3: Component and system testing [11]

validation.

In order to test something, a test method needs to be defined. This method defines what aspects of a component or system is tested, how this aspect is tested and against which reference this aspect is tested. The outcome of a test (pass or fail) is obtained by executing the test method on the realization. Mostly, the aspects defined in the requirements and design are the base for test definitions. Moreover, the requirements and design are used as the reference in the test method. Defining test methods is a separate activity within the engineering process.

2.1.2 Discussion

In the traditional engineering process, the requirements and designs are mostly made in (informal) documents only, i.e., the traditional engineering process is mostly based on documents. This has several disadvantages. First, documents may contain ambiguous or inconsistent information. Moreover, practical experience shows that documents may be incomplete, or outdated. As a result, it is difficult to obtain a good system overview, and to detect inconsistencies and potential problems based on these documents. Second, due to the informal structure of documents, it is hard to process them automatically. This complicates automated processing analysis techniques such as inconsistency detection. This currently leaves manual document reviewing as the main technique used for document analysis. Third, documentation is a static piece of information, which makes it difficult to express and analyze the dynamic system behavior. Fourth, determining the integrated system behavior based on component documentation only, is a difficult task. It requires a considerable amount of component design knowledge, as well as good system overview.

In other words, documentation is not well suited to check the correctness of the system to be built. In traditionally development, this leaves the realizations of the (sub)systems as the only representations that can be tested, verified and validated. As a result, the correctness of the (sub)system can only be validated late in the engineering process, after the (sub)system has been realized.

Furthermore, often the final requirements become clear after a (prototype) implementation has been created. Validation of this implementation gives the insights into the behavior of the system needed to define the final requirements. An implementation can only be realized after a relative detailed design of the system is completed. As a result, iterations in the traditional engineering process take much time and are resource intensive.

These disadvantages are especially true for evolving systems, in which requirements change frequently. Making new (or updated) designs and implementations is time consuming, cost intensive, and short-time goals may cause degradation of the overall quality of the system over time.

2.2 Model-based engineering

To overcome some drawbacks of the traditional document-based engineering process, models representing the (sub)system can be included into the engineering process. This results in the model-based engineering process [11]. Models are an abstract representation of a real (sub)system, designed to show the main object or workings of the (sub)system. Making models enables the use of a range of model-based techniques and tools to support the engineering process. Models can, for instance, be used in experiments to gain knowledge about the real (sub)system. Different types of models can be used, such as scale models of cars to analyze aerodynamics. However, this thesis particularly focuses on models describing the discrete-event behavior of (sub)systems.

The main difference between models and documents is that the constructs used in models have semantics which define precisely what each construct means; documents do not have such semantics. This makes models more consistent and less ambiguous than documents. Furthermore, the well-defined model semantics enables automatic processing by tools. This enables the use of various sophisticated and automated techniques.

Figure 2.4 shows a graphical representation of the model-based engineering process. The initial definition of the requirements S_{R} and system designs S_{D} remains the same as in the traditional engineering process. After the design of subsystems P_{D} or C_{D} is completed, the key properties of the designs are modeled, resulting the models P_{M} and C_{M}, respectively. If multiple (independent) properties are of interest, multiple models can be used, modeling each a different aspect. The original designs P_{D} and C_{D} extended with the models P_{M} and C_{M} are the input for the implementation phases.

In case of supervisory controllers, the model of the supervisory controller can be used directly to generate the implementation of the supervisor. In this case, the model is not used as input to code the supervisor manually. Instead, the model is transformed into the implementation code by code-generators, comparable to how compilers produce machine code from source code.

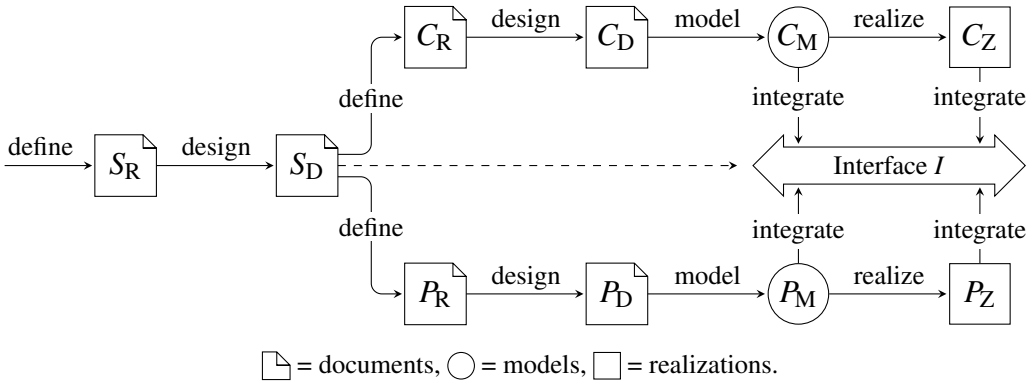


Figure 2.4: Model-based engineering process [11]

2.2.1 Verification and validation

Formal models can be processed by tools. This enables the use of many (automated) verification and validation methods. These methods extend or replace the methods used in the traditional engineering process.

System analysis

The models used in the MBE approach model the *actual* behavior of the components. That is, they model the behavior of the component as it is designed and implemented in the traditional engineering process. The integration of these models yield a system model that is an early representation of the actual system behavior. Therefore, this system behavior includes the design errors that are made in the traditional process. As a result, the emerging behavior of the system may be incorrect, for instance, due to an unforeseen conflict between component designs. By analyzing the integrated model, these design errors can be discovered before the actual implementation of the component is ready. Furthermore, the confidence that the design is correct increases if no errors are found.

To discover the errors, the behavior of the models must be checked whether it corresponds to intended design S_D and it satisfies system requirements S_R. There are multiple techniques to analyze the modeled behavior of the system. A widely used technique is simulation. In simulation one possible model behavior is determined. In cases where the behavior is stochastic or nondeterministic, multiple simulation runs can be used to explore multiple behaviors.

In most cases, simulation is not exhaustive, it does not explore every possible behavior of the system. This means that simulation can only show that the model *might* have correct behavior. It cannot guarantee correctness of the model. If guaranteed correctness of a model is important, formal verification techniques can be used to prove the correctness of the model. In formal verification all possible behaviors of a model are checked against specified properties. These model-based system analysis techniques are shown in Figure 2.5. Both simulation and verification can be performed on component as well as system level. If components are tested in

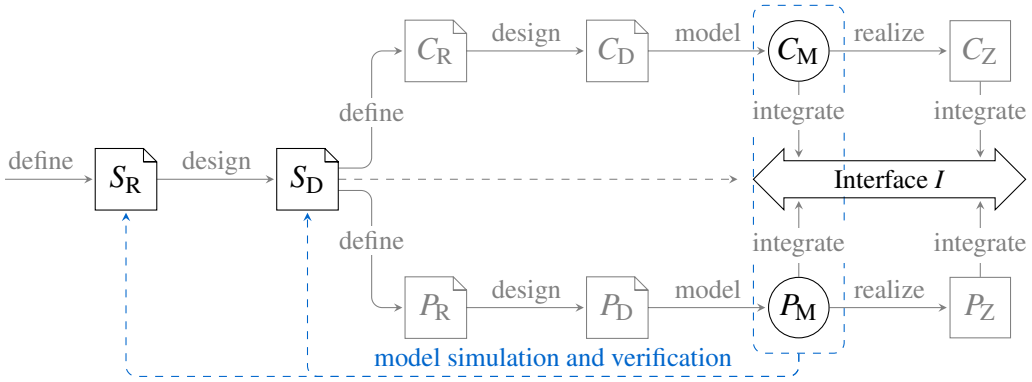


Figure 2.5: Model-based system analysis [11]

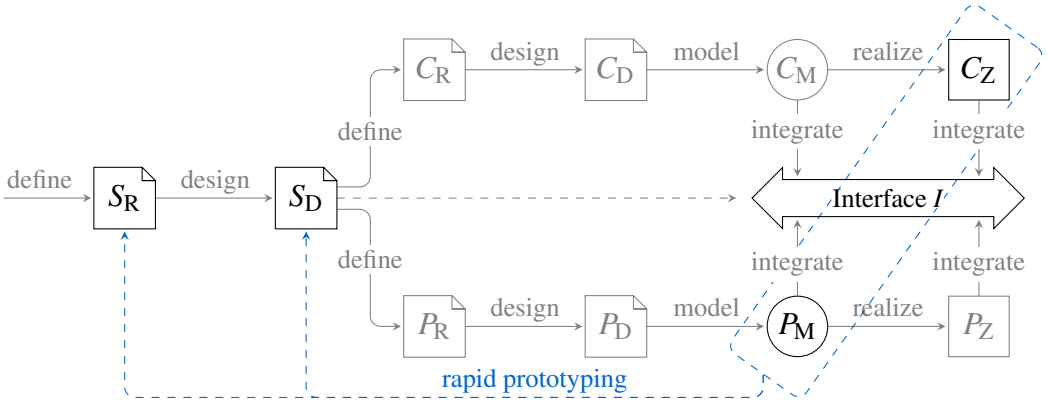


Figure 2.6: Model-based system testing [11]

isolation, the results cannot guarantee system level correctness.

Integration and system testing

Often, not all component realizations are completed at the same time. In traditional engineering, the system can only be tested when all component realizations are completed. However, in model-based engineering the not yet completed realizations can be substituted by the models that represent the respective component. As a result, some component realizations can be integrated with models of the other components. By integrating models with the available component realizations, a *model-based integrated system* is obtained. In this way, the completed components can be tested earlier on a system level. Furthermore, by using models, some parts of the system can be tested for extreme conditions or under insertion of faults, without running into real dangerous situations.

In the supervisory control case, see Figure 2.6, if a model of the supervisor C_M is integrated with the real uncontrolled system, it is called *controller prototyping*. If the supervisor realization C_Z is integrated with a model of the uncontrolled system P_M, it is called *hardware-in-the-loop simulation*.

2.2.2 Discussion

The use of models enables the use of model based techniques to evaluate the system. The use of models helps to find errors in the designs and requirements earlier in the process. This increases overall confidence in the correctness of the system. Furthermore, it decreases the development time and effort. Moreover, making models enforces that developers examine the requirements and designs earlier in more detail. This helps in making requirements more clear and complete early in the engineering process. Model-based engineering can be used to check the correctness of a system based on models. This analysis can be used as a supplement to the traditional engineering process. Therefore the impact of using model-based analysis on the traditional way of working is limited.

The model-based engineering process improves the traditional engineering process. However, model-based engineering helps in discovering errors only. It does not prevent from introducing errors while engineering the system. Although, errors can be discovered early on in the engineering process, they still must be corrected by hand. Furthermore, the components have multiple representations. The relations between these representation is unclear, while they are mostly made by hand. As a result, the correctness of a model X_M does not guarantee the correctness of the implementation X_Z. The correctness of this relation can only be determined by checking the implementation against the model. If the implementation is not checked against the model, only the design of the implementation is improved.

In addition, in evolving systems the requirements change over time. Each requirement change must result in a change in the implementation. Model-based engineering can reduce the time to market of such a change. However, each changed system requirement results in changed requirements and designs for the components. These changes must be traversed thought the different engineering phases by hand. Assessing the impact and making the changes can be complex and error prone.

2.3 Synthesis-based engineering

The model-based engineering process adds models to the traditional engineering process. These models can be used to verify that the system has certain properties. The goal of synthesis-based engineering is to synthesize (automatically derive) models such that it is guaranteed that the system has certain properties. As a result, both the design and the verification of the synthesized component are eliminated. Since the design rules and the properties of the component are included in the algorithms that synthesize the component model.

Figure 2.7 shows a graphical representation of the synthesis-based engineering process. In case of supervisory controller synthesis, the uncontrolled system is develop by using the same process as model-based engineering. For the supervisor the engineering process changes, the manually created design of the supervisor C_D is eliminated. It is replaced by model of the supervisor requirements C_R. Together with model of the uncontrolled system P_M, a model of the supervisor design C_M is

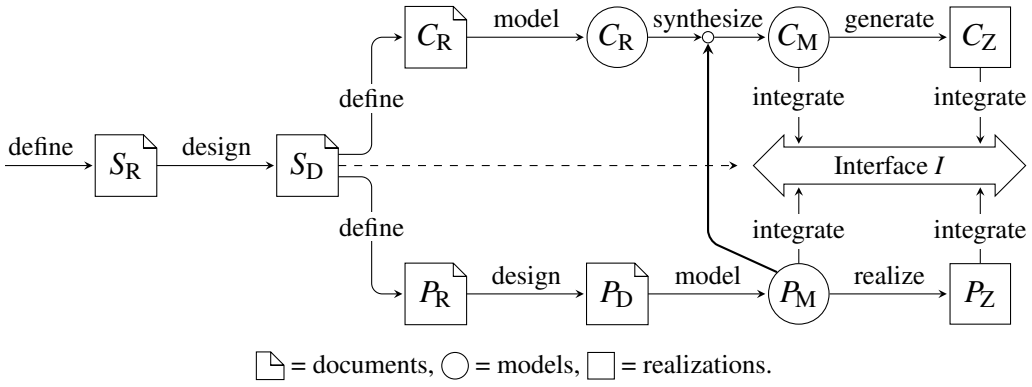


Figure 2.7: Synthesis-based engineering process

synthesized. The synthesized model C_M can be used to automatically generate the implementation C_Z.

2.3.1 Verification and validation

Although synthesis-based engineering changes the engineering process, all model-based verification and validation techniques remain applicable. For the synthesized components, testing and verification of the component implementation can be eliminated. The components are correct by construction, given that the models used as input are correct. For the components that are not synthesized and the system, validation and verification do not change with respect to the model-based engineering process.

2.3.2 Discussion

For supervisory control, the correct supervisor is generated from models of the uncontrolled system and the control requirements. This eliminates the manual design of the supervisor and verification of the supervisor implementation. As a result, the engineering process changes from implementing and debugging the design and the implementation, to designing and debugging the requirements. In other words, the system verification can be eliminated, only the system needs to be validated. This allows for a faster incorporation of requirement modifications into the implementation. In turn, this leads to a reduction in the number of design-test-redesign loops.

The advantages of model-based engineering remain applicable. The models of the uncontrolled system and of the control requirements are unambiguous. Leaving no room for different interpretations. Furthermore, making the models enforces that the requirements are scrutinized earlier in the development process. As a result possible inconsistencies and missing parts are discovered earlier. Simulation of the system allows for dynamical feedback of the system. Together with visualizations, this is a tool to validate the system behavior with the customer in an early stage. This can help to get the set of requirements consistent and meeting the customers

expectations, early in the development process, before prototypes have been built. Finally, the synthesized supervisors are suitable for code generation. Therefore it is relatively independent of the implementation technology.

With respect to evolvability, the engineering loops are shorter, allowing faster implementation of new changes. Changes in the plant or in the control requirements can be realized quickly, without introducing errors. Furthermore, the tracking of changes in the requirements is easier, only the references to the requirement models need to be followed.

Chapter 3

Supervisory control synthesis (SCS) theory

This chapter presents a short overview of the original theory for supervisory control of discrete event systems. Supervisory Control Theory (SCT) was initiated by Ramadge and Wonham in 1989 [80]. Since then, the theory has been extended and algorithms have been improved in numerous ways. For an overview of supervisory control theory and some of its extensions, the reader is referred to [13, 79].

The control of discrete event systems is different from the classical control of continuous time systems. The differences in the nature of the system under control are such that other mathematical frameworks are used to describe the behavior of the system. For instance, for the modeling of the behavior of a discrete event system, explicit modeling of time is irrelevant. Only the order in which the events take place are of interest, not the particular point in time when the events occur. While in continuous dynamics, the progress of the values of the variables as a function of time is a key factor. In the following paragraphs, the basic properties of discrete event systems and controllers for discrete events systems are given.

For the mathematical framework to describe event systems two basic concepts are of interest, namely *discrete states* and *events*. The behavior of the system is described by the sequence of states and/or events that the system visits.

For control, the controlled system consists of two components, the plant (system under control) and the supervisor (controller). Both the plant and the controller are discrete event systems that communicate with each other via discrete signals. The plant informs the supervisor about the current state and/or the events that occurred. Based on the received information, the supervisor decides which events are allowed to happen next in the plant[1]. Figure 3.1 shows the basic control loop of a discrete event system.

[1]Note that from a theoretical perspective, it is irrelevant whether the plant or the supervisor decides which event will happen next. However, from an implementation point of view, it is relevant whether the plant or the supervisor decides which event happens next.

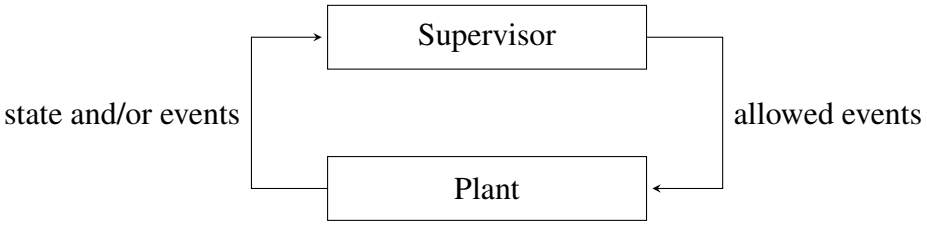


Figure 3.1: Basic control loop for discrete-event systems.

3.1 Control problem

The general control problem for discrete event systems is to find a controller (supervisor) that influences the behavior of the plant in such a way that it meets the control objectives. The control objectives of interest are safety, liveness and minimal restrictiveness.

- *Safety*: guarantee that the controlled system is *not* able to perform forbidden behavior.
- *Liveness*: guarantee that the controlled system is able to perform a minimum level of functionality.
- *Minimal restrictiveness*: the supervisor should not restrict the plant more than necessary. This leaves the behavior as large as possible after control is applied.

The control problem is complicated by the fact that the supervisor cannot prevent arbitrary events from occurring. For instance, a supervisor cannot prevent a sensor from switching. This leads to the notion of controllable and uncontrollable events:

- *Uncontrollable events*. These events cannot be prevented from occurring by the supervisor. These are typically events generated by sensors, for instance 'sensor turned on'.
- *Controllable events*. These events can be prevented from occurring, or can be disabled, by the supervisor. These are typically events related to actuators, for instance 'release clutch'.

3.2 Uncontrolled systems

The original theory of supervisory control synthesis is based on the mathematical framework of languages. In this framework, the behavior of the uncontrolled system (i.e., plant) and the specification for the controlled system (i.e., control requirements) are both given as languages. A language is a set of strings, i.e., a set of sequences of events.

The behavior of a plant P is modeled by a pair of languages (L, L_m). The *language* L consists of the set of all possible strings that the uncontrolled system can generate, and the *marked language* consists of all possible strings that the uncontrolled system can generate whose execution implies the completion of a certain task, hence $L_m \subseteq L$.

Formally, let Σ denote set of events that can occur in plant P, called the *alphabet* of P. A sequence of events, for example $w = \sigma_1\sigma_2\ldots\sigma_n$ where $\sigma_i \in \Sigma$, is called a string over the alphabet Σ. Let Σ^* denote the set of all possible finite strings that can be constructed from the elements of the set Σ, including the empty string ε. The set of all admissible, (i.e., physically possible) strings that the plant can generate is the language $L \subseteq \Sigma^*$. The *prefix closure* of L, denoted by $\overline{L}$, consists of all prefixes of the strings in language L. Intuitively, for a sequence of events to occur in a DES, all of its prefixes must occur first, hence $L = \overline{L}$.

3.3 Controlled systems

Before specifying the behavior of a controller and the behavior of the controlled system, the set of all possible control patterns is introduced. A control pattern is a set of events that are enabled. Clearly, all uncontrollable events must always be enabled. Partition the alphabet into two disjoint subsets $\Sigma = \Sigma_u \cup \Sigma_c$, where Σ_u is the set of *uncontrollable* events, and Σ_c is the set of *controllable* events. Then, the set of all possible *control patterns* is defined as:

$$\Gamma = \{\gamma \in 2^{\Sigma} \mid \gamma \supseteq \Sigma_u\} \tag{3.1}$$

An event $\sigma \in \Sigma$ is enabled (permitted to occur) if $\sigma \in \gamma$, otherwise it is disabled (prohibited from occurring).

A controller (supervisor) restricts the behavior of the plant by dynamically disabling some of the events that the plant can generate. A supervisor is defined as a map:

$$f_c : L \to \Gamma \tag{3.2}$$

specifying for each possible generated string w the control input $f_c(w)$ to be applied at that point. When plant P is controlled by a supervisor C, it operates as before, except that it obeys the additional constraint that, following the generation of a string w, the next event must also be an element of $f_c(w)$, formally let $w \in L$ then $\sigma \in f_c(w)$ and $w\sigma \in L$. Observe that such control is 'permissive': while disabled events are certainly prevented from occurring, enabled events are not necessarily forced to occur.

The closed-looped language of the plant P controlled by supervisor C is the smallest language L^c such that:

1. $\varepsilon \in L^c$
2. $w\sigma \in L^c$ if $w \in L^c, \sigma \in f_c(w),$ and $w\sigma \in L$

The part of the original marked language that survives under supervision is:

$$L_m^c = L^c \cap L_m \tag{3.3}$$

If L_m represents completed tasks, then this language indicates those tasks that can be completed under supervision.

3.4 Existence of supervisors

The goal of the supervisor is to switch the control pattern such that the *closed-loop* behavior of the supervised plant meets the control requirements. Clearly, it is not possible to meet arbitrary control requirements. This results in the following problem: Given a plant P with behavior L, what closed-loop behaviors $K \subseteq L$ can be achieved by supervision. This leads to the two main properties in supervisory control theory, namely *controllability* and *nonblocking*.

A supervisor cannot disable uncontrollable events; therefore it is intuitively clear that a specification K can describe only feasible closed-loop behavior, if it allows uncontrollable events to occur whenever such event is enabled in the plant. This is described by the controllability property.

Definition 1 (Controllability) Let $K \subseteq L$ be a specification language, where $K \neq \emptyset$, and $\Sigma_u \subseteq \Sigma$. K is said to be *controllable* with respect to L and Σ_u if:

$$\overline{K}\Sigma_u \cap L \subseteq \overline{K}$$ □

That is, if any prefix of a string in K, i.e., $w \in \overline{K}$, is followed by an uncontrolled event $\sigma \in \Sigma_u$ in L, i.e., $w\sigma \in L$, then it must also be a prefix of a string in K, i.e., $w\sigma \in \overline{K}$.

A controlled system should at least be able to complete one of its tasks, $L_m^c \neq \emptyset$. The marked language expresses these completed tasks. Therefore, the system must always be able to complete a string to a marked string. This liveness requirement is expressed by the nonblocking property.

Definition 2 (Nonblocking) A DES is *nonblocking* if:

$$L^c = \overline{L_m^c}$$

Otherwise the system is *blocking*. □

That is, every prefix of a string in L_m^c is in the language L^c, therefore, every string in L^c can always be extended to a marked string. Note that from definition 3.3 it is clear that $\overline{L_m^c} \subseteq L^c$, however, the reverse containment may not hold in general.

The existence of a correct supervisor for a given controllable non-blocking specification is given by the controllability theorem:

Theorem 1 (Nonblocking Controllability Theorem) *Let $K \subseteq L_m$, where $K \neq \emptyset$. There exists a nonblocking supervisory controller f_c for L such that:*

$$L_m^c = K \text{ and } L^c = \overline{K}$$

if and only if the following two conditions hold:

1. *K is controllable with respect to L and Σ_u*
2. *K is L_m-closed: $K = \overline{K} \cap L_m$* □

If a supervisory controller exists, it is given by:

$$f_c(w) = \Sigma_u \cup \{\sigma \in \Sigma_c \mid w\sigma \in \overline{K}\} \tag{3.4}$$

3.5 Controller synthesis

The admissible behavior of a system L_a defines the allowed behavior of the system, $L_a \subseteq L$. In general, the admissible language L_a does not meet the properties defined in the previous section. Therefore, in general, there exists no supervisor to accomplish this behavior. The goal of supervisor synthesis is to find a specification $K \subseteq L_a$, such that it meets specified properties, and is 'the largest it can be'. This specification K can be implemented as a supervisor.

Clearly, not every specification is nonblocking and controllable. The basic nonblocking supervisory control problem to construct a correct supervisor that ensures that the controlled system behaves *within* the specification, and is *maximally permissive*. A maximally permissive supervisor allows the 'largest possible' sublanguage of the specification that is nonblocking and controllable.

The largest possible controllable sublanguage of a given language E is called the *supremal controllable sublanguage*, denoted by $E^{\uparrow C}$. This language can be constructed as follows. The class of all controllable sublanguages of a language $E \subseteq \Sigma^*$ is defined as:

$$C(E) := \{K \subseteq E \mid \overline{K}\Sigma_u \cap L \subseteq \overline{K}\} \tag{3.5}$$

This class is not empty since $\emptyset \in C(E)$. Furthermore, this class is closed under arbitrary language union; therefore, $E^{\uparrow C}$ does exist. The supremal element is derived by taking the union of all the elements of the class $C(E)$.

$$E^{\uparrow C} = \bigcup_{M \in C(E)} M$$

Controllability is preserved under arbitrary unions, consequentially:

$$E^{\uparrow C} \in C(E)$$

When the specification is $E_m \subseteq L_m$, that is L_m-closed, then the maximally permissive nonblocking controllable sublanguage is given by:

$$L^c = \overline{E_m^{\uparrow C}}, \text{ and } L_m^c = E_m^{\uparrow C}$$

3.6 Representation

In the event-based SCS approach, both the language of the plant and the language of the specification are represented by automata. An automaton or state-machine is a 5-tuple:

$$G = (Q, \Sigma, \delta, Q_0, Q_m)$$

where:

- Q is the set of discrete *locations q*
- Σ a finite set of *event labels* called the *alphabet*
- $\delta : Q \times \Sigma \to Q$ is the (partial) *transition function*
- $Q_0 \subseteq Q$ is the set of *initial locations*

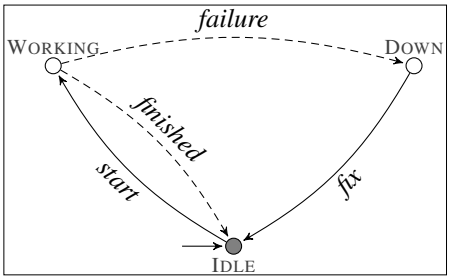


Figure 3.2: Automaton of a simple machine.

- $Q_m \subseteq Q$ is the set of *marked locations*

An automaton starts in one of its initial locations $q_0 \in Q_0$ and changes its locations and generates a sequence of events, according to its transition function δ. The occurrence of an event is signaled by the corresponding event label $\sigma \in \Sigma$. A sequence of events, called a string, belongs to the generated language of an automate if it is executable from the initial location; it belongs to the marked language if the location reached upon execution is a marked location $q_n \in Q_m$. A marked string indicates the completion of a certain task. The set of all possible strings that an automaton G can generate from the initial locations Q_0 is the language $L(G)$. To define the language generated by G, extend the transition function δ of G to a (partial) function on $\Sigma^* \times Q$ by defining:

$$\delta(q,\varepsilon) = q$$
$$\delta(q,s\sigma) = \delta(\delta(q,s),\sigma)$$

The language generated by automaton G is:

$$L(G) = \{w : w \in \Sigma^*, q_0 \in Q_0 \text{ and } \delta(q_0,w) \text{ is defined}\}$$

Similarly, the marked language $L_m(G)$ is the subset of $L(G)$ consisting of all strings that can be generated starting from a location in Q_0 that end in a location in Q_m. It is defined as:

$$L_m(G) = \{w : w \in L(G), q_0 \in Q_0 \text{ and } \delta(q_0,w) \in Q_m\}$$

An automaton is graphically depicted by vertices and edges. The vertices represent the locations, and the edges represent the transitions. Labels at each edge indicate the event that is generated if the transition is executed. The initial location is indicated by an unconnected incoming arrow. Marked locations are indicated by filled vertices. Controllable and uncontrollable events are drawn with solid and dashed edges, respectively.

Figure 3.2 shows an example of an automaton model of a simple machine. In this example, $Q = \{\texttt{Idle},\texttt{Working},\texttt{Down}\}$, $\Sigma = \{\textit{start},\textit{finished},\textit{failure},\textit{fix}\}$, $\delta(\textit{start},\texttt{Idle}) = \texttt{Working}$, $\delta(\textit{finished},\texttt{Working}) = \texttt{Idle}$, $\delta(\textit{failure},\texttt{Working}) = \texttt{Down}$, $\delta(\textit{fix},\texttt{Down}) = \texttt{Idle}$, $Q_0 = \{\texttt{Idle}\}$, and $Q_m = \{\texttt{Idle}\}$.

3.7 Composition of automata

Automata can be modeled in a modular way, i.e. they can be represented as parallel composition of several automata. In order to compose automata, two kinds of parallel composition are used:

- *synchronizing parallel composition*, see [79], which requires synchronous execution of shared events (events with common labels) and interleaved (independent) execution otherwise.
- *interleaving parallel composition*, see [74], which defines interleaved execution for all events.

In this thesis, synchronous parallel composition is denoted by $\|$, and interleaving parallel composition is denoted by $|||$.

Given $G_1 := (Q_1, \Sigma_1, \delta_1, Q_{0,1}, Q_{m,1})$ and $G_2 := (Q_2, \Sigma_2, \delta_2, Q_{0,2}, Q_{m,2})$, the synchronous parallel composition of G_1 and G_2, denoted by $G_1 \| G_2 := (Q, \Sigma, \delta, Q_0, Q_m)$, is defined as:

- $Q := Q_1 \times Q_2$,
- $\Sigma := \Sigma_1 \cup \Sigma_2$,
- $Q_0 := Q_{0,1} \times Q_{0,2}$,
- $Q_m := Q_{m,1} \times Q_{m,2}$,
- for each $q = (q_1, q_2) \in Q$, $\sigma \in \Sigma$:

$$\delta(q,\sigma) := \begin{cases} (\delta_1(q_1,\sigma), \delta_2(q_2,\sigma)) & \text{if } \sigma \in \Sigma_1 \cap \Sigma_2, \delta_1(q_1,\sigma) \text{ and } \delta_2(q_2,\sigma) \text{ defined} \\ (\delta_1(q_1,\sigma), q_2) & \text{if } \sigma \in \Sigma_1 - \Sigma_2 \text{ and } \delta_1(q_1,\sigma) \text{ defined} \\ (q_1, \delta_2(q_2,\sigma)) & \text{if } \sigma \in \Sigma_2 - \Sigma_1 \text{ and } \delta_2(q_2,\sigma) \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

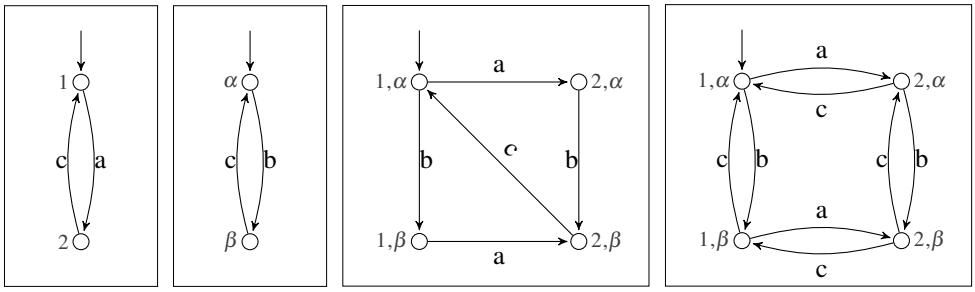
For the interleaving parallel composition, the transition function is redefined to return a *set of* locations:

- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$

The interleaving parallel composition of G_1 and G_2, denoted by $G_1 ||| G_2 := (Q, \Sigma, \delta, Q_0, Q_m)$, is then defined as:

- $Q := Q_1 \times Q_2$,
- $\Sigma := \Sigma_1 \cup \Sigma_2$,
- $Q_0 := Q_{0,1} \times Q_{0,2}$,
- $Q_m := Q_{m,1} \times Q_{m,2}$,
- for each $q = (q_1, q_2) \in Q$, $\sigma \in \Sigma$:
  $\delta(q,\sigma) := \{ (q_1', q_2) \mid q_1' \in \delta_1(q_1,\sigma) \} \cup \{ (q_1, q_2') \mid q_2' \in \delta_2(q_2,\sigma) \}$

Figure 3.3 shows an example for each of the two parallel composition operators. Note that interleaving parallel composition can result in nondeterministic models. Since the classical Ramadge-Wonham framework used in this thesis cannot handle nondeterministic systems, this could in principle lead to problems. However, in



(a) A **(b)** B **(c)** A ‖ B **(d)** A ||| B

Figure 3.3: synchronous and interleaving parallel composition

the cases in this thesis, interleaving does not result in a nondeterministic model. The individual automata themselves are nondeterministic, and the events which are shared by the automata that are composed by interleaving parallel composition, only occur in self-loops.

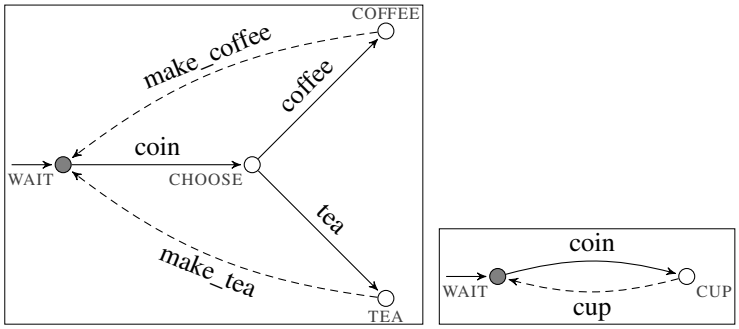
3.8 Supervisor realization

It is possible to realize a supervisor simply as another automaton S that is composed by means of synchronous parallel composition with the plant G. In this case the control actions of S on G are implicit in the transition structure of S. In detail, if $w \in L(S \parallel G)$ then it is required that $w \in L(S)$, and $w\sigma \in L(S)$ only if $\sigma \in f_s(w)$. In addition, if $w \in L(S \parallel G)$, $w\sigma \in (L(G)$, and $\sigma \in f_s(w)$, then $w\sigma \in L(S)$. The first condition ensures that those transitions disabled by f_s do not appear in the transition structure of S; while the second condition ensures that those transitions enabled by f_s, and which are possible in G, do appear in the transition structure of S. S and G are assumed to run in parallel, following the semantics of parallel composition with full synchronization.

3.9 Synthesis algorithm

Section 3.5 specifies that the supremal controllable sublanguage is derived by taking the union of all controllable sublanguages of a specification. In practice, algorithms to compute the supremal controllable sublanguage are based on the automaton representation of a language. Instead of taking the union of controllable sublanguages, states that are reachable by strings that do not conform to the specification are marked as bad.

As an example of an implementation, consider the following synthesis algorithm. The algorithm starts by constructing the plant automaton and marks the states that do not satisfy the requirements as bad. Then the algorithm proceeds via a two step iteration, until a fixed point (fixpoint) is reached. The first step marks all reachable states as bad from which an already bad state can be reached via an uncontrollable



(a) Coin, then coffee or thee. **(b)** Coin, then cup.

Figure 3.4: Coffee machine

event. The second step calculates the set of reachable states (starting from the initial states, and avoiding bad states), and marks all of the reachable states from which a marker state cannot be reached as bad. Note that when a state is marked as bad, the incoming controllable transitions can also be removed. Then the iteration returns to the first step, and goes on until no additional states can be marked as bad. The remaining states represent the controlled behavior.

3.10 Synthesis example: Coffee Machine

To illustrate the computation of a supervisor consider the following example of a coffee machine, see Figure 3.4. The machine consists of the parallel composition of the following two behaviors:

- After a coin is inserted, a choice can be made for coffee or tea, both controllable. Then the machine makes either coffee or tea, represented by uncontrollable events, as shown in Figure 3.4a.
- After a coin is inserted (controllable), the machine dispenses a plastic cup (uncontrollable), see Figure 3.4b.

Computing the parallel composition of these two automata results in the automaton shown in Figure 3.5. It is clear that the coffee machine has unsafe behavior: After the coin is inserted, the choice for coffee or tea can be made, even before the cup is dispensed. As a result, coffee or tea can be made when a cup is not yet ready, resulting in spilled coffee or tea.

To prevent spilling liquids, the control requirement as modeled in Figure 3.6 is introduced. This ensures that coffee or tea can only be made after a cup is dispensed.

Composing the control requirement with the plant model of the coffee machine results in the state `wait.cup` to be marked as bad, see Figure 3.7. In the (first) iteration of the algorithm, the uncontrollable events leading to the bad state are removed, and the source of the transitions are marked as bad, namely states `coffee.cup` and `tea.cup`, see Figure 3.8. From all remaining reachable states, a marker state can now

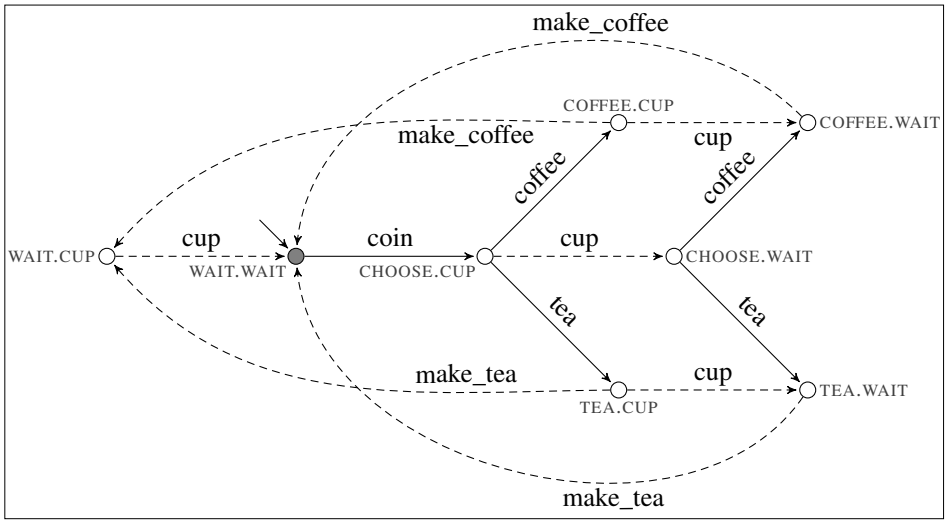


Figure 3.5: Parallel composition of the coffee machine

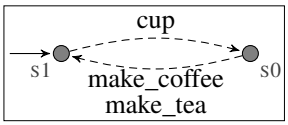


Figure 3.6: Coffee machine control requirement

be reached, so that the iteration terminates. In Figure 3.9, the two states `coffee.cup` and `tea.cup` are removed together with the incoming controlled transitions. Now the automaton conforms to the specification, and is controllable and non-blocking, see Figure 3.10.

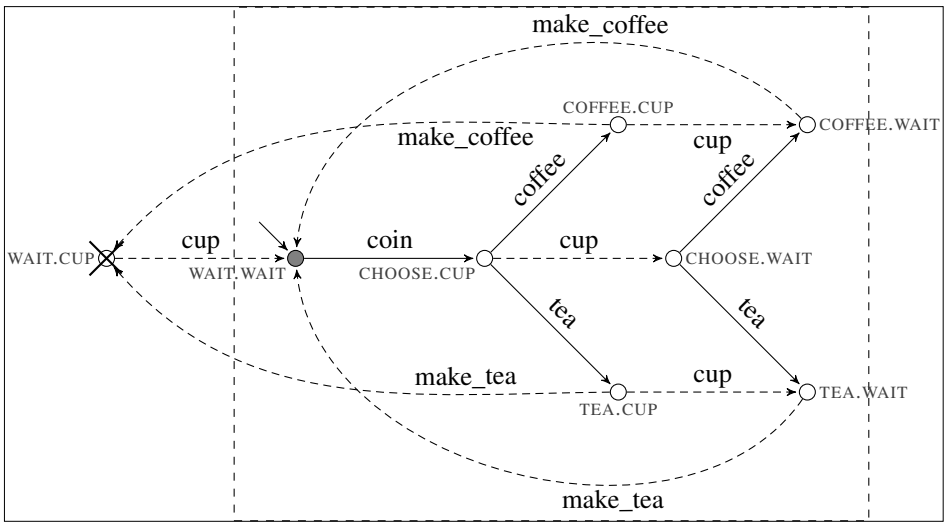


Figure 3.7: Initial marking of a bad state.

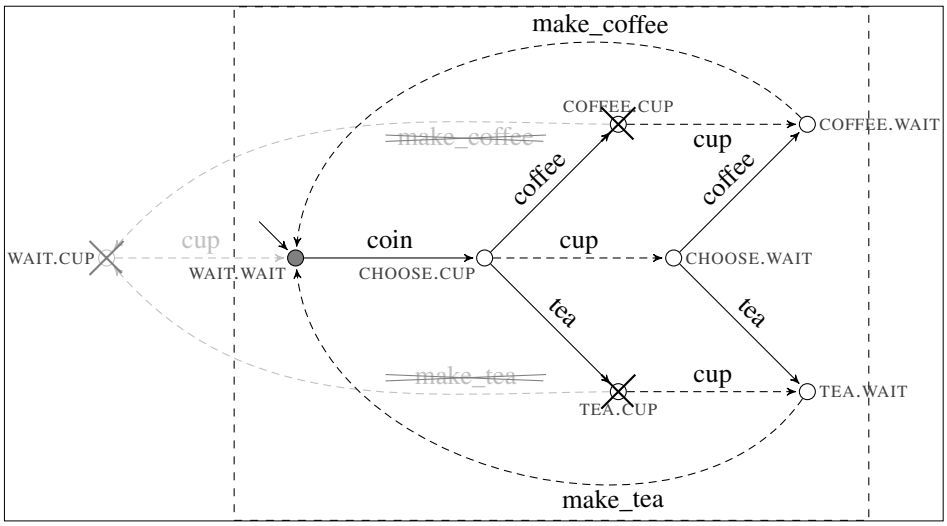


Figure 3.8: Additional marking of bad states via uncontrollable events.

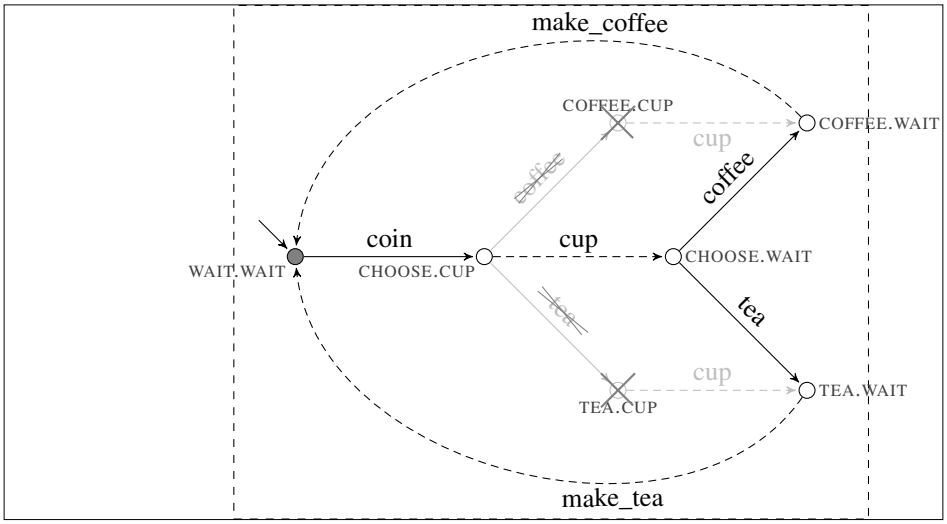


Figure 3.9: Removal of controllable events.

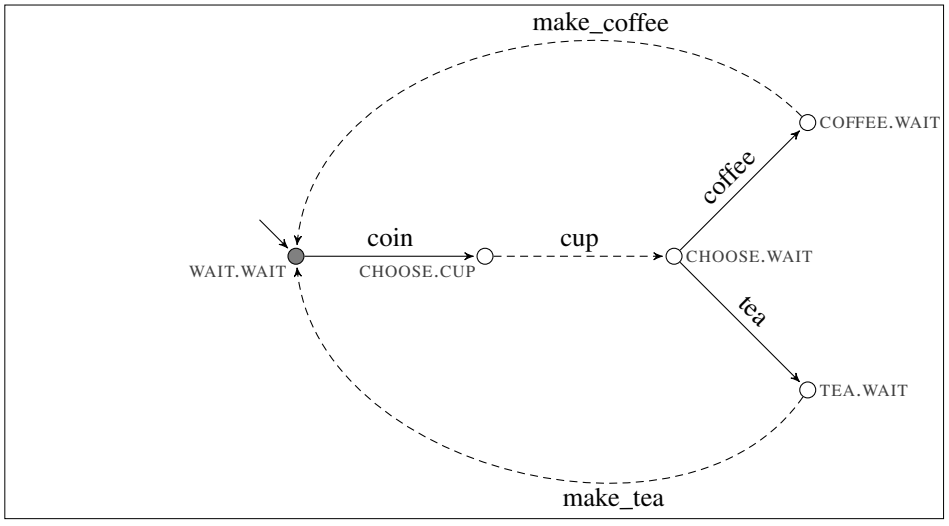


Figure 3.10: Coffee machine safe behavior.

Chapter 4

MRI scanners

Magnetic Resonance Imaging (MRI) is primarily a medical imaging technique used to visualize the internal structure and function of the human body. MRI is a fairly new technique, the first MR image of an inhomogeneous object, two tubes of water, was published in 1973 [37]. The first live human images were reported by Sir Peter Mansfield in 1976. By comparison, the first human X-ray image was taken in 1895. The first human body MRI exam was performed on July 3, 1977 [28]. Producing the image took almost five hours. By 1983, continuous improvements of MRI hardware and software had resulted in whole body imaging systems that were capable of producing high contrast images, with a spatial resolution of under 1mm, in only a few minutes.

Other medical imaging techniques include radiographic techniques (like conventional X-ray, and X-ray Computerized Tomography (CT)) and ultrasound. Radiographic techniques produce shadow images, resulting from the absorption of the X-ray photons by the body. The differences in tissue density provides the contrast in these images. Ultrasound produces echo images, resulting from the partial reflection of the sound waves by the layers between different tissues. Specifically, sound is reflected anywhere there are density changes in the body. MRI uses quantum mechanic properties of atoms to create its images. It does not use ionizing radiation like radiographic techniques. The contrast between different soft tissue in an MRI image is much higher then in a radiographic one. Furthermore, the spatial resolution is higher than in ultrasound. This makes MRI especially useful in neurological, musculoskeletal, cardiovascular and oncological imaging. The following sections describes the basic principle of MRI and the system overview, more detailed descriptions can be found in [78].

4.1 MRI physics

MRI scanners use a strong magnetic field to align the nuclear magnetization of (usually) hydrogen atoms in the body. The nuclei of certain atoms, such as hydrogen atoms, act like rotating microscopic magnets, due to quantum mechanic effects. Normally, these nuclei rotate in any direction. However, when these nuclei are subjected to a magnetic field, they align with this field either parallel or anti-parallel and precess about the field lines of the applied field. At room temperatures, slightly more nuclei align parallel with the applied field. This causes a net macroscopic magnetization of the object in the field, in the same direction as the field, the *longitudinal magnetization*. This net magnetization cannot be measured, because it is very small compared to the applied field.

Radio frequency (RF) fields are used to systematically alter the alignment of nuclear magnetization, causing the hydrogen nuclei to produce a rotating magnetic field that is detectable by the scanner. If the nuclei are excited at their resonance frequency by means of a Radio Frequency (RF) pulse, they align differently to the applied magnetic field. Nuclei can be aligned to rotate perpendicularly to the applied field. This lowers the macroscopic longitudinal magnetization, and creates an rotating magnetization perpendicular to the applied field, the *transverse magnetization*. This rotating transverse magnetization can be measured by means of a coil, since a changing magnetic field induces a current in the coil.

After the RF pulse is stopped, the nuclei return relatively slow to there equilibrium state, restoring the original longitudinal field. The recovery of the longitudinal magnetization is called T_1 relaxation (or spin-lattice relaxation). The decay of the transverse magnetization is called T_2 relaxation (or spin-spin relaxation). These two relaxations are independent, however they cannot be measured independently. Still, T_2 relaxation is much shorter than T_1 relaxation; therefore the signals can be separated. The relaxation times depend on the surrounding tissue type, different tissue types give different relaxation times, this provides the contrast in MR images.

The signal detected by the scanner, can be manipulated by additional magnetic fields to build up enough information to construct an image of the body. The resonance frequency of the nuclei depends on the field strength of the applied field. By making small variations in the applied magnetic field, in x, y and z directions, specific slices in the body can be excitated. This adds spatial encoding to the measured signals. This allows to construct (three dimensional) images of the body.

From the obtained measured signals, the images are reconstructed by applying Fourier transformations.

4.2 MRI system

The main subsystems of an MRI system are the static magnet, the magnetic field gradient system, the RF transmit system, the RF receive system, and the control system. Furthermore, the MRI system must provide subsystems to view and archive the produced images, to position the patient into the magnetic field, to interact with

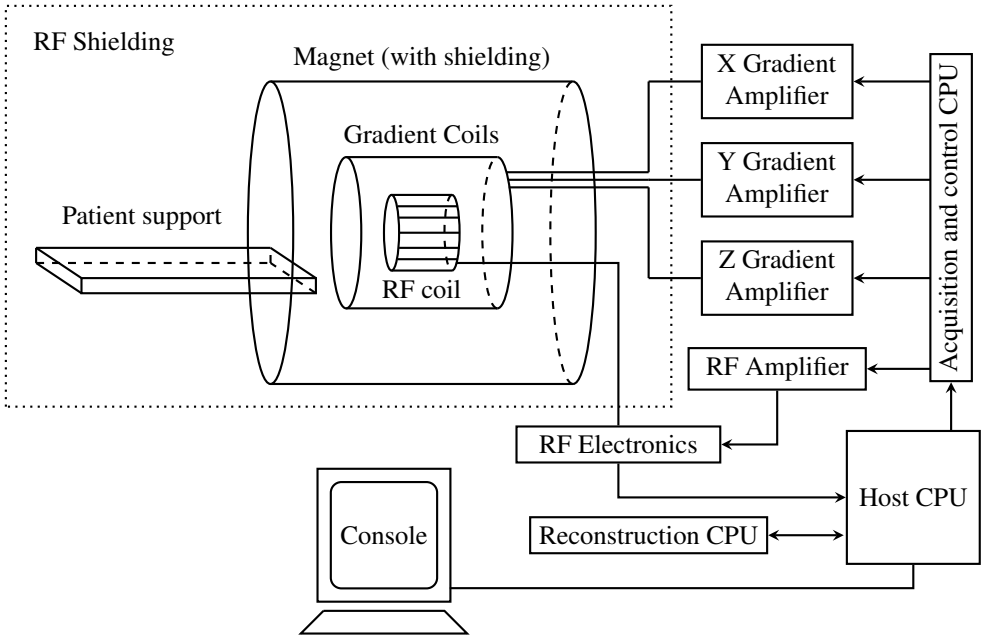


Figure 4.1: Basic MRI hardware.

the patient and to monitor the patient during a scan. A schematic view of the MRI system is given in Figure 4.1.

In modern systems, the static magnetic field is provided by a superconducting electromagnet. A special alloy is cooled by liquid helium; it looses resistance to the flow of electrical current. So once an inductive current has been introduced into the magnet coils, the system can be disconnected from the external power supply, while the current, and thus the magnetic field, is maintained. These magnets provide a stable and homogeneous magnetic field. The field strength provided by the magnet varies 1 T (Tesla) to 7 T. For comparison, the earth's magnetic field varies between 30–60 μT.

The magnetic gradient system is used to vary the magnetic field to add spatial encoding to the signals. It consists of coils in x, y and z direction, and amplifiers to power the coils. The gradient creates magnetic fields that change linearly with position, superimposed on the main magnetic field. The gradient fields can be oriented in any direction, by powering coils in combination. Typical gradient systems can achieve gradients from 20 mT/m to 100 mT/m. The direction of the gradient field can be switched quickly during an measurement. The slew rate of a gradient system is a measure of how quickly the gradients can be ramped up or down. Typical higher performance gradients have a slew rate of up to 100–200 T/ms. Switching these gradient fields causes the acoustic noise characteristic to MR measurements.

The RF transmit system consists of a RF synthesizer, power amplifier and transmission coil, to produce the RF pulses to excited the nuclei. The output power is variable, with a peak power of up to 35 kW.

The RF receiver system consists of a receive coil and signal processing systems. The RF transmission coil can be used as receiver too. However, for better signal to

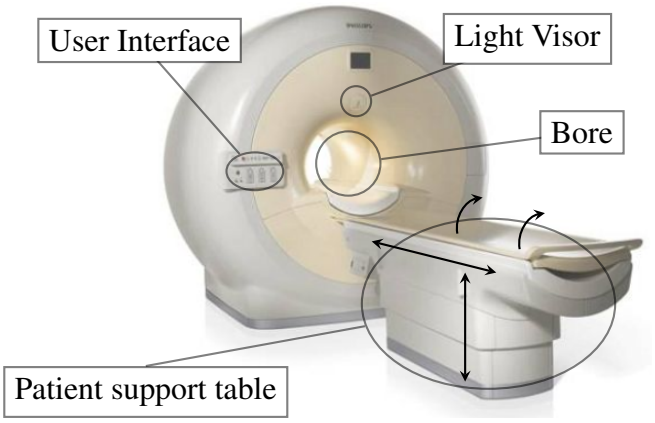


Figure 4.2: MRI scanner

noise ratio close fitting small coils are used. Many different coils are available which fit parts of the body, like the head, knee, wrist. The measured signal is amplified and converted by the signal processing systems. The resulting images are sent to a console for viewing and archiving.

The control system coordinates all previous subsystems. It performs the MR measurements based on the input parameters. It coordinates and synchronizes the gradient system and the RF transmit and receive systems.

The magnetic field generated by the MRI scanner is only homogeneous in a limited area. The area in which images can be taken is only about 50 cm, this area is called the *imaging volume*. The patient support system is used to position the part of the patient to be imaged into the imaging volume. In case that complete body images are needed, which exceed the imaging volume, the patient is moved by the patient support table during the MRI examination.

4.3 The patient environment

The patient support system is used to position a patient in an MRI scanner, see Figures 4.2 and 4.3. The MRI system is operated by two operators. One of the operators is in the examination room together with the patient. The other operator is in the control room that is separated from the examination room. Figure 4.4 shows a schematic layout of the MRI rooms. The main controller of the MRI system is called the MRI host system. Figure 4.5 shows the operator in the control room and the MRI scanner with patient in the examination room.

4.3.1 Support table

The patient support system is used to position a patient in an MRI scanner, see Figure 4.2. An MRI scanner is used mainly in medical diagnosis to render pictures of the inside of a patient non-invasively. The patient support system, see Figure 4.3, can be divided into the following components: vertical axis, horizontal axis and

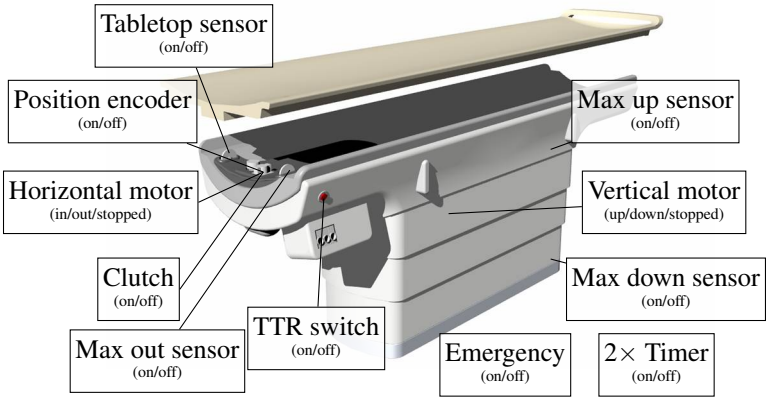


Figure 4.3: Patient table

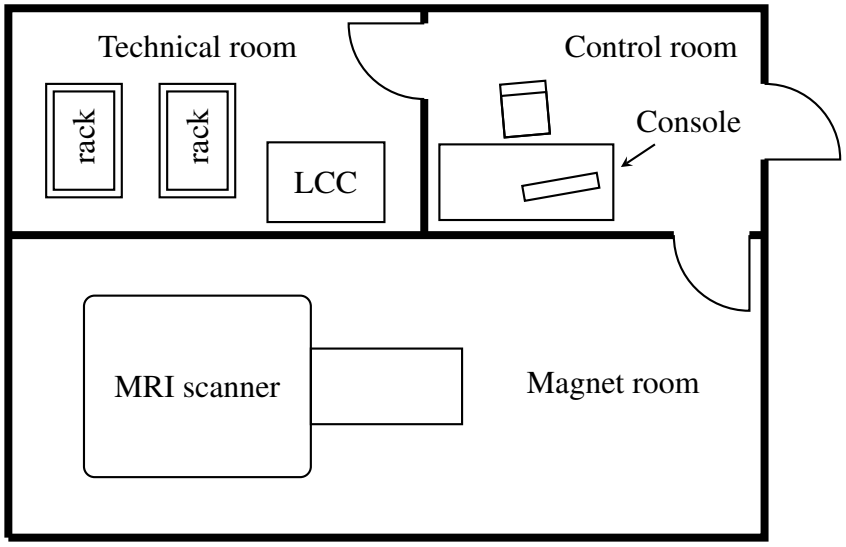


Figure 4.4: MRI rooms plan

user interface. The vertical axis consists of a lift with appropriate motor drive and end-sensors. The horizontal axis contains a removable tabletop which can be moved in and out of the bore, either by hand or by means of a motor drive depending on the state of the clutch. It contains sensors to detect both the presence of the tabletop and its position. The system is equipped with hardware safety systems, namely an emergency stop and the tabletop release switch TTR, that allow the operator to override the control system in emergency situations. Finally, the system contains a light-visor for marking that part of the patient's body which is scanned. This marking is then used by the MRI scanner to determine the correct position of the patient in the bore. The system is controlled via a user-interface UI. This interface contains a tumble-switch to control the movement of the table, and three buttons to control the clutch, the emergency system and the light-visor with automatic positioning. Furthermore, the UI contains LEDs to display the current state of the system to the operator.

Figure 4.5: The patient communication system.

The supervisor should accomplish multiple control objectives. When the operator operates the tumble-switch, the table should move up and down, or in and out of the bore. This depends on the current position of the table and the position of the tumble-switch. When the manual button is pushed, the clutch should be released such that the table can be moved manually by hand. Pushing the light-visor button, should enable the light-visor, and when the button is pushed again, the position is stored for automatic positioning. Finally, the table should not move beyond its end positions, and it should not collide with the magnet.

4.3.2 Communication system

The patient communication (sub)system is responsible for the delivery of all audio signals in the patient environment and control room. Audio signals originate from multiple sources. First, from the patient and the two operators. These signals are captured by microphones. Second, from an audio stream that can be played by the MRI host system to instruct the patient during a scan. For instance, the patient can be instructed to hold his breath during some stages of the scan. Finally, from an auxiliary device such as an MP3-player, that can play music to give extra comfort to the patient. The audio signals are delivered to the patient and the two operators via speakers and headphones.

The operators can control which audio signal is played. Furthermore, the patient has a nurse-call button, to draw attention of the operators in cause of an emergency. When the button is pushed, the operator is notified and the audio channel between the patient and operator should be opened.

For more information on the patient communication system, the reader is referred to Section 7.2.

Chapter 5

Event-based SCS of the patient support table

Background This chapter describes the main steps of event-based control synthesis for the patient support table described in Section 4.3. First, a discrete-event (finite-state automaton) model of the uncontrolled patient support table is created. Only the behavior of the plant in the absence of errors is modeled. Second, the control requirements are modeled as automata. Then, SCT is applied to obtain a supervisor. The obtained supervisor is first tested by means of simulation, using both untimed and hybrid plant models. Second, the actual patient support system is controlled by means of the model of the synthesized supervisor executing in real-time, and also by means of real-time control code, obtained by model transformation from the synthesized supervisor. No errors could be detected in the final controlled patient support system, after extensive testing.

Contribution Dividing the control requirements into small, modular, finite state automata specifications, is shown to lead to models that can be easily adapted to changing requirements. This evolvability of the controller is tested by redesigning it for slightly different control requirements; the design and implementation, of the new controller on an actual MRI scanner, took only half a day, see Section 5.4.2 for more details.

Developing the models and modeling techniques is work of the author. Developing the (proptotype) toolchain was joint work with Ramon Schiffelers. This toolchain was later elaborated by Dennis Hendriks of the Systems Engineering Group at TU/e, resulting in the CIF 3 tool set, available at [69].

Apart from the advantages of applying supervisory control synthesis for the control of the patient support system, in terms of improved evolvability, also a number of deficiencies are indicated. These are elaborated in the 'Concluding remarks' in Section 5.6. They relate to, first, the duplication of information in plant models and control requirements, second, to the limitations imposed by the use of a single initial state, and third, to the limited tool support for debugging supervisory

control specifications.

Finally, note that the models presented in this chapter are slight modifications of the models which were used to synthesize the supervisor that was implemented on existing hardware. For instance, the sensors models in Figures 5.5a and 5.5b are added for clarity. Omitting these models does not change the functionality because the parallel composition of these models with the model in Figure 5.6a is equal to Figure 5.6a itself. These modifications are applied to render the models easier to understand. However, they do not influence the functional behavior of the models. Note that software simulation reveals that the supervisor obtained from the models of this chapter also satisfies the control requirements.

Some of the results of the chapter were announced in conference proceedings [71]. With respect to [71], the main difference is that this chapter presents more detailed models and a modeling methodology, in addition this chapter systematically explains the role of supervisory control in improving evolvability. An abbreviated journal version of this chapter was published as [73].

Outline The outline of the chapter is as follows. Section 5.1 contains an informal description of the case-study. Section 5.2 presents the concepts and design considerations underlying the formal models of the plant components and control requirements. The specifications of the models can be found in Section 5.3. The automatic generation of the supervisory controller, its implementation, and the validation of the obtained controller by means of simulation are all discussed in Section 5.5. Section 5.6 presents conclusions and proposes directions for future research.

5.1 Functionality of the patient support system

The goal of this case study is to design a supervisor for the patient support system of an MRI scanner, see Section 4.3.1. The patient support system is more difficult to control than might appear at first sight. It contains several complex interactions of components. Even though the version of the patient support system that is discussed in this chapter is simplified compared to the real system, the control requirements consist of 62720 reachable states, and 869520 transitions. Another parameter which reflects the complexity is that the plant model together with the model of the control requirements, as discussed in this chapter, consists of 31 automata of 2 to 4 states, and one automaton with 7 states.

The complexity of the designer's task becomes even more apparent when one considers the time which is required to build the control software manually. In fact, it was estimated that one would need a week for manual adaptation of the control software to meet the modified requirements described in Section 5.4.2. Note that with the approach of this chapter, the adaptation of the models of control requirements and the generation of new control software took merely half a day.

5.2 Modeling concepts

The goal of this section is to sketch the structure of the models of the plant and of the control requirements. The detailed models can be found in Section 5.3. The classical Ramadge-Wonham framework [79], see Chapter 3, is used for modeling the plant, the supervisor and the requirements.

5.2.1 Plant model

The plant is represented as a parallel interconnection of automata representing the following plant components:

- Model of the vertical axis VAxis,
- Model of the horizontal axis HAxis,
- Model of the user interface UI, which describes the effect of the actions of the operator on the plant.

The model of the complete uncontrolled patient support system (which is referred to as the plant model) is thus defined as the following synchronous parallel composition:

$$\mathsf{VAxis} \parallel \mathsf{HAxis} \parallel \mathsf{UI}$$

Each component above represents a separate aspect of the plant's functionality. The components themselves are modeled as a parallel interconnection of the models of the: *actuators*, the *sensors*, and the *relations between the actuators and sensors*. The actuator-sensor relations represent the physical structure of the machine, relating actions of actuators to activations of sensors.

Thus for example, the model of the vertical axis is of the form

$$\mathsf{VAxis} \triangleq \mathsf{VActuators} \parallel \mathsf{VSensors} \parallel \mathsf{VRelations}$$

Here, VActuators describes the model of the motor which moves the vertical axis. It essentially defines that the motor can be started and then stopped. The automaton VSensors models the functioning of the sensors which detect if the table is in the maximally down or up position. It defines that a sensor can be switched on when it is off, and vice versa. Finally, VRelations models the actuator-sensor relation: as a result of their physical positions, the vertical sensors cannot be active at the same time. Furthermore, the sensors cannot change their state if the motor is not moving. In somewhat more detail: the maximally up sensor may be activated only when the table is moving up, and the maximally down sensor may be activated only when the table is moving down.

The detailed description of the vertical axis model as well as the models of the other components can be found in Section 5.3.

5.2.2 Control requirements

The control requirements of interest are *safety* requirements. More precisely, the control requirements are sets of sequences of events which the closed-loop system

is allowed to generate. The control requirements are represented by finite-state automata. The language accepted by the automaton is exactly the set of all *safe* sequences of events which the closed-loop system is allowed to generate.

The components of the control requirements reflect the components of the plant. That is, for each plant component the corresponding control requirement is modeled separately. Hence, the model of the control requirements is of the form

VReq || HReq || HVReq || UIReq

Here, VReq is the model of the control requirements for the vertical axis, HReq is the model of the control requirements for the horizontal axis, HVReq is the model of control requirements pertaining to the interaction between the horizontal and vertical axis, and finally UIReq is the model of the control requirements for the interface. For example, VReq formalizes the following requirements. Movement beyond the maximally up position is not allowed. This implies that initiating movement in the upper direction must be prevented when the table is maximally up. Furthermore, movement in the upper direction must be stopped when the table reaches the maximally up position. The detailed models can be found in Section 5.3.

5.2.3 Modeling logical conjunction and disjunction

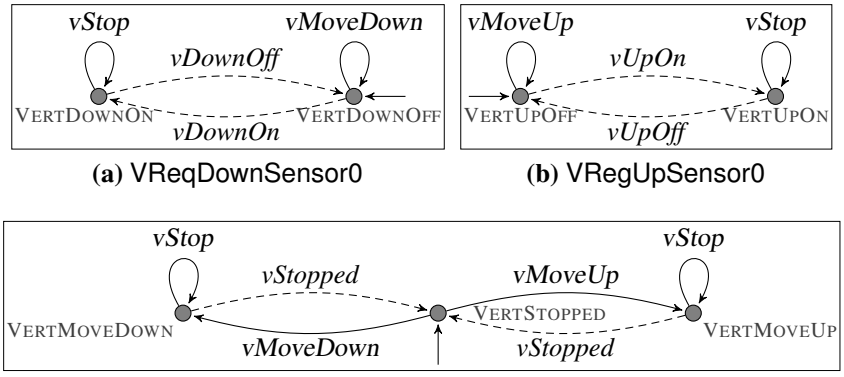
The plant and control requirements are divided into sub-specifications that are each modeled by means of small, relatively independent, automata. Composing requirements by means of a network of parallel automata leads to the conjunction of the requirements, as a result of the semantics of synchronous execution of shared events. To obtain logical disjunction in control requirements is more difficult. This will be explained by means of small fragments of control requirements relating to table movement along the vertical axis, where two end sensors are defined: a vertically down sensor, and a vertically up sensor. Movement is controlled by means of a motor that has commands to move down, move up, or stop. After the stop command has been received, the table slows down to come to a halt, which will be reported by the *vStopped* event.

Consider, for example, the specification of Figure 5.1. The first requirement VReqDownSensor0 specifies for the motor movement that the *vMoveDown* event may occur only when the vertically down sensor is in the state `VertDownOff`. The third requirement VReqMotor specifies that the *vMoveDown* event may occur only when the motor is in the state `VertStopped`. The semantics of synchronous execution of the shared event *vMoveDown* enforces the conjunction of the two requirements: the *vMoveDown* event can occur only in the state described by the predicate $\texttt{VertDownOff} \wedge \texttt{VertStopped}$.

The specification VReqMotor specifies for the stop event that it is allowed only in the state $(\texttt{VertMoveDown} \vee \texttt{VertMoveUp})$. The combination of the three specifications, thus defines that the stop event *vStop* is allowed only in the state

$$\texttt{VertDownOn} \wedge \texttt{VertUpOn} \wedge (\texttt{VertMoveDown} \vee \texttt{VertMoveUp}).$$

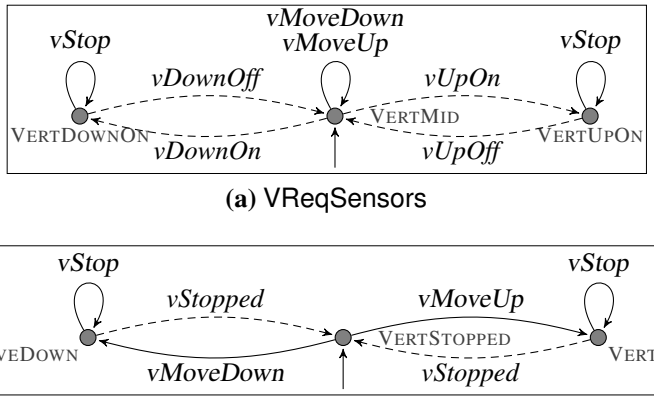
This is obviously not the required behavior, since the stop event is then allowed only



(a) VReqDownSensor0 **(b)** VRegUpSensor0

(c) VReqMotor

Figure 5.1: Models VReq0: incorrect requirements with single stop event



(a) VReqSensors

(b) VReqMotor

Figure 5.2: Model VReq1: incorrect requirements with single stop event

when the down sensor and the up sensor are active at the same time.

To obtain the disjunction of the two states `VertDownOn` and `VertUpOn`, the two sensor models can be merged into one sensor model, as specified by Figure 5.2a. The combination of the two requirements VReqSensors and VReqMotor results in the predicate

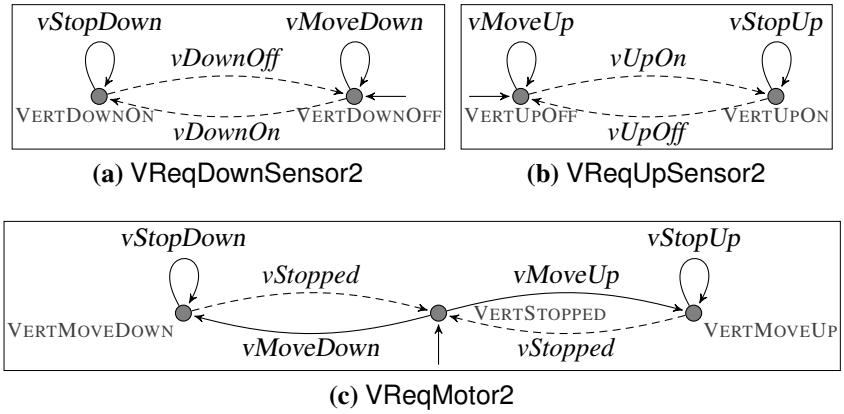
$$(\texttt{VertDownOn} \vee \texttt{VertUpOn}) \wedge (\texttt{VertMoveDown} \vee \texttt{VertMoveUp}),$$

that defines when the *vStop* event is enabled. This also is not correct, because the required predicate is

$$(\texttt{VertDownOn} \wedge \texttt{VertMoveDown}) \vee (\texttt{VertUpOn} \wedge \texttt{VertMoveUp}).$$

Furthermore, the specification from Figure 5.2a defines that the *vMoveDown* event (and also the *vMoveUp* event) is allowed only in the case that the vertically up and down sensors are both off. This requirement is too strong, since it disables downward movement in the case that the vertically up sensor is on.

To obtain the correct specification, a technique that can be referred to as 'event splitting', or 'event duplication' is used: instead of the single event *vStop*, two events



(a) VReqDownSensor2 **(b)** VReqUpSensor2

(c) VReqMotor2

Figure 5.3: Model VReq2: correct requirements with multiple stop events

vStopDown and *vStopUp* are used, leading to the correct specification of Figure 5.3. The motor is switched of whenever any of the two events *vStopDown* or *vStopUp* occurs. For the complete models, see Section 5.3.1.

5.3 Models

Below the formal models for the plants and control requirements are presented. The presentation is done component-wise. That is, first the plant model and the model of control requirements for the vertical axis are presented, then for the horizontal one, then for the interaction of the axes, and finally the plant model and the model of control requirements for the user interface are presented.

In the models, states are denoted by vertices, initial states are indicated by an unconnected incoming arrow, and marked states are denoted by filled vertices. Controllable and uncontrollable events are depicted by solid and dashed edges, respectively. If several event names are indicated on an edge, then this should be understood as a collection of edges (with the same source and target as the depicted edge), with each edge labelled by one of the events. A bold event name indicates a set of events, representing an edge for each event in the set.

5.3.1 Vertical axis

The patient table can move up and down along the vertical axis. The vertical axis contains two end sensors, maximally up and maximally down, and one actuator for the vertical motor drive. The motor can move the table up and down. The system should never move beyond the maximally up and down position.

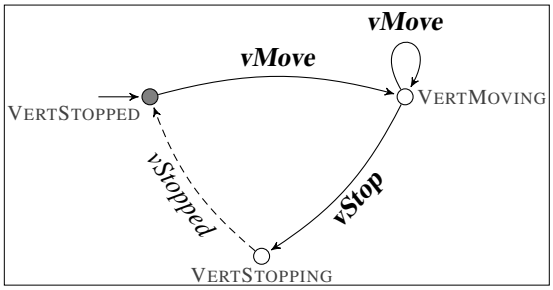


Figure 5.4: Model VActuators: vertical actuators

Plant model (VAxis)

The plant model of the vertical axis consists of the synchronous parallel composition of the models of the actuators, sensors and structure of the vertical axis:

$$\mathsf{VAxis} \triangleq \mathsf{VActuators} \parallel \mathsf{VSensors} \parallel \mathsf{VRelations}$$

The actuator, sensor and structure models are defined as:

$$\mathsf{VActuators} \triangleq \mathsf{VMotor}$$

$$\mathsf{VSensors} \triangleq \mathsf{VUpSensor} \parallel \mathsf{VDownSensor}$$

$$\mathsf{VRelations} \triangleq \mathsf{VSensorsRelation} \parallel \mathsf{VMotorSensorRelation}$$

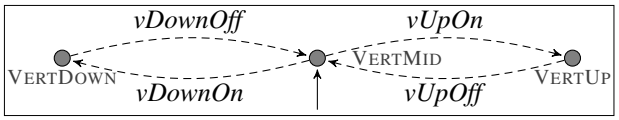
Motor drive (VMotor) The motor is controlled by a resource controller, which controls the brakes, and calculates set-points for the feedback control loop. The model of the motor drive only includes the behavior exposed to the supervisor, see Figure 5.4. Initially, the motor is stopped. From this state, a movement can be started. The event set ***vMove*** is an abbreviation for two move events: ***vMove*** $\triangleq$ {*vMoveUp*, *vMoveDown*}. If the motor is moving and a stop event in the set ***vStop*** (***vStop*** $\triangleq$ {*vStopUp*, *vStopDown*, *vStopTTR*, *vStopTumble*}, is triggered, the motor slows down to come to a halt. When the motor has come to a halt, the event *vStopped* is emitted, and the motor enters the stopped state again. Only the stopped state is marked, because the motor must always be able to return to the stopped state.

Distinguishing different stop events facilitates decomposition of the complete stop behavior into multiple independent requirements. Individual stop events should be enabled in distinct cases. For instance, *vStopUp* is enabled only when the table has reached its maximally up position. By having a different stop event for each case, these stop events do not synchronize, so that the cases can be modeled independently of one another. The stop events *vStopUp* and *vStopDown* are used in the control requirements of Figures 5.7a–5.7c, whereas the stop events *vStopTTR* and *vStopTumble* are used in the control requirements of Figures 5.14a and 5.16a, respectively.

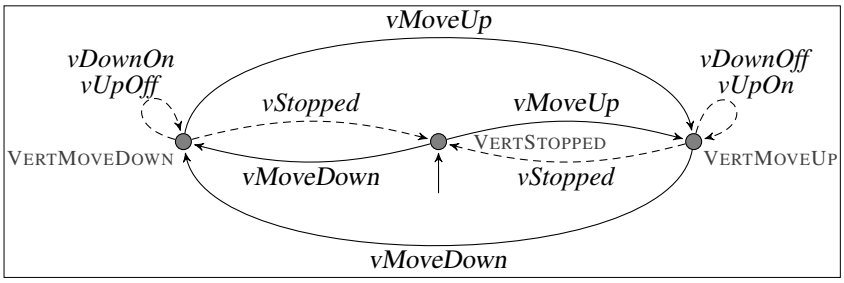
Sensors (VDownSensor, VUpSensor) The maximally down and maximally up sensors are modeled in Figures 5.5a and 5.5b. The sensors are active if the table is at the sensor position, otherwise the sensors are inactive. This is modeled by means of two (marked) states, namely `On` and `Off`. The sensors emit the uncontrollable

(a) VDownSensor **(b)** VUpSensor

Figure 5.5: Models VSensors: vertical sensors



(a) VSensorsRelation



(b) VMotorSensorRelation

Figure 5.6: Models VRelations: vertical structure

events *vDownOn* (*vUpOn*) or *vDownOff* (*vUpOff*), when a sensor becomes active or ceases to be active, respectively. Initially the table is assumed to be neither up or down, so that both end sensors are inactive, indicated by the states `VertDownOff` and `VertUpOff`.

Sensor-sensor relation (VSensorsRelation) The two sensors are never active at the same time, as a result of their physical location. This relation is modeled in Figure 5.6a. The model includes the complete behavior of the two individual sensors.

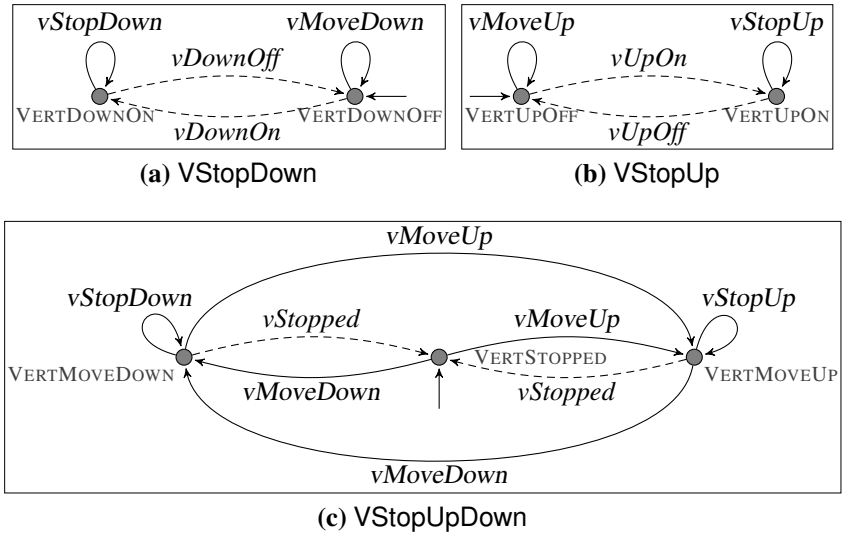
Motor-sensor relation (VMotorSensorRelation) The sensors do not change state when the table is not moving vertically, see Figure 5.6b. Only when the motor drive is moving the table up, the maximally down sensor can turn off (*vDownOff*) and the maximally up sensor can turn on (*vUpOn*), and likewise for the opposite direction.

Control requirements (VReq)

The requirement model of the vertical axis consists of the synchronous parallel composition of multiple control requirements, namely:

$$\mathsf{VReq} \triangleq \mathsf{VStopDown} \parallel \mathsf{VStopUp} \parallel \mathsf{VStopUpDown}$$

Maximally up and down (VStopDown, VStopUp, VStopUpDown) Movement beyond the maximally up position is not allowed. This implies that initiating movement in the upper direction must be prevented when the table is maximally up. Further-



(a) VStopDown **(b)** VStopUp

(c) VStopUpDown

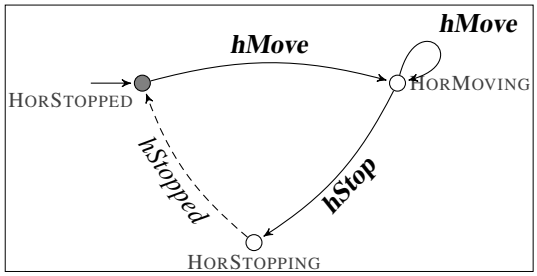
Figure 5.7: Model VReq: vertical axis requirements

more, movement in the upper direction must be stopped when the table reaches the maximally up position. Likewise it is not allowed to move beyond the maximally down position.

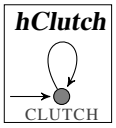
These requirements are modeled in Figures 5.7a–5.7c. First, the event *vMoveDown* (*vMoveUp*) is only allowed if the table is not maximally down (up). Second, the event *vStopDown* (*vStopUp*) is enabled only in the maximally down (up) position. This allows the table to stop when the end position has been reached. Finally, the *vStopDown* (*vStopUp*) event is enabled only if the motor is moving down (up), see Figure 5.7c. The synchronizing semantics of the parallel composition in VStopDown ‖ VStopUp ‖ VStopUpDown ensures that event *vStopDown* (*vStopUp*) is enabled only if the states `VertDownOn` (`VertUpOn`) and `VertMoveDown` (`VertMoveUp`) are both active (see Figures 5.7a, 5.7b, and 5.7c).

5.3.2 Horizontal axis

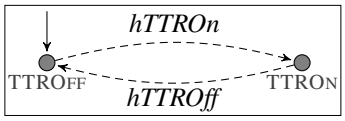
The horizontal axis consists of a removable tabletop on top of the main table. The tabletop (if present) can be moved in and out of the bore. It can be added and removed only in the maximally out position. The presence of the tabletop is detected by a sensor. Like the vertical axis, the horizontal axis contains two end sensors, maximally in and maximally out, and a motor drive. The motor drive is coupled to the tabletop by a clutch. When the clutch is released, the tabletop (if present) can be moved freely by an operator, otherwise the positioning is controlled through the motor drive. Finally, the control of the clutch can be overridden by a hardware safety system, called Table Top Release (TTR). The clutch is released when the TTR switch is active, independently of the controller. The system should never move beyond the maximally in and maximally out positions. Furthermore, the horizontal axis may not move, when the tabletop is not present, when the clutch is released or when the TTR



(a) HMotor



(b) HClutch



(c) HTTRSwitch

Figure 5.8: Model HActuators: horizontal actuators

switch is active.

Plant model (HAxis)

The plant model of the horizontal axis consists of the synchronous parallel composition of the models of the actuators, sensors and structure of the horizontal axis:

$$\mathsf{HAxis} \triangleq \mathsf{HActuators} \parallel \mathsf{HSensors} \parallel \mathsf{HRelations}$$

The actuator, sensor and structure models are defined as:

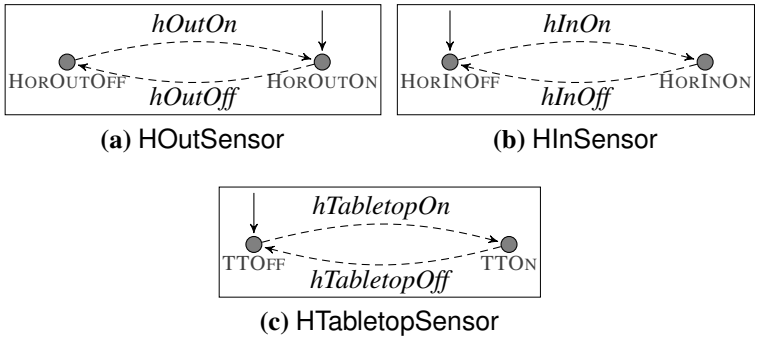
$$\mathsf{HActuators} \triangleq \mathsf{HMotor} \parallel \mathsf{HClutch} \parallel \mathsf{HTTRSwitch}$$

$$\mathsf{HSensors} \triangleq \mathsf{HInSensor} \parallel \mathsf{HOutSensor} \parallel \mathsf{HTabletopSensor}$$

$$\mathsf{HRelations} \triangleq \mathsf{HSensorsRelation} \parallel \mathsf{HActuatorSensorRelations}$$

Actuators (HMotor, HClutch, HTTRSwitch) The horizontal motor drive is similar to the vertical motor drive, see Figure 5.8a. The event set ***hStop*** denotes a number of different stop events, namely ***hStop*** $\triangleq$ {*hStopIn*, *hStopOut*, *hStopTTR*, *hStopTabletop*, *hStopTumble*}. The event set ***hClutch*** is an abbreviation for the clutch events, ***hClutch*** $\triangleq$ {*hClutchOn*, *hClutchOff*}, and the event set ***hMove*** is an abbreviation for the move events, ***hMove*** $\triangleq$ {*hMoveIn*, *hMoveOutNormal*, *hMoveOutRestricted*}. The clutch is modeled with one state (`Clutch`) in which the clutch events are self-looped, see Figure 5.8b. The tabletop release switch is modeled similarly to a sensor, see Figure 5.8c.

Sensors (HInSensor, HOutSensor, HTabletopSensor) The sensors of the horizontal axis are modeled similarly to the sensors in the vertical axis, see Figures 5.9b–(c). Initially, the tabletop is not present (`TTOff`), the maximally out sensor is on (`hOutOn`), and the maximally in sensor is off (`vMaxInOff`).



(a) HOutSensor **(b)** HInSensor

(c) HTabletopSensor

Figure 5.9: Model HSensors: horizontal sensors

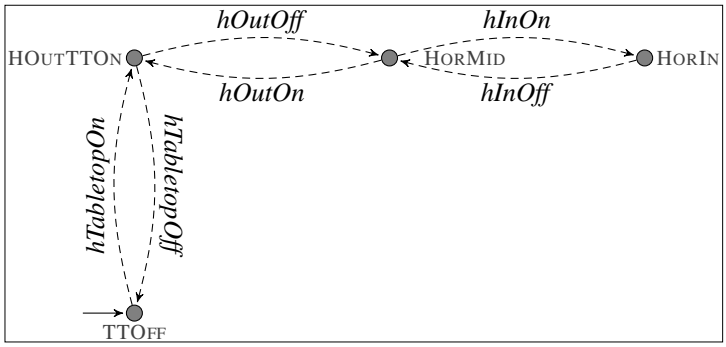


Figure 5.10: Model HSensorsRelation: horizontal structure (synchronizing parallel composition part)

Sensors relations (HSensorsRelation) The two end-sensors cannot be active at the same time, as result of their physical location, see Figure 5.10. Furthermore, the tabletop can only be added and removed in the maximally out position. Note that this model includes the complete behavior of the individual sensors.

Sensor-actuator relation (HActuatorSensorRelations) Only when the tabletop is present and is moving horizontally, can the maximally in and out sensors change state. The tabletop can move horizontally in three distinct cases:

- The clutch is released, see Figure 5.11a; the table can be moved by hand, therefore the sensors can always switch in any order. Note that the state of the clutch is defined by the control system.
- The TTR switch is activated, see Figure 5.11b; the table can be moved by hand, as in the case that the clutch is released. Note that, the state of the TTR switch is defined by the operator.
- If the clutch is applied, and the TTR switch is not active, the movement is controlled by the motor, see Figure 5.11c; analogous to the vertical axis.

That is, if either the clutch is released or the TTR switch is activated, then the table can be moved manually, and hence the sensors at the maximally in and maximally

out positions can be activated in any order. This behavior can be defined by a single, 12-state, automaton, representing the interleaving (non-synchronising) parallel composition (cartesian product) of the three automata of Figure 5.10 (see Figure 3.3 for the difference between synchronising and interleaving parallel composition):

$$\begin{aligned}\mathsf{HActuatorSensorRelations} \triangleq\ & \mathsf{HClutchSensorRelation} ||| \\ & \mathsf{HTTRSensorRelation} ||| \\ & \mathsf{HMotorSensorRelation}\end{aligned}$$

The only shared events in the three automata are the events *hInOn*, *hInOff*, *hOutOn*, *hOutOff*, occurring as self loops. Because the other events (*hClutchOn*, *hClutchOff*, *hTTROn*, *hTTROff*, *hMoveIn*, *hMoveOut*, *hStopped*) each occur in one automaton only, the only difference between the synchronizing and interleaving parallel composition of the three automata is in the self loops: the states, and the transitions between the states are identical. Therefore, the interleaving parallel composition of the three automata is still a deterministic automaton. Each state of the parallel composition (interleaving or synchronous) consists of the three sub-states of the respective automata. The difference between the synchronous and interleaving parallel composition is that the set of self loops of each state of the interleaving parallel composition is obtained by taking the *union* of the sets of self loops of the three sub-states, whereas for the synchronous parallel composition, the *intersection* of the sets of self loops of the sub-states is taken.

Note that if synchronous parallel composition had been used instead of the interleaving one, models would have been obtained which are inconsistent with the physical system. For example, consider the synchronous parallel composition of HClutchSensorRelation and HMotorSensorRelation. Notice that in HClutchSensorRelation, the events *hInOn*, *hInOff*, *hOutOn*, *hOutOff* can occur only in the initial state. In contrast, none of these events can occur in the initial state of HMotorSensorRelation. Hence, none of the events *hInOn*, *hInOff*, *hOutOn*, *hOutOff* can occur in the initial state of the synchronous composition of HClutchSensorRelation and HMotorSensorRelation. This contradicts the physical behavior that should be modeled.

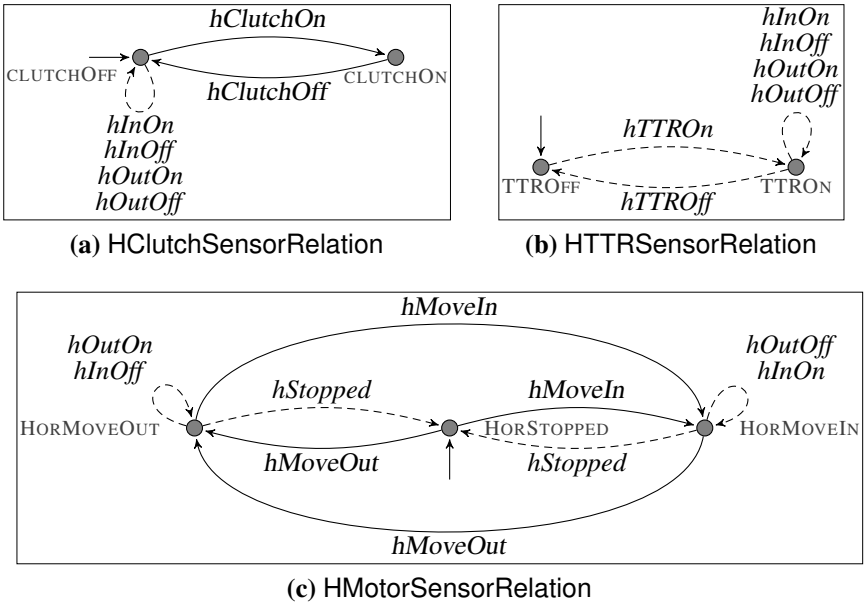
Control requirements (HReq)

The requirement model of the horizontal axis consists of the synchronous parallel composition of multiple control requirements, namely:

$$\begin{aligned}\mathsf{HReq} \triangleq\ & \mathsf{HStopIn} \parallel \mathsf{HStopOut} \parallel \mathsf{HStopInOut} \parallel \\ & \mathsf{HStopTabletop} \parallel \mathsf{HStopTTR} \parallel \mathsf{HClutchMove}\end{aligned}$$

Maximally in and out (HStopIn, HStopOut, HStopInOut) The horizontal axis may not move beyond its maximally in and out positions, see Figures 5.12b–5.12c.

Tabletop move (HStopTabletop) When the tabletop is not present, initiating horizontal movement is not allowed and the motor should be stopped, see Figure 5.12d.



(a) HClutchSensorRelation **(b)** HTTRSensorRelation

(c) HMotorSensorRelation

Figure 5.11: Model HActuatorSensorRelations: Actuator sensor relation (interleaving parallel composition)

TTR move and clutch (HStopTTR) Commands for horizontal movement and clutch commands may only be issued when the TTR switch is off, see Figure 5.12e. However, it cannot be prevented that the TTR switch is turned on while moving. Whenever the TTR switch is turned on, the table should be stopped (*hStopTTR*).

Clutch move (HClutchMove) The tabletop may only be moved by the horizontal motor if the clutch is applied, see Figure 5.12f. If the clutch is not applied, the motor may not move the table. While the motor is moving the table, the clutch may not be released.

5.3.3 Horizontal and vertical axis interaction

There is no physical interaction between the transducers of the horizontal and vertical axis. However, the tabletop might collide with the magnet, when moving inward if the table is not maximally up, or the table might be damaged, when moving downward if it is not maximally out. These situations must be prevented. Therefore, either the maximally out sensor or the maximally up sensor must be on, unless the TTR switch is on. If the TTR switch is on, the table can be moved freely by the operator. In this case, the control system cannot prevent the situation in which both sensors are off.

Plant model (HVNormal)

In Figure 5.13 the internal event *normal* is introduced. This event is used to distinguish two internal states in the requirements, namely, 1) control after TTR is

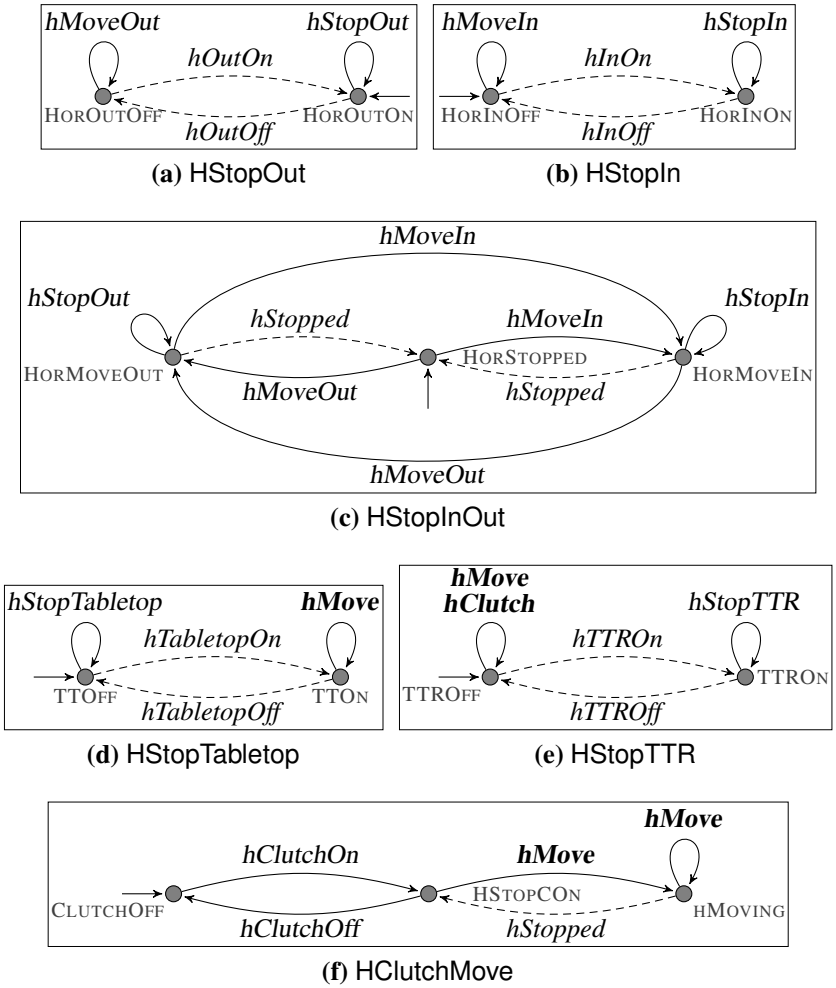


Figure 5.12: Model HReq: horizontal control requirements

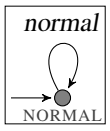


Figure 5.13: Model HVNormal: internal event *normal*

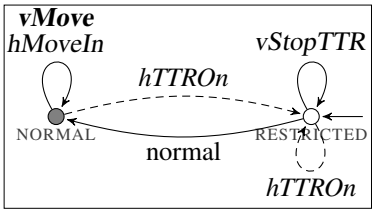
activated, and 2) control after the normal event has occurred, see Figure 5.14a.

Control requirements (HVReq)

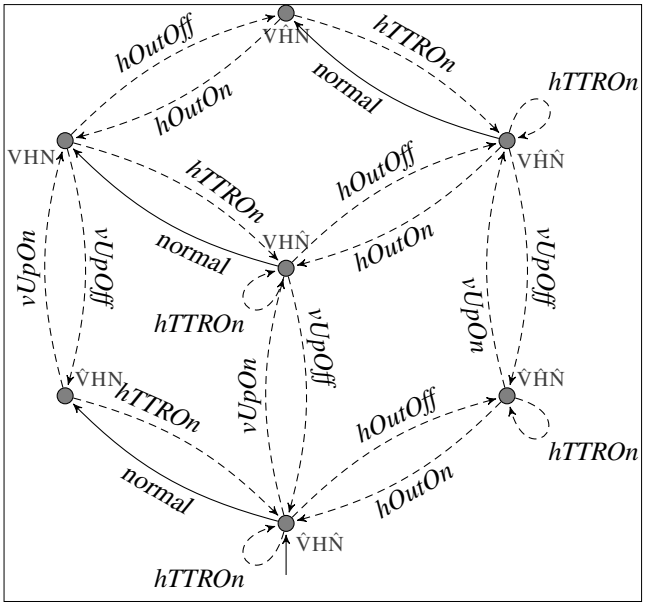
The control requirements for horizontal and vertical interaction are defined by the synchronizing parallel composition of two models:

$$\mathsf{HVReq} \triangleq \mathsf{HVMode} \parallel \mathsf{HVSafe}$$

Movement restrictions (HVMode) Initially the system is in the state `restricted`, see Figure 5.14a. In this state, the table may not move horizontally inwards



(a) HVMode



(b) HVSafe

Figure 5.14: Model HVReq: horizontal and vertical restrictions

(*hMoveIn*), and all vertical movements (***vMove***) are disabled. After the event *normal*, the system enters the state `normal`, in which all movement events are allowed. After occurrence of the event *hTTROn*, the system enters the state `restricted` again.

Normal operation (HVSafe) The system can switch to normal operation if it can be ensured that the system stays either maximally out, or maximally up, see Figure 5.14b. Normal mode is represented by the states $\hat{\texttt{v}}\texttt{hn}$, `vhn` and $\texttt{v}\hat{\texttt{h}}\texttt{n}$. In these states it is ensured that the table remains either maximally out or maximally up. The letters `v`, `h` and `n` represent the states vertically maximally up, horizontally maximally out, and normal, respectively. The hat represents negation, e.g. $\hat{\texttt{v}}$ represents not vertically maximally up. After an event *hTTROn*, any horizontal or vertical position can be reached (corresponding to the states $\hat{\texttt{v}}\texttt{h}\hat{\texttt{n}}$, $\texttt{vh}\hat{\texttt{n}}$, $\texttt{v}\hat{\texttt{h}}\hat{\texttt{n}}$ and $\hat{\texttt{v}}\hat{\texttt{h}}\hat{\texttt{n}}$).

Notice that in Figure 5.14b state $\hat{\texttt{v}}\hat{\texttt{h}}\texttt{n}$ is not present. The control requirement forbids that this state be reached. Therefore, all events leading to this state are disabled in the requirement model. The controller synthesis algorithm ensures that this requirement is met by disabling only controllable events. For instance, in normal mode when the table is not maximally up, the supervisor will ensure that the clutch is enabled and horizontal movement is prohibited.

5.3.4 User interface

The user can control the system by means of a button and a tumble-switch. When the button is pushed, the clutch is released (applied) to switch the table to manual mode (motorized mode). Therefore, this button is called the 'manual button'. In the motorized mode, the position of the tumble-switch determines the movement of the table. A LED is used to indicate whether the system is in the manual mode or in the motorized operation mode. Note that in manual mode, the supervisor can still prevent the table from performing operations requested by the user, such as moving the table motorized.

Plant model (UI)

The user interface only contains external events. The button and switch generate uncontrollable external events. The plant model of the user interface model consists of the synchronous parallel compositions of the actuators:

$$\mathsf{UI} \triangleq \mathsf{UITumbleSwitch} \parallel \mathsf{UIManualButton} \parallel \mathsf{UIManualLED}$$

Tumble-switch (UITumbleSwitch) The tumble-switch can either be in the position up, down, or neutral, see Figure 5.15a. When released, the switch returns to the neutral state, as a result of its physical construction. Therefore only state `Neutral` is marked.

Manual button (UIManualButton) When the manual button is pressed, the event *uManualPushed* is emitted and a timer is set. When the timer has elapsed, a timeout event is emitted. However, when the manual button is pressed before the timer has elapsed, the event *uManualPushed* is emitted again, and the timer is set again. An infinite sequence of rapid presses of the manual button could thus lead to an infinite model. This behavior is simplified to one state where the two events are self-looped, see Figure 5.15b.

LED (UIManualLED) The LED indicates manual or motorized operation mode of the system. It can either be on, off, blinking slowly, or blinking fast. The LED is controlled by events named accordingly, see Figure 5.15c.

Control requirements (UIReq)

The requirement model for the user interface consists of the synchronous parallel composition of multiple control requirements, namely:

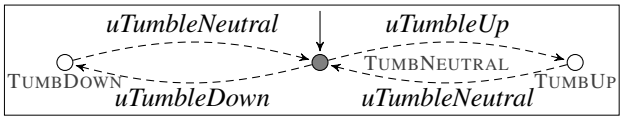
$$\mathsf{UIReq} \triangleq \mathsf{UIReqTumble} \parallel \mathsf{UIManualClutch} \parallel \mathsf{UIReqLED}$$

where

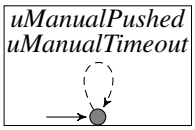
$$\mathsf{UIReqTumble} \triangleq \mathsf{UITumbleMove} \parallel \mathsf{UIHVSwitch}$$

and

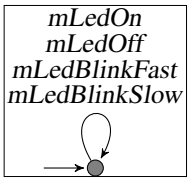
$$\mathsf{UIReqLED} \triangleq \mathsf{UILedModes} \parallel \mathsf{UILedClutch} \parallel \mathsf{UILedSequence}$$



(a) UITumbleSwitch



(b) UIManualButton



(c) UIManualLED

Figure 5.15: Model UI: user interface

Tumble move (UITumbleMove) The position of the tumble-switch determines which kind of movement of the table is allowed. When the tumble-switch is up, the table is only allowed to move up or to move horizontally into the bore. When the switch is in the down position, the table is only allowed to move down or to move horizontally out of the bore. When the tumble-switch is in its neutral position, all movement should be stopped, see Figure 5.16a.

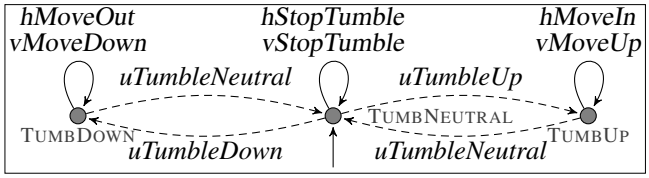
Tumble HV switch (UIHVSwitch) If the table is moving up and reaches the uppermost position, the tumble-switch must return to the neutral position before movement into the bore may begin. Similar behavior is required when moving in the opposite direction. These requirements are modeled in Figure 5.16b.

Manual clutch(UIManualClutch) Pushing the manual button results in the *uManualPushed* event, and starts a timer. As a result, the event *hClutchOn* or *hClutchOff* should be triggered, if one of these events is allowed by the other requirements. If both *hClutchOn* and *hClutchOff* are not allowed before the timer expires, the *uManualTimeout* event will be executed instead, see Figure 5.17.

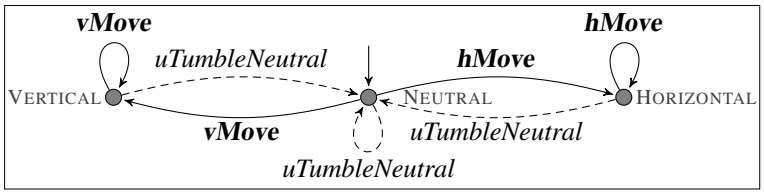
LED (UILedModes, UILedClutch) The LED indicates which operating mode is active, and whether the clutch is applied. The LED blinks if the system is in restricted mode, otherwise the LED is on or off, see Figure 5.18a. If the clutch is applied, the LED is off or blinks slow, see Figure 5.18b. If the clutch is released, the LED is on or it blinks fast. The operating modes are defined in Figure 5.18c.

5.4 Model evolvability

The goal of this section is to present the experimental results on the effect of using supervisory control theory on evolvability of the controller. The experiment involved generating a new controller in order to meet a user request for improved functionality. Note that for the actual experiment, models were used which are slightly different from, but functionally equivalent to, those presented in this chapter, similarly to the differences described earlier.



(a) UITumbleMove



(b) UIHVSwitch

Figure 5.16: Model UIReqTumble: Requirements for tumble and clutch operation

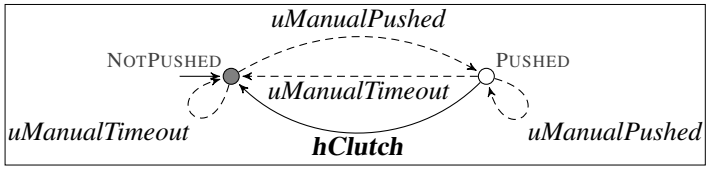
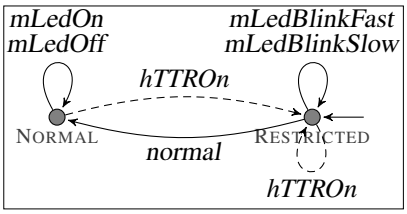
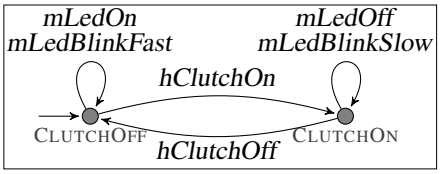


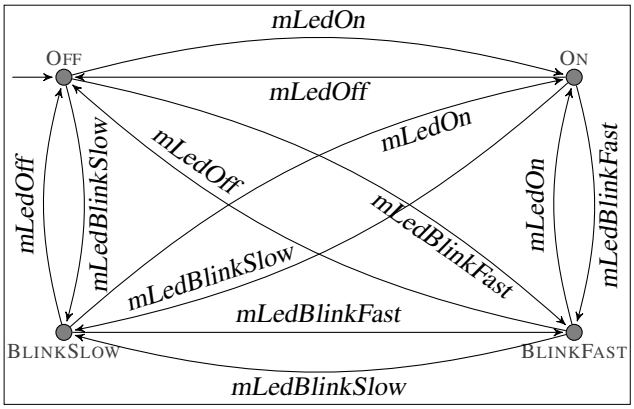
Figure 5.17: Model UIManualClutch: Requirements for manual clutch operation



(a) UILedModes

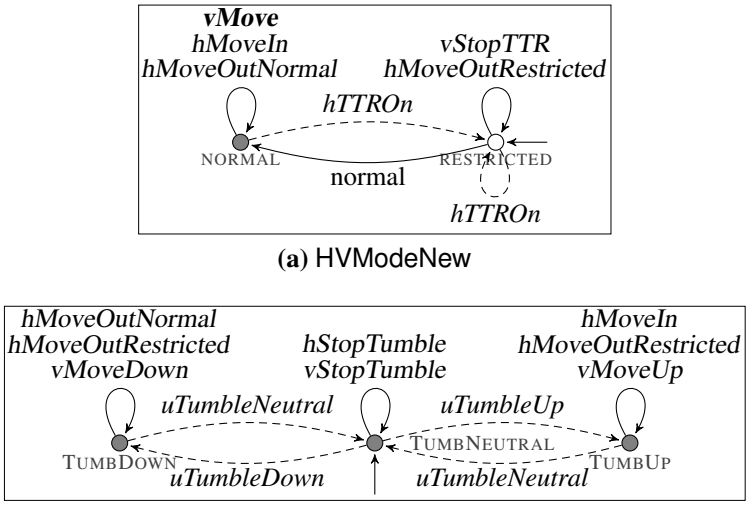


(b) UILedClutch



(c) UILedSequence

Figure 5.18: Model UIReqLED: Requirements for LED indicators



(a) HVModeNew

(b) UITumbleMoveNew

Figure 5.19: Updated requirements for evolvability case

5.4.1 Updated functionality

The control requirements presented in this chapter result in a controller that does not allow inward movement of the tabletop when the table is not maximally up. Therefore, if the user switches the tumble-switch up (which corresponds to inward or upward movement of the tabletop) in this state, the tabletop will not move. This behavior of the closed-loop system was considered to be counter-intuitive for the user. The desired new behavior in this case was, that when the tumble-switch was up, the table should move out until it reached the maximally out position, after which the controller should continue its normal operation by moving to the maximally up position, and subsequently moving into the bore. These new requirements were implemented on the actual patient support table in half a day. Implementing the same requirement using the currently used design approach was estimated to take a week. The updated models are presented below.

5.4.2 Updated control requirements

To model the changed requirements, the event *hMoveOut* is defined as an abbreviation of two events in all figures, apart from Figures 5.14a and 5.16a: $hMoveOut \triangleq \{hMoveOutNormal, hMoveOutRestricted\}$. Figure 5.14a is adapted so that the event *hMoveOutNormal* can occur only in mode Normal, whereas the event *hMoveOutRestricted* can occur only in mode Restricted, leading to Figure 5.19a. In Figure 5.16a, the event *hMoveOut* is replaced by *hMoveOutNormal*, and the event *hMoveOutRestricted* is added to the self-loops in states `TumbleDown` and `TumbleUp`, leading to Figure 5.19b. In this way, operating the tumble switch in either direction in Restricted mode always leads to the only allowed safe movement: out of the bore.

5.5 Generation, implementation and validation of the control software

The plant models and control requirements were modeled using the SCIDE (Supervisory Control Integrated Development Environment) tool set; see [68]. Subsequently, in SCIDE, a corresponding supervisor was generated using the SuSyNA (Supervisory control Synthesis of Nondeterministic Automata) tool set. This tool set is available via [66]. It contains an optimized implementation of the algorithms described in [64]. Calculation takes a few seconds on a Core 2 Duo, 3Ghz, 3Gb computer. The resulting supervisor contains 30,880 states and 264,456 transitions.

5.5.1 Validation of functional correctness

Although supervisory control theory ensures that the controller satisfies the control requirements by construction, it remains a non-trivial task (but still easier than the development of the supervisory controller itself) to define the correct plant and requirement models. Thus, errors or undesired behavior may still be present in the plant models and/or requirement models. To help validate the controlled system, the framework of [59] is used. This framework is based on the model-based engineering paradigm, where models are the primary artifacts in the design process. Below a brief overview of the framework is given, for details see [59]. Remark that the framework is a general one and it has already been applied to other case-studies.

The design process of [59] consists of the following steps:

A) Modeling of the uncontrolled plant and control requirements.

B) Synthesis of the supervisory controller using the models from step A, resulting in a model of the supervisor.

C) Simulation (un-timed) of the parallel composition of the plant models from step A, and the supervisor obtained in step B.

D) More detailed, e.g. timed or hybrid, modeling of the plant models.

E) Simulation (timed or hybrid) of the model of the closed-loop system which is obtained by combining the plant model from step D with the supervisor from step B.

F) Real-time simulation involving the actual plant hardware (the uncontrolled system) coupled with the model of the supervisor obtained in step B.

G) Code generation from the supervisor model obtained in step B.

H) Real-time control of the plant hardware controlled by the realization of the supervisor obtained by step G.

The framework consists of:

- Modeling environments to support the design steps A and D.
- Transformation tools to transform:
    1. The models of step A to input models of the synthesis tools used in step B. This transformation amounts to transforming the model represented in one modelling formalism to an equivalent model in another modelling formalism. This transformation is a purely syntactic one.
    2. The models of steps A, B, and D to input models of the simulation tools used in steps C and E.
- An infrastructure to couple models and realizations of components for step F.
- Code generators to generate code (step G) from the supervisor model obtained in step B.

The transformation and simulation tools that are described in the preceding steps are all based on the Compositional Interchange Format (CIF, see [51, 69, 1]), which has been developed in several European projects, in particular in the HYCON and HYCON2 networks of excellence [32, 31], and in the FP7 Multiform project [50]. Steps C, E, and H are described in more detail in the sections below.

5.5.2 Untimed simulation

Although the supervisor obtained using the SCT framework is guaranteed to meet the formalized control requirements, this does not yet mean that the plant will meet the initial (informal) control requirements. This can happen due to inadequate modelling of the plant or control requirements. Hence, it is still necessary to validate the correctness of the synthesized controller. In case of an incorrect controller, one immediately knows that the mistake must be in the model of the uncontrolled plant or in the control requirements.

To validate the synthesized controller, the state space of the model of the *controlled system* is explored by hand, using user guided simulation. That is, based on different scenarios of occurrence of events, events are chosen manually and executed. Scenarios are sequences of external events, which the environment is expected to generate. The scenarios are determined manually, based on the domain knowledge. It is stressed that scenarios are used to *test* the correctness of the controller, not to *design* the controller.

5.5.3 Hybrid simulation

To validate the dynamic behavior of the plant controlled by the synthesized supervisor, the CIF model of the supervisor is simulated together with (using synchronizing parallel composition) a more detailed, hybrid model of the plant. During simulation, the reaction of the model to various sequences of *external events* (i.e. events generated by the environment, such as error conditions, operator actions, etc.) is

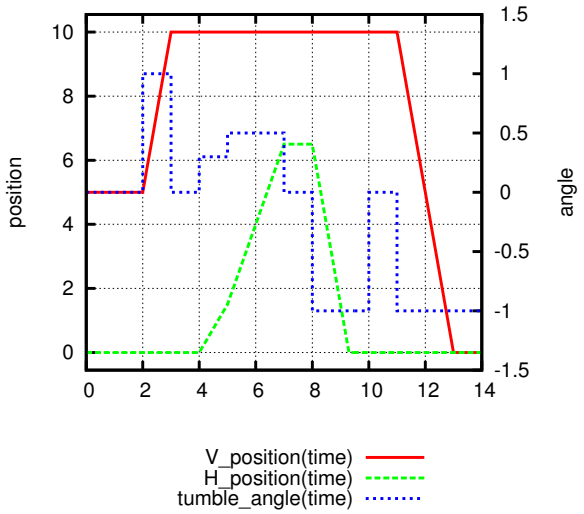


Figure 5.20: Simulation results tabletop movement controlled by tumble-switch.

evaluated. The models of the plant and the various external events are specified in CIF, modeling both discrete-event and continuous-time behavior.

In order to facilitate timed simulation of the closed-loop system, the timing of the events generated by the supervisor should be presented. Note that in SCT framework, the supervisor is assumed to be untimed and in fact it is assumed that it enables/disables events, rather than generating them. Hence SCT does not tell us how to interpret the supervisor's behavior in a timed setting.

Supervisors in a timed environment are interpreted as follows. The events present in the discrete-event plant model are taken to be *urgent*, see [75], in the hybrid model. That is, first all the events which are enabled by the supervisor and which can be executed immediately from the current state of the plant are executed. During the execution of these events, time is not allowed to progress. When there are no more events which are both enabled and can be executed immediately, then time is allowed to progress again.

Figure 5.20 shows the simulation results as a function of time of a representative case where the horizontal and vertical movement of the table-top is controlled by the tumble-switch. The position of the tumble-switch ranges from -1 to + 1. When the tumble-switch is released, its position is 0. Initially, the table is halfway up, and the tabletop is placed onto the table at the maximally out position. The tumble-switch is used to move the table to the upper position. When the table reaches the upper position at time 3, the table stops, and the tumble-switch is released. Then the table is moved inward, first slowly, then faster. After that, at time 7, the movement is stopped. Then, the table is moved out , until the table reaches the maximally out position, where it stops at time 9. The tumble-switch is momentarily released (for 1 time-unit), causing downward table movement until the table reaches its lowest position.

Apart from output of the model variables in a graph as a function of time, as shown in Figure 5.20, the CIF simulator also supports real-time, interactive, simula-

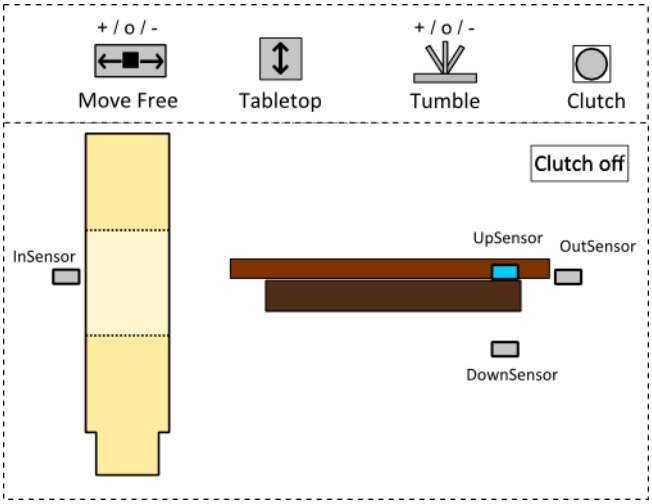


Figure 5.21: SVG image for real-time, interactive, simulation and animation.

tion and animation, based on user supplied images of the system in the standardized SVG (Scalable Vector Graphics) format [77, 27], as shown in Figure 5.21. To ensure that the simulation model can be used in its original form, without the need for user defined animation statements, a user-supplied 'hooks file' [27] defines connections between the dynamic state of the simulation model (e.g. the value of a differential variable that models the position of the table top) and attributes of objects (e.g. the graphical position of the table top) in the user supplied SVG drawing. User interaction with the simulation by means of, for example, pressing buttons in the animation, is interfaced with the simulation by means of execution of events named in the hooks file. More information on 'hooks files', including their format, can be found in [27].

5.5.4 Real-time implementation

When implementing supervisors for actual real-time control, three issues need to be dealt with:

1. The SCT framework is untimed. Implementations, however, are timed. This issue was first discussed in detail by [4].

2. Synthesis generates a set of maximally permissive supervisors, so that for an implementation, choices need to be made. Furthermore, in the original supervisory control theory, events originate from the plant, and can be disabled or enabled by the supervisor. Real-time controllers, however, actually generate events. To support synthesis of controllers, as opposed to supervisors, [24] proposes forced events, [18] proposes a subset of controllable events that are initiated by the controller, and [29] proposes directed controllers, that select at most one controllable event to be enabled at any instant.

An overview of several of the issues discussed above, in the context of PLC-based supervisory controller implementations, is given by [19].

In the actual real-time implementation of the patient support system, the sensors and actuators are connected to an industrial grade control unit. This control unit is connected to a standard PC by means of an IEEE 1394 FireWire high speed serial bus interface. The control unit conditions the sensor signals, and takes care of motion and I/O control. Furthermore, it translates sensor state changes to (uncontrollable) events and it executes the high-level commands (controllable events) generated by the PC. On the PC, the events from the control unit are buffered in an input event queue, and handled by an event handler. After executing an event from the input event queue, the state of the supervisor is updated. If the event queue is empty, the set of controllable events that is allowed by the supervisor in the current state is calculated. From this set, an event is selected and sent to the control unit for execution. In this way, nondeterminism is resolved by giving priority to uncontrollable events over controllable events. This strategy is based on the assumption that the uncontrollable events are properly conditioned by the control unit, ensuring a minimum time interval between successive uncontrollable events from each component. Switches would need to be properly debounced, and switching the tumble switch from the up position, via the neutral position to the down position, for instance, would mean generation of a sequence of the *uTumbleNeutral* and *uTumbleDown* events separated by a minimum time interval, giving the supervisor time to empty the input queue after receipt of the *uTumbleNeutral* event, and to subsequently execute the *hStopTumble* or *vStopTumble* event. Considering various other strategies for resolving nondeterminism is the subject of future research.

The real-time implementation was extensively tested on an actual patient support system: several operating procedures, that are used in practice, were carried out. In addition, attempts were made to generate erroneous behavior by means of very rapid pressing of buttons and switches, and by intentionally giving illegal commands. In all cases, the control system reacted as desired. The system was also operated and tested by Philips employees, but no errors were found. Notice, however, that the control system specifications presented in this chapter do not deal with error detection and error handling, such as broken sensors, or broken actuators. The patient support table that was controlled in the experiments was not taken into actual production, because the decision was taken to employ more ‘off the shelf’ components in the patient support system, such as motors complete with servo feedback control.

5.6 Concluding remarks

This chapter reports on a successful industrial application of supervisory control theory (SCT) to synthesize a supervisory controller for real-time control of a patient support system of an MRI scanner. The use of SCT and tools enables easy adaptation to changing control requirements. In the case of such a change, only the new requirements need to be formally defined. After the formalization step is completed, the theory and tools provided by the supervisory control framework allow automatic

generation of suitable control software.

One way to improve evolvability of the control system is by using controller *synthesis* instead of coding controllers by hand. It is also shown that evolvability can be improved by dividing the control requirements into small, independent, specifications that are modeled by means of finite state automata. In this way, errors can be detected, and attributed to specific parts of the specification. The improved evolvability has been demonstrated by means of a redesign of the control requirements—to incorporate a user change request—followed by automatic control code generation and real-time implementation.

Key aspects of the proposed method that facilitate modeling of control requirements by means of small, independent automata is a) the separation of models into actuator models, sensor models, and models that specify different structural restrictions; b) splitting of events, such as stop events, into a number of independent sub events; and c) introduction of internal events, in this case the '*normal*' event, to allow modeling of different modes of operation.

Using the supervisory control framework has indeed enabled us to focus on specifying and debugging the control requirements and plant models, instead of coding and debugging the control code. However, three issues in the event-based supervisory control framework have been found that need to be addressed:

1. **Duplication of information**. In the event-based supervisory control framework, control requirements and plant models interact exclusively by means of events: control requirements cannot refer to the state of a plant model. To refer to the current state of an automaton of the plant, that state must be reconstructed in the control requirement.

   As an example, consider the control requirement that specifies that the horizontal outward movement of the table needs to be stopped when the table is maximally out and the motor is moving outward. This requirement is specified by the self loop *hStopOut* in the two automata HStopOut and HStopInOut in Figures 5.12a and 5.12c. To reconstruct the required states of the out sensor and the motor in Figures 5.12a and 5.12c, respectively, these automata are copies (apart from the self loops) of the automata of the plant model of the sensor and the motor in Figures 5.9a and 5.11c, respectively.

2. **Limited initialization**. In the event-based supervisory control framework, each automaton has a single initial state. This means that the model of the uncontrolled plant also has a single initial state. In practice, however, controlled systems can usually be in different initial states when the system is switched on. As a result, special initialization sequences are required to ensure consistency of the state of the controlled system with the state of the supervisor.

3. **Limited debugging tool support for supervisory control specifications**. In the process of developing the appropriate specifications of the control requirements and the plant model, inevitably errors can be made. It was experienced, that in many cases, errors in specifications led to supervisors where part, or

all of the desired state space was unreachable. In fact, the case of an empty supervisor was not uncommon. This can be caused by conflicting control requirements. The problem with this behavior is that current control synthesis algorithms and tool implementations do not provide any feedback on the reason why parts of the state space have been removed, making it difficult to find the cause of undesired state space reductions. This is in fact a difficult issue, since the synthesis algorithm has no way of knowing the difference between desired state space reductions to ensure safety, and undesired state space reductions due to erroneous specifications.

In the next two chapters, state-based supervisory control synthesis will be discussed, in an attempt to deal with the three issues raised above.

Chapter 6

State-based SCS of the support table

Background The main three issues in the event-based supervisory control framework that were found in the previous chapter, are first, duplication of information; second, limited initialization in the form of a single initial state; and third, limited tool support for debugging of supervisory control specifications. In an attempt to alleviate these three issues, this chapter discusses development of a supervisory controller for the patient support system, using so-called 'state-based supervisory control synthesis'. This term is somewhat misleading, since it does not *replace* event-based synthesis by state-based synthesis, but rather *extends* it, by allowing direct references to the state of a plant model in control system requirements, and by allowing multiple initial states. Please note that a subset of the functionality of the patient support systems is discussed in this chapter. The emergency system and the light-visor with automated positioning, are not modeled.

Contribution An overview of the main contributions of the chapter follows below. The main contributions are elaborated in more detail in the 'Concluding remarks' in Section 6.5. As state-based supervisory control synthesis tool the open source STSlib tool [43] was used. After developing the control requirements for the patient support case in the syntax allowed by the tool, this syntax appeared to be too restrictive, and too error-prone. This lead to the desire for generalized state-based control requirements, as discussed in Section 6.1. The resulting publication was the combined effort of several of people, and the implementation was done by Dennis Hendriks.

The subsequent redesign based on generalized state-based control requirements, as presented in this chapter, is indeed much more readable and intuitive. The multiple initial states that are allowed by the synthesis algorithm are essential. In fact, any arbitrary safe combination of sensors is allowed as initial state. This allows the actual patient support system to be started in any, arbitrary, safe position.

Finally, the state-based control requirements are developed in a modular way,

as in the event-based case, by dividing the requirements into small, relatively independent components. The same set of experiments, involving simulation, validation, and real-time control as described in the previous Chapter 5, are executed with the state-based supervisory control models. Also in this case, no errors could be detected in the final controlled patient support system, after extensive testing. The limited tool support for debugging of supervisory control specifications was experienced as considerably less troublesome in the case of the state-based specifications as described in the sequel, since the state-based control requirements could be specified, understood, and debugged more easily.

Outline The outline of the chapter is as follows. Section 6.1 presents the required new language concepts for supervisory control specifications based on automata and predicates. Sections 6.2 and 6.3 describe the plant models and control requirements, respectively, of the patient support system. Finally, the required toolchain is described in Section 6.4, and concluding remarks are presented in Section 6.5.

6.1 Supervisory control specification using automata and predicates

In event-based supervisory control as defined in [79, 12, 62, 64], see Chapter 3, the plant and the control requirements are both modeled by finite-state automata. The goal of these models is to capture the dynamic behavior of the uncontrolled system and safe behavior for the controlled system, respectively. Both behaviors are defined as languages, e.g. sequences of events. Automata are used to represent the languages compactly.

In [3, 2], event-based supervisory control is defined in a process algebraic framework. This framework allows specification of event-based models completely without states. It uses the successful termination concept as the equivalent of marked states in automata.

In nonblocking supervisory control of state tree structures, see [45], plant specifications are also modeled by means of automata. However, unlike in event-based supervisory control, multiple initial locations (states) are allowed, and the states of the automata can be referred to in the control requirements. The control requirement specifications are specified as predicates over states and events in two forms: mutual state exclusion and state-event exclusion. The accompanying open source STSlib tool [43] also allows specification of control requirements as automata, by rewriting them as plant automata in combination with control requirements in the form of state-event exclusion.

The allowed input form of the control requirements as implemented in the STSlib tool is quite restricted. Mutual state exclusion requirements are each defined as the negation of a conjunction of location references, where a location reference is the name of a location of an automaton. The predicate over the state in a state-transition exclusion requirement is defined as a conjunction of location references.

To allow a more intuitive way of defining control requirements, [48] proposes

Table 6.1: Syntax of generalized predicates.

$\mathbf{S}$	invariant
$\mathbf{S} \implies \neg\mathbf{E}$	state-event exclusion
$\mathbf{E} \implies \mathbf{S}$	state-event exclusion

generalized state-based control requirements. Generalized state-based requirements extend the input format of state-based supervisory control synthesis by allowing general propositional logic in the mutual state exclusion requirements and in state-transition exclusion requirements. This is done by symbolic preprocessing of the requirements to obtain the form required for the STSlib tool.

Table 6.1 presents the allowed syntax of the generalized state-based predicates. In this table, $\mathbf{S}$ is a predicate over the locations of an automaton, and $\mathbf{E}$ is a set of events. The invariant $\mathbf{S}$ defines the predicate $\neg\mathbf{S}$ as mutual state exclusion predicate. The predicates $\mathbf{S} \implies \neg\mathbf{E}$ and $\mathbf{E} \implies \mathbf{S}$ are both state-event exclusion predicates. The latter form is defined in terms of the first, more primitive form, as $\neg\mathbf{S} \implies \neg\mathbf{E}$.

Finally, to allow the use of mutual state exclusion and state-event exclusion specifications also in plant models, [21] defines a transformation that eliminates these predicates in plant models in terms of a parallel composition of automata.

6.1.1 Plant model

In the state-based supervisory control framework, the goal of the plant model is twofold:

First, the plant model should capture the relevant states of the uncontrolled system, that is the states required for intuitive control requirement specifications. Such states can refer, for instance, to the state of a sensor that can be on and off, and a clutch that can be applied or released. The states of the plant model are used to define control requirements and to implement a controller (supervisor).

Second, the plant model should capture the dynamic behavior of the uncontrolled system. That is, the model should contain the response of the system to the events. It should model in which states an event can occur, and what the new state of the system is after an event occurred. For example, a sensor can only emit the event *sensor_on* if the sensor is in the state off, and after the occurrence of this event, the state of the sensor is on. As in the event-based framework, states can be defined as marked, meaning that these states must be reachable by the supervised plant. The dynamic behavior is used in combination with the control requirements to synthesize the supervisor, in other words, the dynamic behavior is used to determine how the system must be controlled to meet the control requirements.

The plant models are divided into components, such as the vertical and horizontal axis. These components are divided into subcomponents and ultimately into the smallest subcomponents that are physically relevant for control: the actuators, sensors and structure. Actuators change the state of the system, sensors detect the state of the system, and the structure of the system defines the interaction between the actuators

and sensors.

Component models consist of automata and predicates over locations of the automata and events. The sensors and actuators have an observable state. Therefore, the sensors and actuators are modeled by automata. The structure, that defines the relation between the actuators and the sensors, poses constraints on the combinations of the states of the sensors and actuators. The structure is usually stateless, that is, the set of states of the structure is a subset of the set of states of the sensors and actuators. Therefore, it is modeled by predicates over the models of the sensors and actuators.

6.1.2 Control requirements

Control requirements can be defined in terms of predicates or automata. There are two kinds of these automata:

- Automata that define additional states. These states are referred to as internal states. An example is the definition of the states `Normal` and `Restricted` in automaton HVMode in Figure 6.8.
- Automata that define sequences of events as control requirements. The states that are introduced by the automata are not referred to in other specifications. An example is the automaton UIHVSwitch of Figure 6.10 that defines the horizontal-vertical switching requirement.

6.2 Plant models

The goal of this section is to describe the formal models of the plant and of the control requirements. In the models, states are denoted by vertices, initial states are indicated by an unconnected incoming arrow, and marked states are denoted by filled vertices. Controllable and uncontrollable events are depicted by solid and dashed edges, respectively. Multiple events on an edge represent an edge for each event. A bold event name indicates a set of events, representing an edge for each event in the set.

6.2.1 Vertical axis

The table can move up and down along the vertical axis. The vertical axis contains two end sensors, maximally up and maximally down, and one actuator for the vertical motor drive. The motor can move the table up and down.

The plant model of the vertical axis consists of the automaton models of the vertical motor drive and the up and down sensors, that are modeled in the paragraphs ‘Motor drive’ and ‘Sensors’ below. The relations between the sensors and the motor drive, which are imposed by the physical structure of the system, are modeled by predicates over the states and events of the sensor and motor drive models.

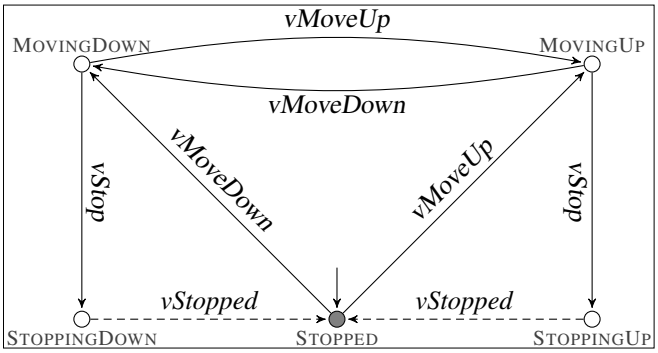
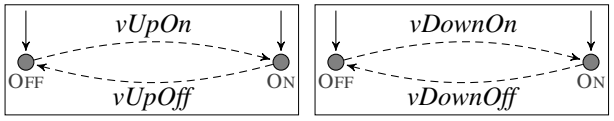


Figure 6.1: Model VMotor of the vertical motor.



(a) VUpSensor **(b)** VDownSensor

Figure 6.2: Models of the vertical sensors.

Motor drive (VMotor) The motor is controlled by a resource controller, which controls the brakes, and the motion feedback control loop. The model of the motor drive only includes the behavior exposed to the supervisor, see Figure 6.1. It is assumed that initially the motor is `Stopped`. Movement is started via events *vMoveUp* and *vMoveDown*. Switching between the movement directions is assumed to be instantaneously. If the motor is moving and the *vStop* event is triggered, the motor slows down to come to a halt. The motor switches to the corresponding location `StoppingUp` or `StoppingDown`. When the motor has come to a halt, the event *vStopped* is emitted, and the motor enters location `Stopped` again.

Sensors (VUpSensor, VDownSensor) The maximally up and maximally down sensors are active if the table is at the sensor position, otherwise the sensors are inactive. This is modeled by means of automata with two locations, namely `On` and `Off`, see Figure 6.2. When they become active or inactive, the sensors emit the uncontrollable events *vUpOn* (*vDownOn*) or *vUpOff* (*vDownOff*), respectively. When the system is started, the table can be in any position. Therefore, each of the locations of the sensors can be the initial location of the sensor.

Component relations The two sensors are never active at the same time, as a result of their physical location. This is modeled by the state exclusion predicate (6.1).

$$\neg(\mathsf{VUpSensor}.\texttt{On} \wedge \mathsf{VDownSensor}.\texttt{On}) \tag{6.1}$$

The vertical sensors only change state when the table moves vertically. Only when the motor drive is moving the table up, the maximally down sensor can turn off (6.2), and the maximally up sensor can turn on, and likewise for the opposite direction (6.3).

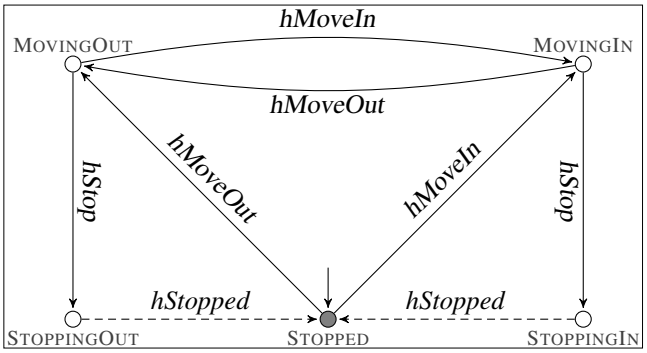
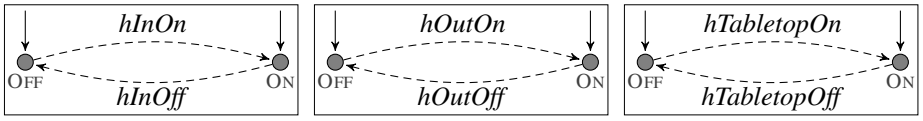


Figure 6.3: Model HMotor of the horizontal motor.



(a) HInSensor **(b)** HOutSensor **(c)** HTabletopSensor

Figure 6.4: Models of the horizontal sensors.

$$\{vDownOff, vUpOn\} \implies \text{VMotor}.\texttt{MovingUp} \vee \text{VMotor}.\texttt{StoppingUp} \tag{6.2}$$

$$\{vDownOn, vUpOff\} \implies \text{VMotor}.\texttt{MovingDown} \vee \text{VMotor}.\texttt{StoppingDown} \tag{6.3}$$

6.2.2 Horizontal axis

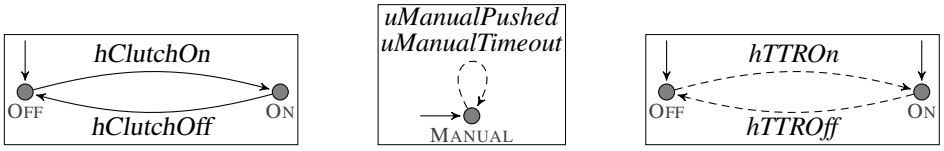
The horizontal axis consists of a removable tabletop on top of the main table. The tabletop can be moved in and out of the bore. The plant model of the horizontal axis consists of the automata models of the horizontal motor drive, the clutch, the maximally in and out sensors, the table top sensor, and the table top release switch.

Motor drive (HMotor) The horizontal motor drive is similar to the vertical motor drive, see Figure 6.3.

Sensors Figure 6.4 presents the models of the sensors of the horizontal axis. The sensors HInSensor, HOutSensor and HTabletopSensor indicate the position of the tabletop, and its presence, respectively. Initially, the tabletop can be in any position. Therefore, any location can be the initial location of the sensors.

Clutch, manual button and table top release switch The models of the clutch (HClutch), manual button (UIManualButton) and table top release switch (HTTRSwitch) are given in Figures 6.5a, 6.5b, and 6.5c, respectively.

The operator can request to toggle the state of the clutch by pressing the manual



(a) HClutch **(b)** UIManualButton **(c)** HTTRSwitch

Figure 6.5: Models of the clutch, manual button, and TTR switch.

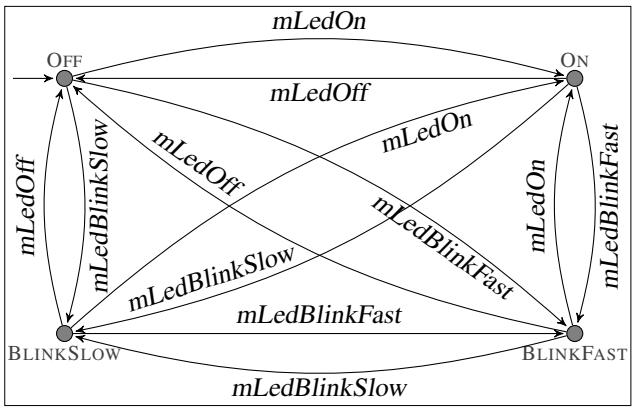


Figure 6.6: Model UIManualLED of the user-interface LED.

button. This causes occurrence of the *uManualPushed* event, and setting of a timer. When the timer has elapsed, a timeout event is emitted. The control system changes the state of the clutch only when it is safe to do so, and only when the timer has not yet expired. However, when the manual button is pressed again before the timer has elapsed, the event *uManualPushed* is emitted again, and the timer is set again. An infinite sequence of rapid presses of the manual button could thus lead to an infinite model. This behavior is simplified to one location where the two events are self-looped, see Figure 6.5b.

LED (UIManualLED) The LED indicates manual or motorized operation mode of the system. It can either be on, off, blinking slowly, or blinking fast. A more detailed explanation of the meaning of the LED indication is given in the section on the control requirements.

Component relations The two end-sensors cannot be active at the same time (6.4), as a result of their physical location. The tabletop can only be added and removed in the maximally out position. When the tabletop is not present, the maximally out sensor remains active. This results in the relation between the table top sensor and horizontally out sensor of (6.5).

$$\neg(\mathsf{HInSensor}.\texttt{On} \wedge \mathsf{HOutSensor}.\texttt{On}) \tag{6.4}$$

$$\mathsf{HTabletopSensor}.\texttt{Off} \implies \mathsf{HOutSensor}.\texttt{On} \tag{6.5}$$

The maximally in and out sensors change state only when the tabletop is present

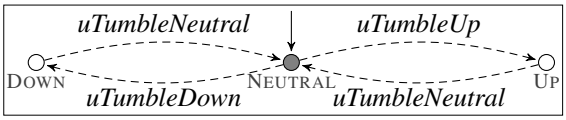


Figure 6.7: Model UITumbleSwitch of the tumble-switch.

and is moving horizontally. The tabletop can move horizontally in three distinct cases:

- The clutch is released; the table can be moved by hand, therefore the sensors can always switch in any order. Note that the state of the clutch is defined by the control system.
- The TTR switch is activated; the table can be moved by hand, as in the case that the clutch is released. Note that, the state of the TTR switch is defined by the operator.
- If the clutch is applied, and the TTR switch is not active, the movement is controlled by the motor, analogous to the vertical axis.

These restrictions on the plant behavior of the maximally in and out sensors are specified by the following two state-event exclusions:

$$\{hOutOn, hInOff\} \implies \begin{array}{l} \mathsf{HClutch}.\texttt{Off} \vee \\ \mathsf{HTTRSwitch}.\texttt{On} \vee \\ \mathsf{HMotor}.\texttt{MovingOut} \vee \mathsf{HMotor}.\texttt{StoppingOut} \end{array} \tag{6.6}$$

$$\{hOutOff, hInOn\} \implies \begin{array}{l} \mathsf{HClutch}.\texttt{Off} \vee \\ \mathsf{HTTRSwitch}.\texttt{On} \vee \\ \mathsf{HMotor}.\texttt{MovingIn} \vee \mathsf{HMotor}.\texttt{StoppingIn} \end{array} \tag{6.7}$$

6.2.3 Tumble switch

Figure 6.7 shows the model of the tumble switch. The tumble switch can either be in the position up, down, or neutral. When released, the switch returns to the neutral state, as a result of its physical construction. Therefore, only location `Neutral` is initial and marked.

6.3 Control requirements

6.3.1 Normal and restricted operating modes

The table can collide with the magnet when moving inward if the table is not maximally up. The table can also be damaged when moving downward if it is not maximally out. These situations should be prevented. Therefore, either the

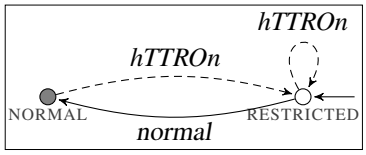


Figure 6.8: Model HVMode defining the Normal and Restricted control modes

maximally out sensor or the maximally up sensor must be on, unless the TTR switch is on. If the TTR switch is on, the table can be moved freely by the operator. In such a case, the control system cannot prevent the potentially unsafe situation in which both sensors are off.

To facilitate specification of the requirements for the interaction between the horizontal and the vertical axis two control modes are introduced, namely, 1) control after TTR is activated (`Restricted`), and 2) control after the normal event has occurred (`Normal`), as shown in Figure 6.8. Detection of the occurrence of a potentially unsafe event (*hTTROn*) is decoupled from the response to this situation. The detection is modeled in the automaton. The appropriate response is modeled in the control requirements.

When the system is in mode `Normal` it must be ensured that the system is either maximally up, or maximally out, and that the TTR switch is off, see control requirement (6.8). This prevents the table from colliding with the magnet. The requirements also ensures that the normal mode is only entered if it can be ensured that the system stays within the safe states (maximally up or maximally out).

$$\mathsf{HVMode}.\texttt{Normal} \implies \mathsf{HTTRSwitch}.\texttt{Off} \wedge (\mathsf{VUpSensor}.\texttt{On} \vee \mathsf{HOutSensor}.\texttt{On}) \tag{6.8}$$

6.3.2 Vertical axis control

Maximally up and down Although the upper most and lower most position must be reachable, movement beyond the maximally up and maximally down position is not allowed. This implies that initiating movement in the upper direction is only allowed when the table is not maximally up (6.9). Likewise it is only allowed to initiate movement in the lower direction when the table is not maximally down (6.10). Furthermore, movement must be stopped if and only if either the table is moving up (or down) and the table is maximally up (or down), or the tumble switch is in the neutral position, or the control mode is HVMode.`Restricted` (6.11).

$$\{vMoveUp\} \implies \begin{array}{l} \mathsf{VUpSensor}.\texttt{Off} \wedge \\ \mathsf{UITumbleSwitch}.\texttt{Up} \wedge \\ \mathsf{HVMode}.\texttt{Normal} \end{array} \tag{6.9}$$

$$\{vMoveDown\} \implies \begin{array}{l} \mathsf{VDownSensor}.\texttt{Off} \wedge \\ \mathsf{UITumbleSwitch}.\texttt{Down} \wedge \\ \mathsf{HVMode}.\texttt{Normal} \end{array} \tag{6.10}$$

$$\{vStop\} \implies \begin{array}{l} (\mathsf{VUpSensor}.\texttt{On} \wedge \mathsf{VMotor}.\texttt{MovingUp}) \vee \\ (\mathsf{VDownSensor}.\texttt{On} \wedge \mathsf{VMotor}.\texttt{MovingDown}) \vee \\ \mathsf{UITumbleSwitch}.\texttt{Neutral} \vee \\ \mathsf{HVMode}.\texttt{Restricted} \end{array} \tag{6.11}$$

6.3.3 Horizontal axis control

Motor control

The horizontal axis should never move beyond the maximally in and maximally out positions. Furthermore, the horizontal axis may not move, when the tabletop is not present, when the clutch is released or when the TTR switch is active.

$$\{hMoveIn\} \implies \begin{array}{l} \mathsf{HInSensor}.\texttt{Off} \wedge \\ \mathsf{UITumbleSwitch}.\texttt{Up} \wedge \\ \mathsf{HTabletopSensor}.\texttt{On} \wedge \\ \mathsf{HVMode}.\texttt{Normal} \end{array} \tag{6.12}$$

$$\{hMoveOut\} \implies \begin{array}{l} \mathsf{HOutSensor}.\texttt{Off} \wedge \\ \mathsf{UITumbleSwitch}.\texttt{Down} \wedge \\ \mathsf{HTabletopSensor}.\texttt{On} \end{array} \tag{6.13}$$

$$\{hStop\} \implies \begin{array}{l} (\mathsf{HInSensor}.\texttt{On} \wedge \mathsf{HMotor}.\texttt{MovingIn}) \vee \\ (\mathsf{HOutSensor}.\texttt{On} \wedge \mathsf{HMotor}.\texttt{MovingOut}) \vee \\ \mathsf{UITumbleSwitch}.\texttt{Neutral} \vee \\ \mathsf{HTabletopSensor}.\texttt{Off} \vee \\ \mathsf{HTTRSwitch}.\texttt{On} \end{array} \tag{6.14}$$

The motor should not be moving when the tabletop is not present. However, it is not possible to prevent the table from moving horizontally without tabletop, because the tabletop can be removed by the operator while the table is moving horizontally. In other words, the supervisor cannot ensure that the table is not moving when the operator removes the tabletop. To model this requirement, initiation of horizontal movement is allowed only when the tabletop is present (6.12), 6.13. In addition, horizontal movement must be stopped when the tabletop is not present (6.14).

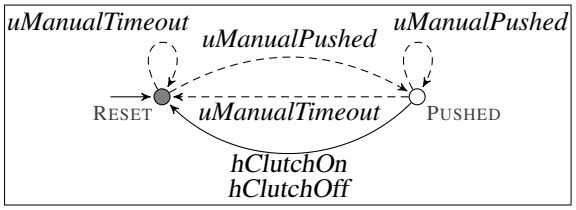


Figure 6.9: Model UIManualClutch of the manual button modes.

Clutch control

The manual button toggles the table between manual and motorized mode (if allowed). In the motorized mode, the position of the tumble switch determines the movement of the table. In the manual mode, the operator can move the tabletop by hand. When the button is pushed, the clutch is released (or applied) to switch the table to manual mode (or motorized mode). Therefore, this button is called the 'manual button'. Note that in manual mode, the supervisor can still prevent the table from performing operations requested by the user, such as moving the table motorized.

Clutch motor interaction The tabletop may only be moved by the horizontal motor if the clutch is applied:

$$\neg \mathsf{HMotor}.\texttt{Stopped} \implies \texttt{HClutch.On} \tag{6.15}$$

Manual operation of the clutch (UIManualClutch) Figure 6.9 shows that pushing the manual button should trigger an event *hClutchOn* or *hClutchOff*, if one of these events is allowed by the other requirements. If both *hClutchOn* and *hClutchOff* are not allowed, the push event is ignored when a timeout occurs.

Clutch and TTR switch interaction Clutch commands may not be issued when the TTR switch is on:

$$\{hClutchOn, hClutchOff\} \implies \mathsf{HTTRSwitch}.\texttt{Off} \tag{6.16}$$

LED control

The LED indicates which operation mode is active, and whether the clutch is applied. The LED blinks if the system is in restricted mode. In normal mode, the LED is on or off. If the clutch is applied, the LED is off or blinks slow. If the clutch is released, the LED is on or it blinks fast.

$$\{mLedOn\} \implies \mathsf{HVMode}.\texttt{Normal} \wedge \texttt{HClutch.Off} \tag{6.17}$$
$$\{mLedOff\} \implies \mathsf{HVMode}.\texttt{Normal} \wedge \texttt{HClutch.On} \tag{6.18}$$
$$\{mLedBlinkSlow\} \implies \mathsf{HVMode}.\texttt{Restricted} \wedge \texttt{HClutch.On} \tag{6.19}$$
$$\{mLedBlinkFast\} \implies \mathsf{HVMode}.\texttt{Restricted} \wedge \texttt{HClutch.Off} \tag{6.20}$$

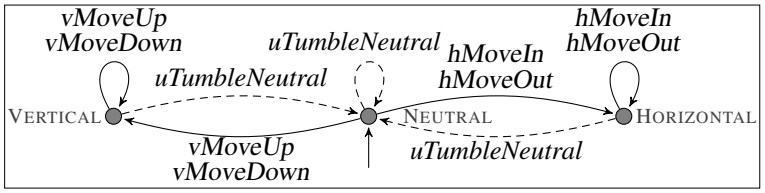


Figure 6.10: Model UIHVSwitch of the horizontal-vertical switching requirement

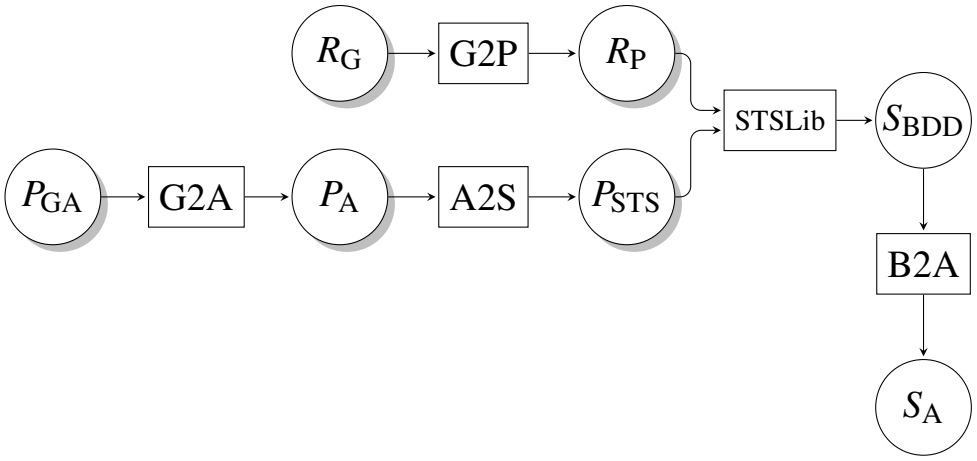


Figure 6.11: State-based tool chain, where P = Plant, R = Requirement, S = Supervisor, and the subscripts A = Automata, G = Generalized predicates, GA = Automata with Generalized predicates, P = Predicates, STS = State Tree Structures, BDD = Binary Decision Diagram.

6.3.4 Tumble switch horizontal-vertical mode (UIHVSwitch)

If the table is moving up and reaches the upper most position, the tumble switch must return to the neutral position before movement into the bore may begin. Similar behavior is required when moving in the opposite direction. These requirements include the history of event occurrences, therefore they cannot be modeled by predicates. An automaton is used to model this requirement, see Figure 6.10.

6.4 Supervisory control synthesis toolchain

Figure 6.11 gives an overview of the tool chain used to synthesize the supervisor. First, the plant model P_{GA} that consists of automata, and possibly mutual state exclusion specifications and state-event exclusion specifications, is converted to a plant model P_A consisting of automata only, using the algorithm defined in [21]. Then, the plant automata and control requirement specifications are transformed into the input format for the STSLib tool [43]: the generalized control requirement predicates R_G are transformed to the negated conjunctive form R_P, as defined in [48], and the plant automata P_A are transformed to the corresponding state tree structure form P_{STS}. The latter transformation is straightforward. All automata are added as a

child to the root AND-superstate of the STS model. The automata that are part of the requirements are added to the root AND-superstate as well. These automata are marked as 'memories'. The STSLib synthesis algorithm translates any restriction on uncontrollable events made by such memory automata to predicates. In this way, the memory automata cannot constrain the uncontrolled behavior of the plant.

The supervisor S_{BDD} is synthesized using the STSLib tool. It returns for each controllable event a predicate encoded as a binary decision diagram (BDD). This predicate defines when the event is enabled by the supervisor. These BDDs and their associated events are then transformed to the single location automaton S_{A}, that has a guarded self loop for each controllable event.

For a more detailed overview of the tool chain, the interested reader is referred to [67].

6.5 Concluding remarks

A straightforward comparison of the models of Chapters 5 and 6, that discuss event-based and state-based supervisory control of the patient support system, respectively, illustrates the intuitiveness and conciseness of generalized state-based control requirements. Duplication of information, that was observed in event-based control requirements, see Section 5.6, has been completely eliminated. Experience at Philips Healthcare has also shown that generalized state-based control requirements are intuitive for both domain engineers and software engineers, since they closely match the view of the systems in terms of states, transitions between states, and restrictions on allowable states and state-transitions. The most striking example of this, is the specification of the safety restrictions for horizontal and vertical interaction of the patient support table. Figure 6.12 shows the event-based specification (copied from Figure 5.14b), and Control requirement 6.21 shows the corresponding state-based specification (copied from Specification 6.8):

$$\begin{aligned} \mathsf{HVMode}.\mathtt{Normal} \implies & \mathsf{HTTRSwitch}.\mathtt{Off} \wedge \\ & (\mathsf{VUpSensor}.\mathtt{On} \vee \mathsf{HOutSensor}.\mathtt{On}) \end{aligned} \tag{6.21}$$

As a result of the more intuitive, and simplified control requirement specifications, the specifications were easier to develop, understand, and debug. The synthesized state-based supervisor was implemented for real-time control of the actual patient support system, and was tested and found to operate correctly, just as the event-based supervisor (see Section 5.5.4).

A crucial improvement of the synthesized state-based supervisor versus the event-based supervisor is that the state-based supervisor can deal with multiple initial states of the controlled system. In the event-based specifications of the plant models, for example, each sensor has one initial state only, whereas in the state-based specifications, each sensor has two initial states. As a result, the event-based plant and supervisor (see Chapter 5), have one initial state only. Therefore, the patient support table has to be moved into the specified initial state, before the event-based supervisory controller can be activated. The state-based supervisor on the other hand,

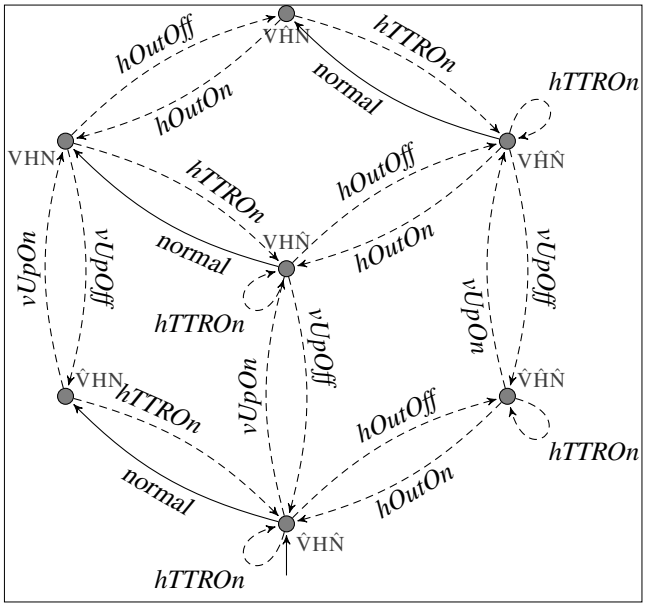


Figure 6.12: Event-based horizontal and vertical safety interaction.

can be activated immediately, independently of the initial state of the patient support system.

In principle, state-based supervisory control specifications and the associated synthesis algorithms lead to supervisors that can be represented much more efficiently than their event-based counterparts, as described in [44, 45]. This is due to the fact that state-based synthesis algorithms separate the two basic tasks of a supervisor, namely, first, to keep track of the state of the controlled plant, and second, to make control decisions based on the current state of the plant. In state-based supervisory control, the first task is executed by one or more automata, and the second tasks is implemented by means of a set of control functions that define for each state, and for each event, whether or not the event is allowed. In this way, the parallel composition of the automaton that keep track of the state of the plant can be preserved, the state-space is not expanded. In event-based supervisory control, these two tasks are integrated in the same automaton, that keeps track of the state, and at the same time implements the control functions. As a result, the supervisor that is derived in Chapter 5, which implements the same behavior as the supervisor that is derived in this chapter, contains 30,880 states. To implement the supervisor that is derived in this chapter, only 46 states need to be tracked.

The current state-based supervisory control synthesis tool chain, as defined in Figure 6.11, can, however, still lead to large and inefficient supervisors. Such blow-up in synthesized supervisors is mainly caused by the use of disjunction in control requirements, as for example in Control requirement (6.14), which is repeated below:

$$\begin{aligned}\{hStop\} \implies & (\mathsf{HInSensor}.\texttt{On} \wedge \mathsf{HMotor}.\texttt{MovingIn}) \vee \\ & (\mathsf{HOutSensor}.\texttt{On} \wedge \mathsf{HMotor}.\texttt{MovingOut}) \vee \\ & \mathsf{UITumbleSwitch}.\texttt{Neutral} \vee \\ & \mathsf{HTabletopSensor}.\texttt{Off} \vee \\ & \mathsf{HTTRSwitch}.\texttt{On}\end{aligned} \tag{6.22}$$

The use of disjunction leads to blow-up due to the transformation to the negated conjunctive form R_{P}, as defined in [48] (see Section 6.4). Blow-up can be avoided by partitioning the stop event into multiple, different stop events, as has been done in Chapter 5 on event-based supervisory control of the patient support system. Such a partitioning could also be done by an improved synthesis algorithm, such as described in Section 7.10. Alternatively, the input syntax of the synthesis tool (in this case, STSlib [43]) could be extended to deal directly with disjunction in BDD manipulations. Finally, the use of BDDs in the synthesis algorithm is always prone to the well known variable ordering problem, which is NP-complete, see [70].

As a final remark, note that although the formalism used in this chapter is state-based, states must be expressed by means of *automata*, and cannot be expressed by means of variables. This restriction is not a problem for the patient support case. For the patient communication system, as discussed in the next chapter, however, the use of variables for supervisory control synthesis is essential.

Chapter 7

State-based SCS of the communication system

Background Experience with application of supervisory control synthesis in industry, see previous chapters and [7, 46, 21], has shown that key aspects of the successful application of supervisory control synthesis in industry are the *expressiveness* and *user-friendliness* of the formalism for the specification of plant models and control requirements. Currently, requirements are *defined* by domain experts, and the corresponding control code is *implemented* by software experts. For example, the required number of pages that a printer is supposed to print, or the action to be taken when a sheet is stuck in the printer, or the functionality of the user interface for an MRI scanner are all defined by experts on the design and operation of a printer, or MRI scanner, respectively. Software experts then implement these informal requirements as executable code.

To facilitate the use of formal models of the plant and control requirements as the basis of controller synthesis and subsequent real-time code generation, the specification formalism should be expressive and intuitive enough to support definition and understanding of the plant models and control requirements by both domain experts *and* software experts. Experience has demonstrated that the ability to refer to the *state* of the controlled system is a key aspect in the definition of formal control requirements by domain experts.

Contribution The contribution of this chapter is the definition and use of a (subset of a) specification formalism for supervisory control synthesis that is both expressive and intuitive enough to be used by both domain experts and software experts. The language is shown to facilitate systematic, *state-based* and *event-based*, *compositional* specification of a control system for a patient communication system of an MRI scanner.

A modular approach is generally considered beneficial for the design of complex systems; see for example [20]. For the design of *evolvable* systems, modularity is especially valuable, since it implies decomposition of the system in a number of

smaller, relatively independent subsystems, that interact via well-defined interfaces. In this way, changes in a subsystem do not affect other subsystems, as long as the interfaces between the subsystems remain unchanged. This chapter also shows that the specification formalism can deal with both event-based and state-based interfaces.

To support systematic, modular specification of models for supervisory control synthesis, *observers* are introduced. Observers are well known from feedback control theory for continuous-time dynamical systems [36]. In this chapter, observers are shown to be extremely usefull for supervisory controller design. Observers records sequences of events in terms of states, where the states are represented by values of variables, and/or automaton locations, that are required for the specification of control requirements. For the variables, two classes are introduced: the independent and the dependent variables, where the values of the dependent variables can be defined in terms of functions of the independent variables.

The specification formalism integrates concepts from supervisory control synthesis based on state tree structures; see [44], with concepts from CIF (see [1, 74]), extended automata (see [62]), and generalized state-based control requirements from [46].

The requirements modeled in this chapter are a snapshot of the requirements for the actual communication system. At the time of writing, the system was still under development. The requirements were changing due to interaction with the customer.

A slightly modified version of this chapter was published as [72].

Outline This chapter is structured as follows. The patient communication system is described in Sections 4.3 and 7.2. Section 7.3 describes the specification formalism, and Section 7.4 discusses the control architecture used for modeling the patient communication system. The plant model, the requirements models and the supervisor model are presented in Sections 7.5, 7.6 and 7.7, respectively. Section 7.8 describes how changes from a state-based interface to an event-based interface can be readily incorporated into the control system by straightforward addition of plant components and control requirements. Sections 7.9 and 7.10 present the synthesized supervisor for the changed model and the used tool chain, respectively. Finally, Section 7.11 presents concluding remarks.

7.1 Different flavors of supervisory control theory

In centralized event-based supervisory control synthesis, the plant and control requirements are each modeled by means of a set of finite automata, synchronizing on shared event labels; see Chapters 3 and 5. The synthesis algorithm leads to a single supervisor. Although this form of supervisory control synthesis is conceptually simple, it is practically infeasible to handle large complex systems due to the exponentially high computational complexity resulting from synchronous composition of local component and requirement models. To cope with the complexity issue, many new event-based techniques have been introduced, such as interface-based hierarchical synthesis [39], and distributed aggregative synthesis [64]. For a more

complete overview, see [63]. In interface-based synthesis, the plant consists of several components located at various levels. The components interact via interfaces. The components are controlled by local supervisors that can be locally synthesized, as long as all interfaces can remain fixed during synthesis. In distributed aggregative synthesis, local supervisors are also used, but the main technique used to avoid high synthesis complexity is model abstraction. The common denominator among event-based synthesis methods is the use of automata to model the plant and control requirements. Control requirements are thus specified as sequences of events.

A different method of supervisory control synthesis proposes the use of state tree structures as the underlying model of the plant, and state-based expressions as control requirements. The STS symbolic synthesis [44] algorithm utilizes binary decision diagrams to manipulate states. It is highly efficient and results in a single, efficiently encoded, supervisor. State tree structures are a superset of automata. They enable hierarchical modeling and support efficient storage. State-based control requirements restrict the behavior of the plant by forbidding combinations of states of the parallel components. These state-based control requirements are referred to as *mutual state exclusion* requirements. There are also *state-transition exclusion* requirements, which specify that events are disabled for a given combination of states. Apart from these state-based control requirements, the method also allows so-called dynamic state feedback, in which automata are used to specify control requirements as sequences of events, by transforming such automata to a form suited for state-based supervisory control. By means of this dynamic state feedback, event-based control requirements can also be defined in the state-based supervisory control framework. However, the allowed input form of the state-based control requirements as implemented in the STSlib [43] tool is quite restricted. Mutual state exclusion requirements are defined as the negation of a conjunction of location references, where a location reference is the name of a location of an automaton. The predicate over the state in a state-transition exclusion requirement is defined as a conjunction of location references.

To allow a more intuitive way of defining control requirements, [46] proposes generalized state-based control requirements. Generalized state-based requirements extend the input format of state-based supervisory control synthesis by allowing general propositional logic in the mutual state exclusion requirements and in state-transition exclusion requirements. This is done by symbolic preprocessing of the requirements to obtain the form required for the STSlib tool. Using generalized state-based control requirements to define the control requirements for a coordinator of maintenance procedures of a high-tech printer by Océ show a considerable gain in the process of formalizing the informally specified control requirements; see [47].

Another way of extending event-based supervisory control with state-based functionality is by extending automata with variables, guards and updates. Multiple frameworks that extend automata with variables exist, such as [15, 81, 22, 62]. In [15], it is assumed that variables are updated by at most one automaton. In [81], automata extended with variables are used to implement a supervisor. In [22], it is assumed that variables are local to each automaton. In [62], variables are global

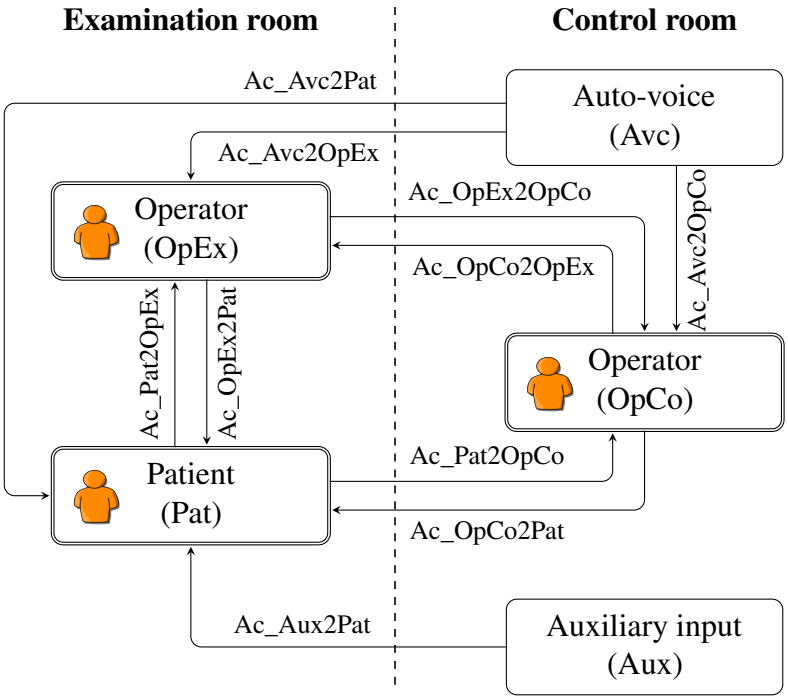


Figure 7.1: Overview of the audio channels

and can be updated by any automaton. These automata are referred to as extended automata. The algorithms defined in [62] are implemented in the Supremica [65] tool.

7.2 Description of the patient communication system

In this chapter a supervisor for the patient communication system, see Section 4.3.2, is generated. The communication system consists of audio channels that should be opened and closed depending on the input on the different user interfaces.

7.2.1 Audio channels

The audio signals traverse from the sources to the targets via audio channels. Figure 7.1 shows an overview of the audio channels. Audio sources are depicted by single line boxes, while audio source and sinks are depicted by double line boxes. The audio channels are depicted by arrows connecting the sources to the targets. Audio channels can be opened and closed. Audio channels are unidirectional, meaning that audio signals are transferred in one direction only. For two-way communication, an audio channel in each direction must be open.

7.2.2 Communication modes

The communication system can operate in various communication modes. In each mode, various entities communicate with one another. A communication mode may be active or inactive. The following communication modes are implemented in the system:

- `OpEx_ListenToPat`: The operator in the examination room listens to the patient.
- `OpEx_TalkWithPat`: The operator in the examination room talks with the patient (two way conversation).
- `OpCo_ListenToPat`: The operator in the control room listens to the patient.
- `OpCo_TalkWithPat`: The operator in the control room talks with the patient (two way conversation).
- `OpCo_ListenToOpEx`: The operator in the control room listens to the operator in the examination room.
- `OpCo_TalkWithOpEx`: The operator in the control room talks with the operator in the examination room (two way conversation).
- `AutovoiceToAll`: The auto-voice messages are sent to the patient and both operators.
- `MusicToPatient`: The patient listens to music from the auxiliary audio source.

When a communication mode is active, the relevant channels must be open to allow the communication corresponding to that mode. Otherwise, the channels should be closed. Table 7.1 shows which channels are used for each communication mode.

Note that in principle all communication modes could be enabled at the same time. However, requirements may prevent some modes being enabled simultaneously. For instance, the requirement that the patient may only hear one source at a time prevents the modes `MusicToPatient` and `OpCo_TalkWithPat` to be active simultaneously. These conflicts are resolved by introducing priorities for the communication modes. In the case described above, the priorities ensure that mode `MusicToPatient` can only be active if mode `OpCo_TalkWithPat` is not active.

7.2.3 Requests

The communication modes in the communication system are activated and deactivated via requests to activate or deactivate a mode. These requests are made by the host and the users. The users of the system are the patient, the OpEx and the OpCo. The MRI host system makes requests via a network. The host is informed if a request is accepted or rejected via the same network. The users make requests

Communication mode	Channels to open
`OpEx_ListenToPat`	Ac_Pat2OpEx
`OpEx_TalkWithPat`	Ac_Pat2OpEx Ac_OpEx2Pat
`OpCo_ListenToPat`	Ac_Pat2OpCo
`OpCo_TalkWithPat`	Ac_OpCo2Pat Ac_Pat2OpCo
`OpCo_ListenToOpEx`	Ac_OpEx2OpCo
`OpCo_TalkWithOpEx`	Ac_OpEx2OpCo Ac_OpCo2OpEx
`AutovoiceToAll`	Ac_Avc2Pat Ac_Avc2OpEx Ac_Avc2OpCo
`MusicToPatient`	Ac_Aux2Pat

Table 7.1: Communication modes with the corresponding channels

via buttons. The users are informed about the internal state of the communication system via indicators (LEDs).

A communication mode can be active only if there is a corresponding request. If no other conflicting higher priority communication modes are active, the request is granted immediately; otherwise the requested communication mode is activated as soon as all other higher priority conflicting communication modes are inactive.

For each communication mode, there is an indicator that visualizes the current state of the mode. Each indicator distinguishes three cases:

- communication mode is active;
- communication mode is inactive and requested;
- communication mode is inactive and not requested.

The buttons and the associated indicators together form the user interface. Table 7.2 shows the buttons that are in the system, and for each button to which communication mode it relates. Furthermore, it shows which communication modes are related to the host.

7.3 Specification formalism

This section informally introduces a subset of the supervisory control specification formalism used to specify the plant model and the control requirements of the patient communication system. This formalism is based on a small subset of the general Compositional Interchange Format (CIF) for hybrid systems, see [1, 74]. The subset has several concepts in common with the *extended automata* defined in [62];

User	Button	Communication mode
Patient	Nursecall	`OpCo_ListenToPat`
OpEx	Talk with patient	`OpEx_TalkWithPat`
	Listen to patient	`OpEx_ListenToPat`
	Talk to OpCo	`OpCo_ListenToOpEx`
	Patient music	`MusicToPatient`
OpCo	Talk with patient	`OpCo_TalkWithPat`
	Listen to patient	`OpCo_ListenToPat`
	Talk with OpEx	`OpCo_TalkWithOpEx`
	Listen to OpEx	`OpCo_ListenToOpEx`
	Patient music	`MusicToPatient`
Host		`OpCo_ListenToPat`
		`AutovoiceToAll`

Table 7.2: Relation between the user interface buttons and the communication modes

in particular, untimed automata with shared discrete variables, shared uncontrollable and controllable events, guards, and assignments. Differences are that this formalism has several additional concepts from CIF: the independent and dependent variables (referred to as algebraic variables in CIF), the invariants, and multiple initial locations.

7.3.1 Automata

An automaton consists of locations Q, variables V, alphabet Σ, and transitions E. The variables are divided into independent and dependent variables. They are discrete, and have a finite domain. The values of independent variables can be changed by means of assignments. An automaton can assign only its own variables. The value of a variable of another automaton can be referred to by prefixing such a variable by the name of the automaton that declares the variable. E.g. if a variable x is declared in an automaton A, then the value of that variable can be referred to as $A.x$ in another automaton. Independent boolean variables that are not explicitly initialized are initialized to false by default. The value of each dependent variable is obtained by evaluation of its defining expression. The events are divided into controllable and uncontrollable events. The supervisor can disable controllable events in certain states, whereas uncontrollable events cannot be disabled. The supervisor can prevent occurrence of uncontrollable events in certain states only by disabling controllable events in all states that may lead to occurrence of the uncontrollable event. The defined automata are composed in parallel and synchronize on shared events. An edge consists of a number of optional components: an event, a guard in the form of a predicate over variables and location references, and a multi-assignment on one or more variables.

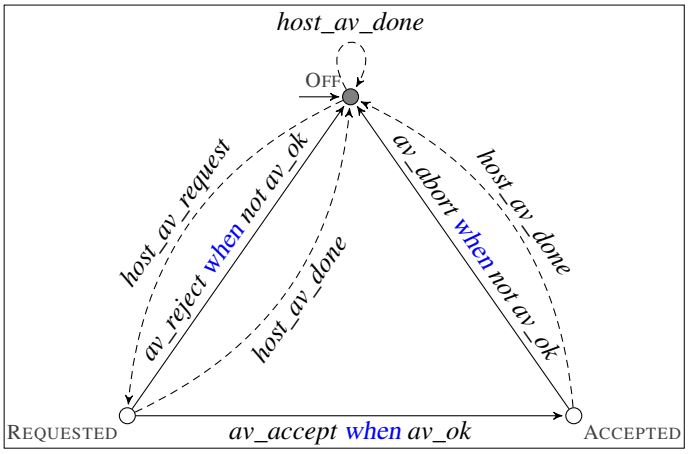


Figure 7.2: Example automaton `Ex`.

A graphical representation of a supervisory control automaton is shown in Figure 7.2. The automaton has locations `Off`, `Requested`, `Accepted`, controllable events *av_reject, av_accept, av_abort*, uncontrollable events *host_av_request, host_av_done*, and guard expressions that are represented by *not av_ok* and *av_ok*.

Locations are modeled by vertices, and transitions are modeled by arrows. All states labels are unique. For instance, in Figure 7.2, `Ex.Off` refers to state `Off` in automaton `Ex`. Controllable and uncontrollable transitions are represented by solid and dashed arrows, respectively.

Dependent variables are declared as aliases:

```
alias DEPENDENTVAR = EXPR
```

where EXPR is an expression over variables, location references and constants. Both independent and dependent variables can be used in EXPR as long as no circular dependencies are introduced.

A transition is labeled as follows:

```
EVENTLABEL when GUARD do MULTI-ASSIGNMENT
```

where:

- GUARD: Optional guard predicate over variables V and states Q. The predicates **P** are defined as follows:

  $$\mathbf{P} ::= \mathit{true} \mid \mathtt{s} \mid v = e \mid \mathtt{not}\ \mathbf{P} \mid \mathbf{P}\ \mathrm{op}\ \mathbf{P}$$

  where `s` is a location reference, $v \in V$ is an variable, e is a boolean or integer expression over variables and constants, and $\mathrm{op} \in \{\ and, or, =>\ \}$ is a logical operator. Note that only the subset of the language that is actually used in this chapter is presented.

- EVENTLABEL: An event label $\sigma \in \Sigma$.

- MULTI-ASSIGNMENT: Optional multi-assignment $v_1 := e_1, \cdots, v_n := e_n$ where v_i is an independent variable and e_i is an expression over variables (independent and/or dependent) and constants.

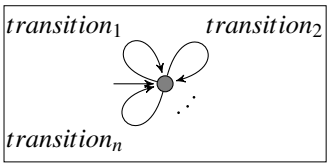


Figure 7.3: Graphical representation of an automaton with a single location.

```
event transition1
event transition2
 ...
event transitionn
```

Figure 7.4: Textual representation of an automaton with a single location.

The guard and the keyword 'when' are omitted when the guard is always true. The multi-assignment and the keyword 'do' are omitted when the variables are not updated.

Execution of a transition transforms the state before execution, the *old* state, to a new state. The guards and the expressions e_i, including any dependent variables (if present), are evaluated in the old state. In the new state, the values of the assigned (independent) variables are equal to the values of the expressions e_i evaluated in the old state. The values of the independent variables that have not been assigned remain unchanged.

The well-known 'if then else' statement can be associated with an event by means of the following notation:

```
EVENTLABEL when GUARD do if GUARD'
                           then MULTI-ASSIGNMENT1
                           else MULTI-ASSIGNMENT2
                        end
```

which is an abbreviation for:

```
EVENTLABEL when GUARD and     GUARD' do MULTI-ASSIGNMENT1
EVENTLABEL when GUARD and not GUARD' do MULTI-ASSIGNMENT2
```

A textual representation is introduced for an automaton with a single location and self-loops, as shown in Figure 7.3. The textual model hides the single location, which is not relevant in this case. The textual representation of this automaton is shown in Figure 7.4. This textual representation is easer to read for models that have many self-loops.

7.3.2 Requirement invariants

A requirement invariant is a predicate **P** over variables and location references (as defined in Section 7.3.1) that must always evaluate to true. The invariant defines which states are safe in the controlled system. Requirement invariants are used in Section 7.8.4. However, plant invariants are not used in this chapter.

7.4 Control architecture

The control architecture of the patient communication system is shown in Figure 7.5. MRI host H and user U generate uncontrollable events σ_{H_U} and σ_{U_U}, respectively,

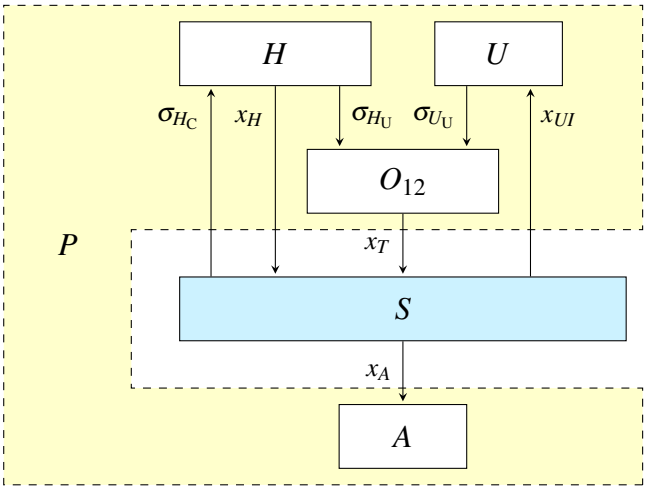


Figure 7.5: Control architecture

that activate or deactivate requests for communication modes in the observer O_{12}. This observer uses variables to keep track of whether a communication mode is requested. It consists of the two observers `O1` and `O2`, presented in Listings 7.1 and 7.2, respectively. The domain of the state of the supervisor is the union of the domain of the state x_T of the observer and the domain of the state x_H of the host. The output states x_{UI} and x_A, that are made available to the indicators of the user interface U and to the audio channels A, respectively, are defined by functions on the state of the supervisor. The controllable events σ_{H_C} that are sent to the host H are enabled/disabled by the supervisor on the basis of the supervisor's state.

7.5 Plant model P

The plant model is a parallel composition of the models of the host H, user U, audio channels A, and observer O_{12}.

7.5.1 Host model H

The host behavior consists of two parts: host notifications, and host auto-voice requests. These aspects of host behavior are described in the following subsections.

Host notifications

The interface σ_{H_U} (see Table 7.3) from host H to observer O_{12} is event-based. The host informs the communication system about host state changes that might require actions to be taken by the communication system; For instance, to improve the workflow, or to warn that a hazardous situation might occur. The communication system is notified when a scan starts and stops (workflow), and when the table starts and stops moving (hazard; the patient's fingers might become trapped), see

	event	contr.
σ_{H_U}	`host_scan_started`	No
	`host_scan_stopped`	No
	`host_tablemove_started`	No
	`host_tablemove_stopped`	No

Table 7.3: Events from host to observer

the events specified in Table 7.3. The host itself does not restrict the occurrences of these host notification events. Therefore, this aspect of the host behavior is not explicitly modeled. Note that the absence of restrictions for some events can be modeled as a single state automaton with self-loops for each of the events. Adding such an automaton does not change the meaning of the model if the event already occurs in other automata.

Host auto-voice requests

The host can request enabling of the communication mode `AutovoiceToAll`. In this communication mode, an auto-voice message is played to the patient and to the operators. In certain circumstances, such an auto-voice request from the host may be ignored by the communication system. Therefore, the communication control system must notify the host whether the request is accepted or rejected. This is done according to the events listed in Table 7.4. The host behavior is specified by automaton `H` as presented in Figure 7.6.

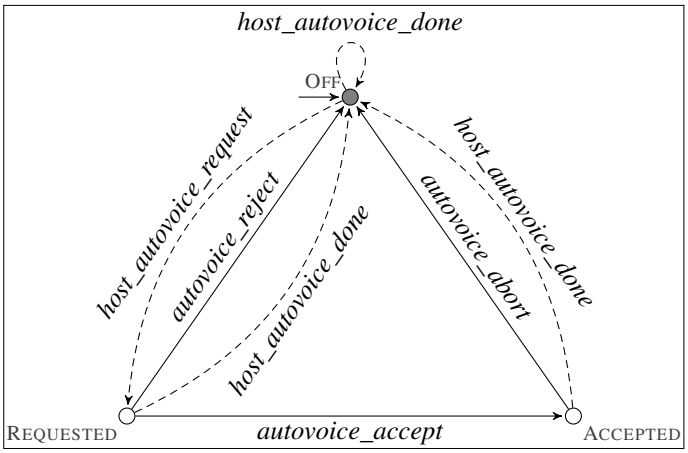


Figure 7.6: Host plant automaton `H`

Initially, the automaton is in mode `Off`. After a request from the host, the automaton changes to mode `Requested`. The request is accepted or rejected via the controllable events `autovoice_accept` and `autovoice_reject`, respectively. An accepted `AutovoiceToAll` request from the host, which results in playing of the auto-voice message, can be aborted by the supervisor via event `autovoice_abort`.

	event	contr.
σ_{H_C}	autovoice_reject	Yes
	autovoice_accept	Yes
	autovoice_abort	Yes

Table 7.4: Events from supervisor to host

	automaton location
x_H	Off
	Requested
	Accepted

Table 7.5: Locations in interface from host to supervisor

The host can also terminate AutovoiceToAll requests by means of the uncontrollable event host_autovoice_done. The state x_H (see Table 7.5) of the host automaton is made available to the supervisor in the form of location references: the boolean expression H.ℓ, where ℓ can be any of the three location names of automaton H, is true if and only if that location is active.

7.5.2 User model U

The interface σ_{U_U} (see Table 7.6) from user U to observer O_{12} is event-based. When a user pushes a button, an event is generated which is sent to the observer. The user interface has two kinds of buttons: push-buttons and hold-buttons. A push-button generates an event <button>_pushed when it is pushed. A hold-button generates two events: <button>_pressed and <button>_released. The user interface itself does not restrict the occurrence of these events in any way. Therefore this interface is not explicitly modeled.

The interface between the supervisor S and the user U is state-based. The state of the indicators in the user interface is a function of the state of the supervisor. Each indicator has three states: inactive, hold and active. Therefore, an indicator is modeled by a dependent enumerated variable with the values $\{Inactive, Hold, Active\}$. The relevant variables are given in Table 7.7.

7.5.3 Observer model O_{12}

The observer O_{12} of Figure 7.5 is modeled by two automata O1 and O2, shown in Listings 7.1 and 7.2, respectively. The observers receive uncontrollable events from the host H and the user U, and relate these events to request for communication modes. The requests are modeled by the variables shown in Table 7.8.

Note that the observers do not impose any restrictions on the occurrence of the events. Therefore, the observers could also have been introduced as part of the

	event	contr.
σ_{U_U}	bt_opex_talkwithpat_pushed	No
	bt_opex_listentopat_pushed	No
	bt_opex_talktoopco_pushed	No
	bt_opex_patmusic_pushed	No
	bt_opco_listentopat_pushed	No
	bt_opco_listentoopex_pushed	No
	bt_opco_patmusic_pushed	No
	bt_opco_talkwithpat_pressed	No
	bt_opco_talkwithpat_released	No
	bt_opco_talkwithopex_pressed	No
	bt_opco_talkwithopex_released	No
	bt_pat_nursecall_pushed	No

Table 7.6: Events from user interface to observer

	output variable
x_{UI}	Indc_OpEx_ListenToPat
	Indc_OpEx_TalkWithPat
	Indc_OpCo_ListenToPat
	Indc_OpCo_TalkWithPat
	Indc_OpCo_ListenToOpEx
	Indc_OpCo_TalkWithOpEx
	Indc_AutovoiceToAll
	Indc_MusicToPatient

Table 7.7: User interface output variables

	requested communication mode
x_T	Rq_OpEx_ListenToPat
	Rq_OpEx_TalkWithPat
	Rq_OpCo_ListenToPat
	Rq_OpCo_TalkWithPat
	Rq_OpCo_ListenToOpEx
	Rq_OpCo_TalkWithOpEx
	Rq_MusicToPatient

Table 7.8: Observer request variables

control requirements instead of as part of the plant. There would be no difference for the resulting synthesized supervisor.

The observers are essential for evolvability of the control system, because they relate sequences of input actions to values of the request variables. These request variables were considered to be highly intuitive by the domain specialists, because they allow reasoning about the system at a higher level of abstraction. All changes to input action sequences required for communication mode requests can remain local to the observer specifications, as long as the names of the request variables do not change.

Observer O1

Observer O1 declares the variables from Table 7.8, apart from variable Rq_OpCo_ListenToPat, which is declared in observer O2. The variables are of type boolean. If the value of a request variable is true, the associated communication mode is requested, otherwise no action is taken. Note that omission of the initial value of a variable of a certain type, defaults to initialization of that variable to the default initial value for that type. For boolean variables, the default initial value is false. Therefore, the request variables are initially false. The requests that are modeled by observer O1 in Listing 7.1 are toggled or always set to true or false when an event occurs.

Note that the events bt_opex_talkwithpat_pushed, host_scan_started and host_scan_stopped are included in both Listings 7.1 and 7.2. Due to the synchronous composition of these two automata, occurrence of an aforementioned event causes multiple requests for modes to be enabled (or disabled) simultaneously. The two automata O1 and O2 could be combined in one automaton. In that case, the assignments associated to the events bt_opex_talkwithpat_pushed, host_scan_started and host_scan_stopped in both automata would have to be combined as multi-assignments.

The variable Rq_OpEx_TalkWithPat is declared in observer O1, where its value is updated. The value is used in observer O2, where it is referred to as O1.Rq_OpEx_TalkWithPat.

Observer O2

Listing 7.2 shows how the requests for communication mode OpCo_ListenToPat are modeled. The OpCo listens to the patient when the patient pushes the nursecall-button or when table is moving. Furthermore, the OpCo overhears the conversation between the OpEx and the patient. Finally, the OpCo can activate and deactivate the mode OpCo_ListenToPat whenever he or she likes. The four distinct cases must not influence one another. Therefore, four variables are used to model the request for communication mode OpCo_ListenToPat, each modeling a sub-request. The sub-request can only be deactivated by the same entity that activated it, with exception of the OpCo who can deactivate all sub-requests. The required behavior is modeled by the dependent variable (*alias*) Rq_OpCo_ListenToPat, which is

```
1  plant O1:
2    uncontrollable
3      host_scan_started, host_scan_stopped,
4      bt_opex_listentopat_pushed, bt_opex_talkwithpat_pushed,
5      bt_opex_talktoopco_pushed, bt_opex_patmusic_pushed,
6      bt_opco_listentoopex_pushed, bt_opco_patmusic_pushed,
7      bt_opco_talkwithopex_pressed, bt_opco_talkwithopex_released,
8      bt_opco_talkwithpat_pressed, bt_opco_talkwithpat_released;

10   var bool Rq_OpEx_ListenToPat,  Rq_OpEx_TalkWithPat,
11            Rq_OpCo_ListenToOpEx, Rq_OpCo_TalkWithOpEx,
12            Rq_MusicToPatient,    Rq_OpCo_TalkWithPat;

14   // host:
15   event host_scan_started             do Rq_OpEx_ListenToPat  := false
16                                        , Rq_OpEx_TalkWithPat  := false;
17   event host_scan_stopped             do Rq_OpEx_ListenToPat  := true;

19   // user interface buttons:
20   event bt_opex_listentopat_pushed
21       do Rq_OpEx_ListenToPat  := not Rq_OpEx_ListenToPat;
22   event bt_opex_talkwithpat_pushed
23       do Rq_OpEx_TalkWithPat  := not Rq_OpEx_TalkWithPat;
24   event bt_opex_talktoopco_pushed
25       do Rq_OpCo_ListenToOpEx := not Rq_OpCo_ListenToOpEx;
26   event bt_opex_patmusic_pushed
27       do Rq_MusicToPatient    := not Rq_MusicToPatient;
28   event bt_opco_listentoopex_pushed
29       do Rq_OpCo_ListenToOpEx := not Rq_OpCo_ListenToOpEx;
30   event bt_opco_patmusic_pushed
31       do Rq_MusicToPatient    := not Rq_MusicToPatient;
32   event bt_opco_talkwithopex_pressed  do Rq_OpCo_TalkWithOpEx := true;
33   event bt_opco_talkwithopex_released do Rq_OpCo_TalkWithOpEx := false;
34   event bt_opco_talkwithpat_pressed   do Rq_OpCo_TalkWithPat  := true;
35   event bt_opco_talkwithpat_released  do Rq_OpCo_TalkWithPat  := false;
36 end
```

Listing 7.1: Observer O1

defined in terms of the four independent variables Rq_OpCo_ListenToPat_i, $i \in \{\texttt{opco}, \texttt{opex}, \texttt{host}, \texttt{nursecall}\}$, thus resulting in one request variable for this mode that can be used in the remainder of the specifications. The suffix of the variables indicates —for each submode—which entity enabled the request.

7.5.4 Rationale for using automata with variables

Analysis of the specification of the second observer in Listing 7.2 provides the rationale for using specifications based on automata that have been extended with variables. The majority of the control requirements can easily be specified using automata without variables, where each boolean variable is represented by an au-

```
1  plant O2:
2    uncontrollable
3      host_tablemove_started, host_tablemove_stopped,
4      bt_pat_nursecall_pushed, bt_opco_listentopat_pushed;

6    var bool Rq_OpCo_ListenToPat_nursecall, Rq_OpCo_ListenToPat_host,
7             Rq_OpCo_ListenToPat_opex,      Rq_OpCo_ListenToPat_opco;

9    alias bool Rq_OpCo_ListenToPat = Rq_OpCo_ListenToPat_opco
10                                    or Rq_OpCo_ListenTopat_opex
11                                    or Rq_OpCo_ListenToPat_host
12                                    or Rq_OpCo_ListenToPat_nursecall;

14   // host:
15   event O1.host_scan_started    do Rq_OpCo_ListenToPat_opco := false;
16   event O1.host_scan_stopped    do Rq_OpCo_ListenToPat_opco := true;
17   event host_tablemove_started  do Rq_OpCo_ListenToPat_host := true;
18   event host_tablemove_stopped  do Rq_OpCo_ListenToPat_host := false;

20   // user interface buttons:
21   event bt_pat_nursecall_pushed
22       do Rq_OpCo_ListenToPat_nursecall := true;
23   event O1.bt_opex_talkwithpat_pushed
24       do Rq_OpCo_ListenToPat_opex      := not O1.Rq_OpEx_TalkWithPat;
25   event bt_opco_listentopat_pushed
26       do if Rq_OpCo_ListenToPat
27             then
28               Rq_OpCo_ListenToPat_nursecall := false,
29               Rq_OpCo_ListenToPat_host      := false,
30               Rq_OpCo_ListenToPat_opex      := false,
31               Rq_OpCo_ListenToPat_opco      := false
32             else
33               Rq_OpCo_ListenToPat_opco      := true
34             end;
35 end
```

Listing 7.2: Observer O2

tomaton with two locations: one for the value `true`, and one for the value `false`. In particular, all control requirements where variables are updated to a literal value (in this case either `true` or `false`), can be easily represented by means of two edges in the specific automaton: a self loop in the location corresponding to the assigned value, and a transition from the other location to the location corresponding to the assigned value. There are, however, two specifications where the variables cannot so easily be eliminated: in particular lines 23 and 24, where the new value of variable `Rq_OpCo_ListenToPat_opex` depends on the value of *another* variable `O1.Rq_OpEx_TalkWithPat`; and lines 25 - 34, where a guard is used that is represented by the variable `Rq_OpCo_ListenToPat`, the value of which is defined in terms of a *disjunction* of four other variables. The variables in these control requirements can be eliminated according to the algorithm defined in [62], but at the

	output variable
x_A	Ac_Pat2OpEx_Opened
	Ac_Pat2OpCo_Opened
	Ac_OpEx2Pat_Opened
	Ac_OpEx2OpCo_Opened
	Ac_OpCo2Pat_Opened
	Ac_OpCo2OpEx_Opened
	Ac_Sys2Pat_Opened
	Ac_Sys2OpEx_Opened
	Ac_Sys2OpCo_Opened
	Ac_Mus2Pat_Opened

Table 7.9: Audio interface output variables

cost of duplication/multiplication of the events (bt_opex_talkwithpat_pushed and bt_opco_listentopat_pushed) used in the requirements. The resulting specifications, without variables, but with the additional artificially created events, would be much less intuitive than the presented specifications.

7.5.5 Audio channel model A

The audio channels are controlled by directly setting the state of the channels to opened or closed. Each channel is modeled by a boolean variable. If the value is true, the channel is opened, otherwise it is closed. The variables are given in Table 7.9.

7.6 Control requirements model

Based on the values of the variables and location references, supervisor S generates control outputs. The outputs can be event-based or state-based, depending on the interface for the control output. The supervisor enables and disables events used in observer O_{12} and host H. The supervisor directly sets the indicator variables in U and the variables for the audio channels in A.

7.6.1 Communication mode priorities

The operators and the patient can each listen to multiple audio sources. For clarity, no more than one audio source may be heard at any one time. This implies that for each target, at most one of the audio channels to that target may be in the state open at one time. Therefore, communication modes may only be enabled at the same time if all corresponding channels can be opened at the same time.

Table 7.10 defines a partial order on the priorities of the communication modes. The tick marks in the table indicate the required resources for each mode. There is an ordering among communication modes only if they share resources. For such

modes, the highest listed mode in Table 7.10 has the highest priority. However, if communication modes do not share resources, they can be active at the same time.

The requirements that define when each communication mode is enabled are derived from the priority table. A mode is enabled when its associated request variable is true, and when there is no higher priority mode enabled that shares one or more resources. For instance, mode `OpEx_TalkWithPat` is enabled only when the modes `OpCo_TalkWithPat` and `OpCo_TalkWithOpEx` are not active, whereas the modes `MusicToPatient` and `OpCo_TalkWithOpEx` are unordered: they can be active at the same time. These priority requirements are specified using aliases that define the values of the (dependent) communication mode variables in terms of the (independent) request variables and other, already defined, communication mode variables. In this way, the value of each communication variable is ultimately defined as a function of request variables. This is essential for the evolvability of the control system. Changes in the required priority rules can remain local to the alias definitions, because the only interaction with the other parts of the control system specification is by means of the names of the requests and the communication modes.

```
alias bool
  OpCo_TalkWithPat  = O1.Rq_OpCo_TalkWithPat,
  OpCo_ListenToPat  = O2.Rq_OpCo_ListenToPat
                      and not OpCo_TalkWithPat,
  OpCo_TalkWithOpEx = O1.Rq_OpCo_TalkWithOpEx
                      and not OpCo_TalkWithPat
                      and not OpCo_ListenToPat,
  OpCo_ListenToOpEx = O1.Rq_OpCo_ListenToOpEx
                      and not OpCo_TalkWithPat
                      and not OpCo_ListenToPat
                      and not OpCo_TalkWithOpEx,
  OpEx_TalkWithPat  = O1.Rq_OpEx_TalkWithPat
                      and not OpCo_TalkWithPat
                      and not OpCo_TalkWithOpEx,
  OpEx_ListenToPat  = O1.Rq_OpEx_ListenToPat
                      and not OpCo_TalkWithOpEx
                      and not OpEx_TalkWithPat,
  AutovoiceToAll    = H.Accepted
                      and Autovoice_OK,
  MusicToPatient    = O1.Rq_MusicToPatient
                      and not OpCo_TalkWithPat
                      and not OpEx_TalkWithPat
                      and not AutovoiceToAll;
```

where `Autovoice_OK` is defined as:

```
alias bool Autovoice_OK = not OpCo_TalkWithPat
                          and not OpCo_ListenToPat
                          and not OpCo_TalkWithOpEx
                          and not OpCo_ListenToOpEx
                          and not OpEx_TalkWithPat
                          and not OpEx_ListenToPat;
```

	Resources (Targets)		
Boolean variable	Patient	OpCo	OpEx
OpCo_TalkWithPat	✓	✓	
OpCo_ListenToPat		✓	
OpCo_TalkWithOpEx		✓	✓
OpCo_ListenToOpEx		✓	
OpEx_TalkWithPat	✓		✓
OpEx_ListenToPat			✓
AutovoiceToAll	✓	✓	✓
MusicToPatient	✓		

Table 7.10: Communication mode priorities

7.6.2 Auto-voice

Auto-voice requests are accepted only if the corresponding audio channels can be opened, indicated by `Autovoice_OK`. If the channels remain closed or be closed due to an active higher priority communication mode, the request is rejected or aborted, respectively. This behavior is defined by the control requirement `AV` defined below, in combination with the host model `H` presented in Figure 7.6.

```
requirement AV:
  event autovoice_accept when O1.Autovoice_OK;
  event autovoice_reject when not O1.Autovoice_OK;
  event autovoice_abort  when not O1.Autovoice_OK;
end
```

7.6.3 Channel open and close

Table 7.1 in Section 7.2.2 defines—for each communication mode—which channels must be open when the mode is active. Based on this table, the globally defined aliases below, define for each channel when it is open and when it is closed. The aliases define the interface between the communication modes in the control system, and the actual channels in the hardware specification, and in this way contribute to the evolvability of the control system design.

```
alias bool
  Ac_Pat2OpEx_Opened  = OpEx_TalkWithPat
                        or OpEx_ListenToPat,
  Ac_Pat2OpCo_Opened  = OpCo_TalkWithPat
                        or OpCo_ListenToPat,
  Ac_OpEx2Pat_Opened  = OpEx_TalkWithPat,
  Ac_OpEx2OpCo_Opened = OpCo_TalkWithOpEx
                        or OpCo_ListenToOpEx,
  Ac_OpCo2Pat_Opened  = OpCo_TalkWithPat,
  Ac_OpCo2OpEx_Opened = OpCo_TalkWithOpEx,
  Ac_Aux2Pat_Opened   = MusicToPatient,
```

```
Ac_Avc2Pat_Opened   = AutovoiceToAll,
Ac_Avc2OpEx_Opened  = AutovoiceToAll,
Ac_Avc2OpCo_Opened  = AutovoiceToAll;
```

7.6.4 Indicators

For each communication mode, there is an indicator that visualizes the current state of the mode. The value of each indicator is specified by means of a conditional expression. An indicator is in state `Active` if the corresponding communication mode is active. It is in state `Hold` if the mode is requested but not active, and it is in state `Inactive` when the mode is not requested. For example for communication mode `OpEx_ListenToPat`:

```
alias LEDS
  Indc_OpEx_ListenToPat =
    if OpEx_ListenToPat
    then Active
    else if Rq_OpEx_ListenToPat then Hold else Inactive end
    end;
```

7.7 Supervisor model *S*

The supervisor consists of the aliases that define the states of the audio channels, the indicators, and the synthesized control functions for the controllable events. The control functions are synthesized using the method described in Section 7.10. The supervisor control functions for the control problem defined in Sections 7.5 and 7.6 are equivalent to the control requirements as defined in Section 7.6.2. Therefore, for this control problem, the control functions could be implemented by means of the control requirement automata.

Note that the control requirements defined in the preceding sections are the result of an iterative engineering process. In each iteration, the synthesized supervisor is tested by means of interactive, user-guided simulation to establish whether the defined control requirements define the behavior required by the customer. In these iterations, the control requirements can be blocking and/or uncontrollable (meaning that uncontrollable events are disabled by the control requirements). In such cases, synthesis is a valuable tool to help find errors in the control requirements by simulating the synthesized nonblocking and controllable supervisor on the plant model.

7.8 Event-based output

In the preceding sections, the interfaces from the supervisor to the user and to the audio channels are state-based. The output of the supervisor is a function of the communication modes and the other variables of the model. In this section, the

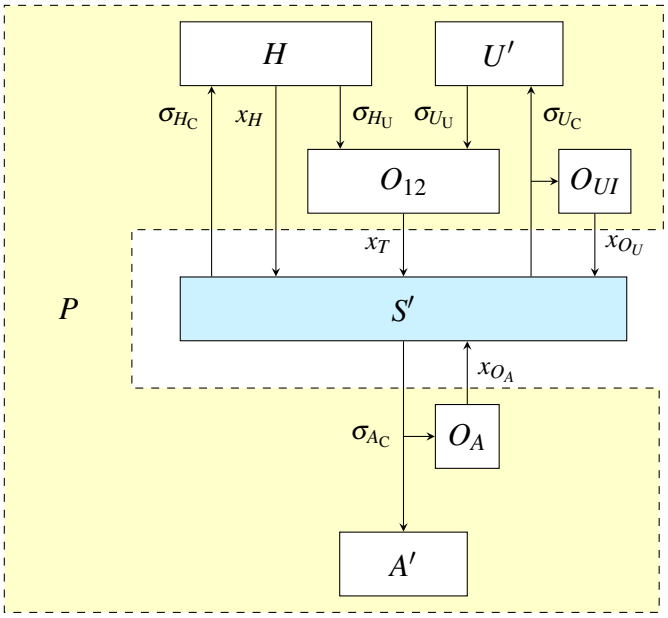


Figure 7.7: Control architecture for event-based output

interface between the supervisor and the user and audio channels is changed to be event-based: instead of defining the output state as a function of the state of the plant and the observer, events are used as outputs.

Figure 7.7 shows the new control architecture. The signals to users U' and audio channels A' are changed from variables ($x_{...}$) to events ($\sigma_{...}$), and two observers are added to the plant. Observer O_{UI} keeps track of the state of the indicators in U' by monitoring the events sent to U'. The state of O_{UI} is used by the supervisor to enable and disable events that are sent to U'. Observer O_A has the same purpose for the states and the events of A'.

The host model H remains the same. Also the events generated by the users U' remain the same. Therefore, the observer model O_{12} also stays unchanged. In fact, the new model clearly demonstrates the evolvability of the control system design: the changes can be implemented by means of *additional* specifications, without changes to the original specifications. The specifications of the state-based output variables defined in Sections 7.6.3 and 7.6.4 are no longer required in an event-based output system, but the specifications need not be removed. The only component that changes is the supervisor S, which can be generated using a synthesis algorithm.

In the following sections, the changes to the plant model and control requirement model are discussed.

7.8.1 User interface model U'

The state of the indicators is changed by sending events to the user interface. Table 7.11 shows the events related to mode `OpEx_ListenToPat`. For the other modes, similar events are used. The user interface accepts the events in any order, and is

	output event	contr.
σ_{U_C}	`indc_opexlistentopat_active`	Yes
	`indc_opexlistentopat_hold`	Yes
	`indc_opexlistentopat_inactive`	Yes

Table 7.11: New user interface output events

	output event	contr.
σ_{A_C}	`ac_opco2pat_open`	Yes
	`ac_opco2pat_close`	Yes

Table 7.12: New audio interface output events

therefore not explicitly modeled.

7.8.2 Audio channel model A'

The state of the audio channels is also changed by sending events. Table 7.12 shows the event for the audio channel between the OpCo and the patient. For the other channels, similar events exist. The interface accepts the events in any order, and is therefore not explicitly modeled.

7.8.3 Observers O_{UI} and O_A

The indicators in U' and the audio channels in A' do not provide feedback about the current state of the system. Therefore, observers are added to the plant model. These observers provide the current state of the indicators and audio channels for the supervisor. The indicator observer model for the mode `OpEx_ListenToPat` is given below. The state of the indicator is modeled using an enumerated variable with the values `Inactive`, `Hold` and `Active`. The other indicators are modeled in a similar way.

```
enum LEDS = {Inactive,Active,Hold};

plant O_UI:
  var LEDS Indc_OpEx_ListenToPat = Inactive;

  event indc_opexlistentopat_active
        do Indc_OpEx_ListenToPat := Active;
  event indc_opexlistentopat_hold
        do Indc_OpEx_ListenToPat := Hold;
  event indc_opexlistentopat_inactive
        do Indc_OpEx_ListenToPat := Inactive;
  // ... and so on for other indicators
end
```

The plant model `T_A` is the observer for the audio channels. The model below specifies only the channel between the Patient and the OpEx. The state of the audio channel is modeled using a boolean variable. The audio channel is opened and closed via controllable events `ac_pat2opex_open` and `ac_pat2opex_close`, respectively. The models of the other audio channels are similar.

```
plant O_A:
  var bool Ac_Pat2OpEx_Opened;

  event ac_pat2opex_open  do Ac_Pat2OpEx_Opened := true;
  event ac_pat2opex_close do Ac_Pat2OpEx_Opened := false;
  // ... and so on for other audio channels
end
```

7.8.4 Control requirement models

The control requirements defined in Section 7.6 remain unchanged. For event-based output, three types of requirements are added: requirements related to mutual exclusion of audio sources, to opening and closing of channels, and to setting of indicator states.

Mutual exclusion of audio sources

To prevent that the operators and the patient can hear multiple sources during transitions between control modes, a mutual exclusion requirement is added. This requirement prevents that channels with the same target may be open at the same time.

For compact notation of mutual exclusion requirements, a function is introduced. A set of mutually exclusive states imply that if any one of the states in a set is active, all other states must be inactive. The function is defined for a set of state predicates $\mathcal{SP}$:

$$mutex(\mathcal{SP}) = \bigwedge_{x \in \mathcal{SP}} \left(x \implies \left(\bigwedge_{y \in \mathcal{SP}, y \neq x} \neg y \right) \right) \tag{7.1}$$

For example $mutex(x,y,z) =$

$$\big(x \implies (\neg y \wedge \neg z)\big) \wedge \big(y \implies (\neg x \wedge \neg z)\big) \wedge \big(z \implies (\neg x \wedge \neg y)\big)$$

At all times, the following mutual exclusion rules must be obeyed for the patient, the operator in the examination room and the operator in the control room:

```
requirement Channel_mutex:
  invariant
    mutex( Ac_OpCo2Pat_Opened
         , Ac_OpEx2Pat_Opened
         , Ac_Avc2Pat_Opened
         , Ac_Aux2Pat_Opened ),
    mutex( Ac_OpCo2OpEx_Opened
         , Ac_Pat2OpEx_Opened
         , Ac_Avc2OpEx_Opened ),
```

```
    mutex( Ac_Pat2OpCo_Opened
         , Ac_OpEx2OpCo_Opened
         , Ac_Avc2OpCo_Opened )
end
```

Opening and closing of the channels

The following requirement defines when a channel open or close event may occur. The other channels are opened and closed using similar rules.

```
requirement Audio_Channel_events:
  event ac_pat2opex_open
       when ( OpEx_TalkWithPat or OpEx_ListenToPat )
       and not Ac_Pat2OpEx_Opened;
  event ac_pat2opex_close
       when not ( OpEx_TalkWithPat or OpEx_ListenToPat )
       and Ac_Pat2OpEx_Opened;
end
```

Setting the indicator states

The following requirement defines when the output events for the OpEx listen to Pat indicator may occur. The other output events are defined using similar rules:

```
requirement Indicator_events:
  event indc_opexlistentopat_active
       when Rq_OpEx_ListenToPat and OpEx_ListenToPat
            and Indc_OpEx_ListenToPat != Active;
  event indc_opexlistentopat_hold
       when Rq_OpEx_ListenToPat and not OpEx_ListenToPat
            and Indc_OpEx_ListenToPat != Hold;
  event indc_opexlistentopat_inactive
       when not Rq_OpEx_ListenToPat
            and Indc_OpEx_ListenToPat != Inactive;
end
```

7.9 Supervisor model S'

As an example of the result of the synthesis procedure, the generated supervisory control rules for the audio channel from the patient to the OpEx in the form of a supervisory control automaton is shown below. The restrictions that the other channels must be closed before the channel can be opened are the result of the synthesis algorithm:

```
event ac_pat2opex_open
     when not ac_Pat2OpEx_Opened
          and not Ac_Avc2OpEx_Opened
          and not Ac_OpCo2OpEx_Opened
          and ( OpEx_TalkWithPat or OpEx_ListenToPat )
event ac_pat2Opex_close
```

```
when ac_Pat2OpEx_Opened
     and not ( OpEx_TalkWithPat or OpEx_ListenToPat )
```

7.10 Toolchain

Figure 7.8 shows the toolchain that was used to synthesize the supervisor.

First, the automata of the plant model $P_{\mathbb{A}}$ are transformed into regular automata P_{A} by elimination of the variables, using the algorithm defined in [62]. The output of this algorithm are automata without variables P_{A}, and a mapping table M that maps the variables, states and events in the $\mathbb{A}$ automata to states and events in the regular automata.

Second, control requirement automata $R_{\mathbb{A}}$ are translated to (generalized) state-transition predicates $R_{\mathrm{STP}_{\mathbb{A}}}$ [46]. Note that all requirements are modeled by self-loops without multi-assignments. Therefore, the translation is straightforward. A transition:

```
event EVENTLABEL when GUARD
```

is translated to the following state-transition predicate:

$$\rightarrow\texttt{EVENTLABEL} \Rightarrow \texttt{GUARD}\downarrow$$

which is equivalent to:

$$\neg\,\texttt{GUARD}\downarrow \Rightarrow \neg \rightarrow\texttt{EVENTLABEL}$$

meaning that the event EVENTLABEL is disabled in all states where the guard GUARD does not hold.

Third, state-transition predicates $R_{\mathrm{STP}_{\mathbb{A}}}$ and requirement invariants $R_{\mathrm{Inv}_{\mathbb{A}}}$ are mapped to the states of the plant without variables, resulting in $R_{\mathrm{STP}_{\mathrm{A}}}$ and $R_{\mathrm{Inv}_{\mathrm{A}}}$.

Fourth, regular plant automata P_{A} with the corresponding control requirements $R_{\mathrm{Inv}_{\mathrm{A}}}$ and $R_{\mathrm{STP}_{\mathrm{A}}}$ are used to synthesize the supervisor $S_{\mathrm{STP}_{\mathrm{A}}}$. The synthesis algorithm defined in [44], and the preprocessing transformation defined in [46] are used to synthesize the supervisor. These algorithms take as input the plant modeled as parallel automata and the requirements modeled as predicates over the states and events of the automata. The result of this synthesis method is a set of predicates $S_{\mathrm{STP}_{\mathrm{A}}}$, which define for each controllable event when the event is enabled. These predicates are defined over the states of the regular plant automata P_{A}.

Using mapping M, the synthesized state-transition predicates $S_{\mathrm{STP}_{\mathrm{A}}}$, are mapped to predicates over the states and variables that are used in the plant model $P_{\mathbb{A}}$. The resulting state-transition predicates are transformed to automata $S_{\mathbb{A}}$. This supervisor can be simulated together with plant model $P_{\mathbb{A}}$.

Alternatively,

7.11 Concluding remarks

First, the concepts of a specification formalism for supervisory control synthesis that is expressive and intuitive enough to be used by both domain experts and software

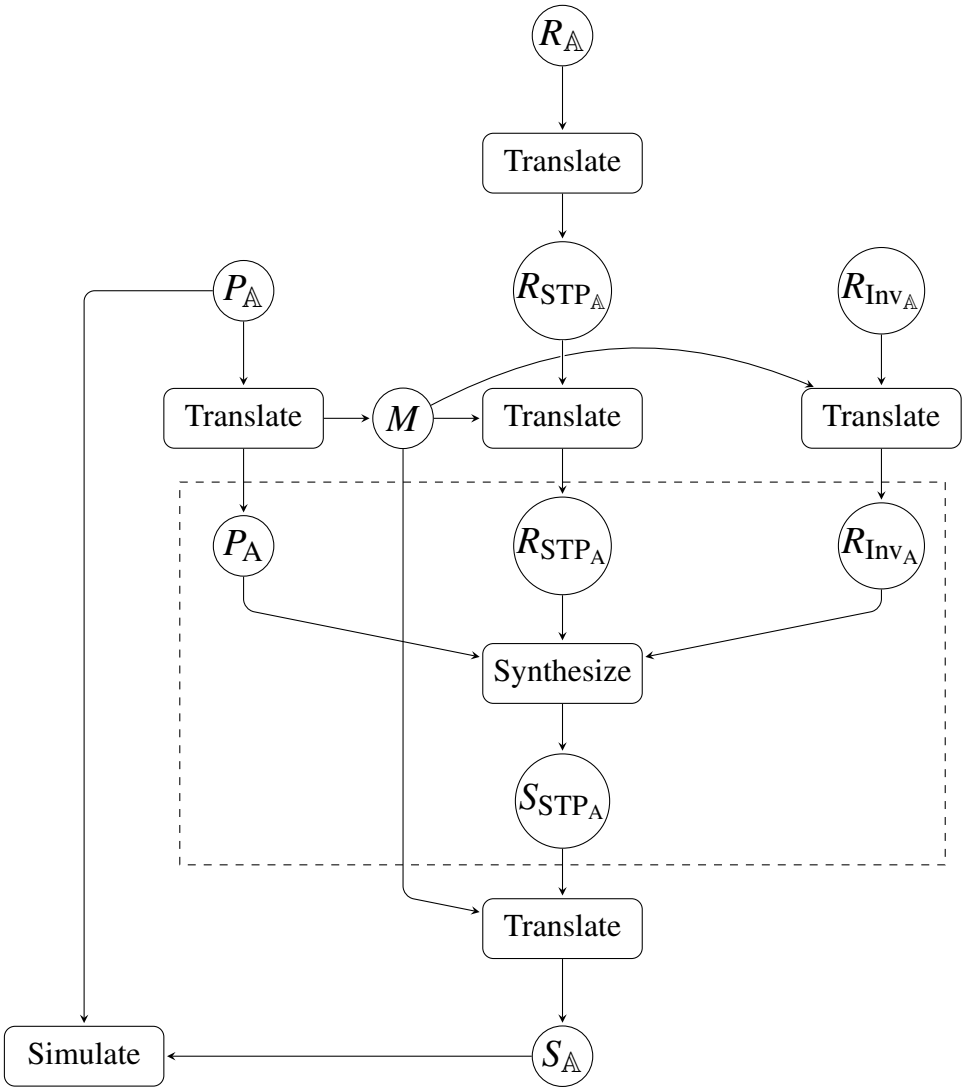


Figure 7.8: Toolchain, where P = Plant, R = Requirement, S = Supervisor, and the subscripts $\mathbb{A}$ = Automata with variables, A = Regular automata, STP = State-Transition Predicate, and Inv = Invariant.

experts is defined. The formalism is based on untimed automata and invariants, that can each be of type 'plant' or 'requirement'. Automata consist of locations, edges, and variables. An edge of an automaton consists of an event label, a guard, and a multi-assignment. The variables are divided in two classes: independent variables that can be assigned, and dependent variables whose values are obtained by evaluation of their defining expressions.

Second, the supervisory control specification formalism is applied for the control system design of a patient communication system of an MRI scanner. The application illustrates how the division of the specification into a number of small, relatively independent components with well-defined interfaces increases the evolvability of the control system.

Essential for the increased evolvability is the use of observers and both dependent and independent variables. The observers record the history of events, generated by the user interface and the host, in the independent request variables that model the requests for specific communication modes. This decouples the recording of the input events, from the requirements that specify which communication mode can be activated. These requirements define the value of each (dependent) communication mode variable as a function of its associated (independent) request variable and the values of other, higher priority, communication variables. Thus, changes in the observers and changes in the control requirements can be made independently of each other, as long as the *interface* between the observer and the control requirements in terms of the request variables remains unchanged.

It is explained why the use of variables in automata can lead to specifications that are more intuitive than equivalent specifications based on automata without variables. This is especially so in the case of specifications where the new value of a variable in an update depends on other variables, and in the case of disjunctions in guard expressions.

The output of the supervisor is defined as a function on the communication mode variables. This definition of the output of the supervisor as a function on the state of the observers, where observers do not restrict the occurrence of uncontrollable events in any way, leads to supervisory control specifications that are very easy to develop, understand and debug, and where deadlock is impossible. Deadlock is impossible, since observers do not restrict the occurrence of events, invariants and guards are not used (are always true), and the output functions are always defined.

As an illustration of the evolvability of the control system design, the specification is updated for event-based output, by adding events, observers, and requirement invariants that ensure mutual exclusion of communication channel activation. Thus, the defined communication mode interface decouples the output specification from the other control system requirements. The only component that is changed in the specification for event-based output is the supervisor, which can be generated using a supervisory control synthesis algorithm.

For implementation of the supervisory control synthesis algorithm, a sequence of steps connecting already existing supervisory control synthesis tools by means of various translations has been defined. These steps were manually executed to obtain the synthesized supervisors. Currently, the steps are automated. Future work is to define a synthesis algorithm that can directly manipulate the defined plant models and control requirements. This would eliminate the translation steps and reduce the complexity of the synthesis algorithm. A synthesis algorithm based on extended automata is defined in [62]. However, that algorithm does not support requirement invariants and results in a monolithic supervisor automaton without variables.

The automated steps shown in Figure 7.8 execute within seconds. The experiences with the supervisory control synthesis algorithm based on state tree structures ([44]) on bigger examples are also quite positive. The supervisor for a patient support system of an MRI scanner, where the uncontrolled system consisted of 6.3 billion states, is generated in seconds. However, to deal with much bigger systems, it is

expected that some form of modular supervisory control is required.

One of the obstacles in realizing evolvable systems is the difficulty to capture the customers needs. Customers find it difficult to envision what the system's behavior will be and what exact behavior is desired, when the desired system is not yet available. Therefore, the customer cannot tell in advance what the exact requirements for the system are. This is one of the reasons that informal control requirements are usually incomplete and and/or ambiguous. Only after the system has been created, can the customer evaluate if the system obeys the exact customer needs. As a result, iterations are required in all development stages to make the customer needs clear.

For interaction with the customer, the synthesized supervisor can be interactively simulated together with the plant model. For this purpose, a computer visualization of the user interface, with clickable buttons, and a visualization of the state of the communication channels can be connected to the interactive simulation. Such an interactive simulation based on a generated supervisor allows for early user feedback to establish whether or not the defined control requirements correspond to the behavior required by the customer. This interaction with the customer in the form of iterative testing of the synthesized controller by means of interactive simulation, and subsequent updates of the formal control requirements, allows the generation of a consistent set of requirements that meets the customer's expectation in a relatively short period of time. The same approach can also be used in the case of later changes if the system evolves over time.

The specifications presented in this chapter were used as the basis for the control system specification of the actual patient communication system, which was implemented by an external party. At the time of writing, the system was still under development, and the requirements were still changing due to interaction with the customer.

Chapter 8

Concluding remarks

In the traditional approach to controller design, behavioral requirements are informally specified by domain engineers, and software is coded by software engineers. This leads to long development cycles, and to code and requirements that are difficult to develop, debug, maintain, and adapt. Observed erroneous behavior of the system under test can be caused by, among others, ambiguous or inconsistent control requirements, miscommunication between domain engineer and software coder, and errors in the control code. This can be especially problematic when the functionality of existing products needs to evolve over time, such as in the case of MRI scanners that are upgraded to state of the art functionality over a period of ten years.

To address these issues, this thesis proposes the use of a single model of the control requirements and the uncontrolled plant, and to generate the control code by means of supervisory control synthesis. Although the supervisory control theory required for this originated as early as 1987 [57], it has not been until recently that the theory and associated tools have matured to a level that has enabled successful application in industry. This thesis has discussed two of such applications: supervisory control of the patient support system and of the patient communication system of MRI scanners. Recently, also the controller for the Gradient Amplifier of MRI scanners has been specified by means of supervisor control synthesis [23], and results were so positive that a successor project has been started to investigate supervisory control for all major components of MRI scanners.

A major factor in the success of supervisory control synthesis in industrial applications is the extension of event-based supervisory control with generalized state-based control requirements, as introduced in this thesis, and the availability of adequate tooling [67, 69]. Experience at Philips Healthcare has shown that generalized state-based control requirements are intuitive for both domain engineers and software engineers, since they closely match the view of the systems in terms of states, transitions between states, and restrictions on allowable states and state-

transitions. The use of state-based control requirements leads to models that are easier to develop, understand, and debug, and duplication of information, as can be observed in event-based supervisory control, is completely eliminated.

State-based supervisory control synthesis using state tree structures, as introduced by [43], also facilitates the use of multiple initial states, which is essential for actual real-time control of industrial systems. This was illustrated by the patient support table case. The real-time controller that was initially developed using event-based supervisory control (see Chapter 5), had one initial state only. Therefore, the patient support table had to be moved into the specified initial state, before the supervisory controller could be activated. The state-based supervisory controller that was later developed (see Chapter 6), could be activated immediately, independently from the initial state of the patient support system.

Even though extension of the event-based supervisor control framework with state references via locations of automata is an enormous step forward, Chapter 7 shows that additional state references via *variables* are essential for intuitive modeling of the various modes of operation of a patient communication system. An additional advantage of having automata with variables is that they facilitate state-based output. That is, the definition of the values of output variables as a function of the state of the control system. In combination with the use of observers, that record sequences of events in terms of states, this leads to specifications that are easy to develop, understand, and adapt. The new tool chain, as proposed in Figure 7.10, which includes variables for supervisory control synthesis, is currently implemented, see [69].

Bibliography

[1] J. C. M. Baeten, D. A. van Beek, D. Hendriks, A. T. Hofkamp, D. E. Nadales Agut, J. E. Rooda, and R. R. H. Schiffelers. Multiform Deliverable D1.1.2: Definition of the Compositional Interchange Format. `http://www.multiform.bci.tu-dortmund.de/images/stories/multiform/deliverables/multiform_d112.pdf`, 2010.

[2] J. C. M. Baeten, D. A. van Beek, B. Luttik, J. Markovski, and J. E. Rooda. A process-theoretic approach to supervisory control theory. In *Proceedings of ACC 2011*, pages 4496–4501. IEEE, 2011.

[3] Jos C. M. Baeten, Bert van Beek, Allan van Hulst, and Jasen Markovski. A process algebra for supervisory coordination. In *PACO*, pages 36–55, 2011.

[4] S. Balemi. Input/output discrete event processes and communication delays. *Discrete Event Dynamic Systems: Theory and Applications*, 4(1):41–85, 1994.

[5] S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. F. Franklin. Supervisory control of a rapid thermal multiprocessor. *IEEE Transactions on Automatic Control*, 38(7):1040–1059, 1993.

[6] Stuart Bennett and D. A. Linkens. *Computer control of industrial processes*. Peter Peregrinus on behalf of the Institution of Electrical Engineers, 1982.

[7] E. Bertens, R. Fabel, M. Petreczky, D. A. van Beek, and J. E. Rooda. Supervisory control synthesis for exception handling in printers. In *Proceedings Philips Conference on Applications of Control Technology*, 2009.

[8] Robert H. Bishop. *Mechatronic Systems, Sensors, and Actuators*. CRC Press, 2 edition, 2008.

[9] B. Brandin and F. Charbonnier. The supervisory control of the automated manufacturing system of the AIP. *Proceedings Fourth International Conference*

on Computer Integrated Manufacturing and Automation Technology, pages 319–324, 1994.

[10] B. A. Brandin. The real-time supervisory control of an experimental manufacturing cell. *IEEE Transactions on Robotics and Automation*, 12(1):1–14, 1996.

[11] N. C. W. M. Braspenning. *Model-Based Integration and Testing of High-Tech Multi-disciplinary Systems*. PhD thesis, Eindhoven University of Technology, 2008.

[12] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. International Series on Discrete Event Dynamic Systems. Springer-Verlag, 1999.

[13] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Springer, New York, 2nd edition, 2007.

[14] Vigyan Chandra, Zhongdong Huang, and Ratnesh Kumar. Automated control synthesis for an assembly line using discrete event system control theory. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 33(2):284–289, 2003.

[15] Yi-Liang Chen and Feng Lin. Modeling of discrete event systems using finite state machines with parameters. In *Proceedings of the IEEE International Conference on Control Applications*, pages 941–946, 2000.

[16] Gde. O. Costa, B. Tortelli, E. A. P. Santos, and M. A. Busetti. Design and implementation of a low cost control system for a manufacturing cell. In *IEEE Conference on Robotics, Automation and Mechatronics*, volume 1, pages 281–286 vol.1, 2004.

[17] M. H. de Queiroz and J. E. R. Cury. Synthesis and implementation of local modular supervisory control for a manufacturing cell. In *Proceedings of the Sixth International Workshop on Discrete Event Systems*, pages 377–382, 2002.

[18] P. Dietrich, R. Malik, W. Wonham, and B. Brandin. Implementation considerations in supervisory control. In *Synthesis and Control of Discrete Event Systems*, pages 185–201. Kluwer, 2002.

[19] M. Fabian and A. Hellgren. PLC-based implementation of supervisory control for discrete event systems. In *Proceedings 37th IEEE Conference on Decision and Control*, pages 3305–3310, Tampa, 1998.

[20] Richard E. Fairley. *Software Engineering Concepts*. McGraw-Hill, New York, 1985.

[21] S. T. J. Forschelen, J. M. van de Mortel-Fronczak, R. Su, and J. E. Rooda. Application of supervisory control theory to theme park vehicles. In *Proceedings of the International Workshop on Discrete Event Systems*, pages 303–309, 2010.

[22] B. Gaudin and P. H. Deussen. Supervisory control on concurrent discrete event systems with variables. In *Proceedings American Control Conference*, pages 4274–4279, 2007.

[23] J. W. P. Geurts. Supervisory control of MRI subsystems. Master's thesis, Eindhoven University of Technology, 2012.

[24] C. H. Golaszewski and P. J. Ramadge. Control of discrete event processes with forced events. In *Proceedings 26th IEEE Conference on Decision and Control*, volume 26, pages 247 – 251, Los Angeles, 1987.

[25] Í. Hasdemir, Salman Kurtulan, and Leyla Gören. An implementation methodology for supervisory control theory. *The International Journal of Advanced Manufacturing Technology*, 36(3):373–385, 2008.

[26] A. Hellgren, M. Fabian, and B. Lennartson. Modular implementation of discrete event systems as sequential function charts applied to an assembly cell. In *Proceedings of the IEEE International Conference on Control Applications*, pages 453–458, Mexico City, 2001.

[27] D. Hendriks. CIF 2 simulator SVG visualization. `http://update.se.wtb.tue.nl/docs/cif2sim_svg_vis.pdf`, 2012.

[28] W. S. Hinshaw, P. A. Bottomley, and G. N. Holland. Radiographic thin-section image of the human wrist by nuclear magnetic resonance. *Nature*, 270:722–723, December 1977.

[29] Jing Huang and Ratnesh Kumar. Optimal nonblocking directed control of discrete event systems. *IEEE Transactions on Automation Science and Engineering*, 5(4):620 – 629, 2008.

[30] Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements engineering*. Springer, 2005.

[31] HYCON 2 consortium. Highly-complex and networked control systems. `http://www.hycon2.eu/`, 2010.

[32] HYCON Network of Excellence. Hybrid control: Taming heterogeneity and complexity of networked embedded systems. `http://www.ist-hycon.org/`, 2005.

[33] Rolf Isermann. *Mechatronic Systems: Fundamentals*. BirkhÃd'user, 2003.

[34] Sangkyun Kim, Jinwoo Park, and Robert C. Leachman. A supervisory control approach for execution control of an FMC. *International Journal of Flexible Manufacturing Systems*, 13(1):5–31, 2001.

[35] J. Kosecka and L. Bogoni. Application of discrete events systems for modeling and controlling robotic agents. In *Proceedings IEEE International Conference on Robotics and Automation*, pages 2557–2562 vol.3, San Diego, 1994.

[36] H. Kwakernaak and R. Sivan. *Linear optimal control systems*. J. Wiley and Sons, New York, 1972.

[37] P. C. Lauterbur. Image Formation by Induced Local Interactions: Examples Employing Nuclear Magnetic Resonance. *Nature*, 242:190–191, March 1973.

[38] S. C. Lauzon, A. K. L. Ma, J. K. Mills, and B. Benhabib. Application of discrete-event-system theory to flexible manufacturing. *IEEE Control Systems Magazine*, 16(1):41–48, 1996.

[39] R. J. Leduc, Pengcheng Dai, and Raoguang Song. Synthesis method for hierarchical interface-based supervisory control. *IEEE Transactions on Automatic Control*, 54(7):1548–1560, 2009.

[40] R. J. Leduc and W. M. Wonham. Discrete event systems modeling and control of a manufacturing testbed. In *Canadian Conference on Electrical and Computer Engineering*, volume 2, pages 793–796 vol.2, Montreal, 1995.

[41] Jing Liu and Houshang Darabi. Ramadge-Wonham supervisory control of mobile robots: lessons from practice. In *Proceedings IEEE International Conference on Robotics and Automation*, volume 1, pages 670–675, Washington, 2002.

[42] Oscar Ljungkrantz, Knut Åkesson, Johan Richardsson, and Kristin Andersson. Implementing a control system framework for automatic generation of manufacturing cell controllers. In *2007 IEEE International Conference on Robotics and Automation*, pages 674–679, Roma, 2007.

[43] Chuan Ma. Stslib. `https://github.com/chuanma/STSLib`, 2011.

[44] Chuan Ma and W. M. Wonham. *Nonblocking supervisory control of state tree structures*, volume 317/2005 of *Lecture Notes in Control and Information Sciences*. Springer, 2005.

[45] Chuan Ma and W. M. Wonham. Nonblocking supervisory control of state tree structures. *IEEE Transactions on Automatic Control*, 51(5):782–793, 2006.

[46] J. Markovski, K. G. M. Jacobs, D. A. van Beek, L. J. A. M. Somers, and J. E. Rooda. Coordination of resources using generalized state-based requirements. In *Proceedings 10th International Workshop on Discrete Event Systems*, pages 300–305, 2010.

[47] J. Markovski, D. A. van Beek, J. C. M. Baeten, and L. J. A. M. Somers. Deliverable D-WP6-4 Implementation and industrial evaluation of the solutions. `http://www.c4c-project.eu/uploads/files/D-WP6-4.pdf`, 2008.

[48] J. Markovski, D. A. van Beek, R. J. M. Theunissen, K. G. M. Jacobs, and J. E. Rooda. A state-based framework for supervisory control synthesis and verification. In *Proceedings 9th IEEE Conference on Decision and Control*, pages 3481–3486, 2010.

[49] M. Moniruzzaman and P. Gohari. Implementing supervisory control maps with PLC. In *Proceedings American Control Conference*, pages 3594–3599, New York, 2007.

[50] MULTIFORM consortium. Integrated multi-formalism tool support for the design of networked embedded control systems MULTIFORM. `http://www.multiform.bci.tu-dortmund.de`, 2008.

[51] D. E. Nadales Agut, D. A. van Beek, and J. E. Rooda. Syntax and semantics of the compositional interchange format for hybrid systems. *Journal of Logic and Algebraic Programming*, 82(1):1–52, 2013.

[52] M. Noorbakhsh and A. Afzalian. Design and PLC based implementation of supervisory control for under-load tap-changing transformers. In *Proceedings International Conference on Control, Automation and Systems*, pages 901–906, Seoul, 2007.

[53] M. Nourelfath and E. Niel. Modular supervisory control of an experimental automated manufacturing system. *Control Engineering Practice*, 12(2):205–216, 2004.

[54] Jean-Fran cois Pétin, David Gouyon, and Gérard Morel. Supervisory synthesis for product-driven automation and its application to a flexible assembly cell. *Control Engineering Practice*, 15(5):595–614, 2007.

[55] C. Potts. Software-engineering research revisited. *Software, IEEE*, 10(5):19–28, 1993.

[56] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, 1987.

[57] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[58] E. A. Santos, M. A. Busetti, and A. D. Vieira. Control synthesis and implementation for an integrated manufacturing system based on supervisory control theory. In *Proceedings IEEE International Conference on Control Applications*, pages 1885–1890, 2006.

[59] R. R. H. Schiffelers, R. J. M. Theunissen, D. A. van Beek, and J. E. Rooda. Model-based engineering of supervisory controllers using CIF. In *Proceedings of the 3rd International Workshop on Multi-Paradigm Modeling*, volume 21 of *Electronic Communications of the EASST*, pages 1–10, Denver, 2009.

[60] Kiam Tian Seow and M. Pasquier. Supervising passenger land-transport systems. *IEEE Transactions on Intelligent Transportation Systems*, 5(3):165–176, 2004.

[61] Daniel B. Silva, Agnelo D. Vieira, and Eduardo F. R. Loures. Dealing with routing in an automated manufacturing cell: a supervisory control theory application. *International Journal of Production Research*, 49(16):4979–4998, 2011.

[62] Markus Sköldstam, Knut Åkesson, and Martin Fabian. Supervisory control applied to automata extended with variables - revised. Technical Report R001/2008, Chalmers University of Technology, Sweden, 2008.

[63] R. Su, D. A. van Beek, and J. E. Rooda. Deliverable D2.2.1 Report on methods and prototype tool for the supervisory control of complex systems. Technical report, MULTIFORM consortium, 2010.

[64] Rong Su, J. H. van Schuppen, and J. E. Rooda. Aggregative synthesis of distributed supervisors based on automaton abstraction. *IEEE Transactions on Automatic Control*, 55(7):1627–1640, 2010.

[65] Supremica. The official website for the supremica project, 2010.

[66] Systems Engineering Group TU Eindhoven. http://se.wtb.tue.nl/sewiki/supcon/susyna, 2012.

[67] Systems Engineering Group TU/e. Overview of supervisory control tools. `http://se.wtb.tue.nl/sewiki/supcon/cif_svc_overview`, 2011.

[68] Systems Engineering Group TU/e. SCIDE graphical editor. `http://se.wtb.tue.nl/sewiki/supcon/scide/start`, 2011.

[69] Systems Engineering Group TU/e. CIF toolset. `http://cif.se.wtb.tue.nl`, 2013.

[70] Seiichiro Tani, Kiyoharu Hamaguchi, and Shuzo Yajima. The complexity of the optimal variable ordering problems of shared binary decision diagrams. In K. W. Ng, P. Raghavan, N. V. Balasubramanian, and F. Y. L. Chin, editors, *Algorithms and Computation*, number 762 in Lecture Notes in Computer Science, pages 389–398. Springer-Verlag, January 1993.

[71] R. J. M. Theunissen, R. R. H. Schiffelers, D. A. van Beek, and J. E. Rooda. Supervisory control synthesis for a patient support system. In *Proceedings of the European control conference*, pages 4647–4652, Budapest, Hungary, 2009.

[72] R. J. M. Theunissen, D. A. van Beek, and J. E. Rooda. Improving evolvability of a patient communication control system using state-based supervisory control synthesis. *Advanced Engineering Informatics*, 26(3):502–515, 2012.

[73] Rolf J. M. Theunissen, M. Petreczky, Ramon R. H. Schiffelers, Dirk A. van Beek, and Jacobus E. Rooda. Application of supervisory control synthesis to a patient support table of a magnetic resonance imaging scanner. *IEEE T. Automation Science and Engineering*, pages 20–32, 2014.

[74] D. A. van Beek, P. Collins, D. E. Nadales Agut, J. E. Rooda, and R. R. H. Schiffelers. New concepts in the abstract format of the Compositional Interchange Format. In A. Giua, C. Mahuela, M. Silva, and J. Zaytoon, editors, *3rd IFAC Conference on Analysis and Design of Hybrid Systems*, pages 250–255, Zaragoza, 2009.

[75] D. A. van Beek, P. J. L. Cuijpers, J. Markovski, D. E. Nadales Agut, and J. E. Rooda. Reconciling urgency and variable abstraction in a hybrid compositional setting. In Krishnendu Chatterjee and Thomas Henzinger, editors, *Formal Modeling and Analysis of Timed Systems*, volume 6246 of *LNCS*, pages 47–61. Springer, 2010.

[76] Piërre van de Laar and Teade Punter. *Views on Evolvability of Embedded Systems*. Embedded Systems Institute. Springer, 2010.

[77] W3C. W3C SVG working group. `http://www.w3.org/Graphics/SVG/`, 2011.

[78] Dominik Weishaupt, Victor D. Köchli, and Borut Marincek. *How does MRI work? an introduction to the physics and function of magnetic resonance imaging*. Springer Berlin Heidelberg, 2006.

[79] W. M. Wonham. *Supervisory control of discrete-event systems*. Dept. Elect. Comput. Eng., Univ. Toronto, Toronto, ON, Canada, 2012.

[80] W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal on Control and Optimization*, 25(3):637–659, 1987.

[81] Y. Yang and R. Gohari. Embedded supervisory control of discrete-event systems. In *Proceedings IEEE International Conference on Automation Science and Engineering*, pages 410–415, 2005.

Curriculum vitae

Rolf Theunissen was born on January 19th, 1980 in Nuland, the Netherlands. In 1998, he finished VWO at the Mondriaan College in Oss. From 1998 to 2006 he studied Mechanical Engineering at the Eindhoven University of Technology, the Netherlands. In the Systems Engineering Group, he performed his graduation project on the topic 'Process algebraic linearization of hybrid Chi'. After graduation in 2006, he started his Ph.D. project in the same group on the topic 'Supervisory Control in Health Care Systems'. This Ph.D. project was part of the DARWIN project on evolvabily, carried out in close cooperation with Philips Healthcare, the Embedded Systems Institute, and other industrial and academical partners. In 2011, Rolf joined Nspyre, where he currently works on control system design via Domain Specific Languages for ASML.

Titles in the IPA Dissertation Series since 2009

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty

of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Morali. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations*. Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice*. Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars*. Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns*. Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. *Spiking Neural P Systems*. Faculty of Mathematics and Natural Sciences, UL. 2011-25

A. Khosravi. *Optimal Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2012-01

A. Middelkoop. *Inference of Program Properties with Attribute Grammars, Revisited*. Faculty of Science, UU. 2012-02

Z. Hemel. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

T. Dimkov. *Alignment of Organizational Security Policies: Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

S. Sedghi. *Towards Provably Secure Efficiently Searchable Encryption*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference*. Faculty of Science, Mathematics and Computer Science, RU. 2012-06

K. Verbeek. *Algorithms for Cartographic Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2012-07

D.E. Nadales Agut. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation*. Faculty of Mechanical Engineering, TU/e. 2012-08

H. Rahmani. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2012-09

S.D. Vermolen. *Software Language Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

L.J.P. Engelen. *From Napkin Sketches to Reliable Software*. Faculty of Mathematics and Computer Science, TU/e. 2012-11

F.P.M. Stappers. *Bridging Formal Models – An Engineering Perspective*. Faculty of Mathematics and Computer Science, TU/e. 2012-12

W. Heijstek. *Software Architecture Design in Global and Model-Centric Software Development*. Faculty of Mathematics and Natural Sciences, UL. 2012-13

C. Kop. *Higher Order Termination*. Faculty of Sciences, Department of Computer Science, VUA. 2012-14

A. Osaiweran. *Formal Development of Control Software in the Medical Systems Domain*. Faculty of Mathematics and Computer Science, TU/e. 2012-15

W. Kuijper. *Compositional Synthesis of Safety Controllers*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems*. Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor*. Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins*. Faculty of Mathematics and Computer Science, TU/e. 2013-09

S. van der Burg. *A Reference Architecture for Distributed Software Deployment*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

J.J.A. Keiren. *Advanced Reduction Techniques for Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2013-11

D.H.P. Gerrits. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points*. Faculty of Mathematics and Computer Science, TU/e. 2013-12

M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

M.J.M. Roeloffzen. *Kinetic Data Structures in the Black-Box Model*. Faculty of Mathematics and Computer Science, TU/e. 2013-14

L. Lensink. *Applying Formal Methods in Software Development*. Faculty of Science, Mathematics and Computer Science, RU. 2013-15

C. Tankink. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants*. Faculty of Science, Mathematics and Computer Science, RU. 2013-16

C. de Gouw. *Combining Monitoring with Run-time Assertion Checking*. Faculty of Mathematics and Natural Sciences, UL. 2013-17

J. van den Bos. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics*. Faculty of Science, UvA. 2014-01

D. Hadziosmanovic. *The Process Matters: Cyber Security in Industrial Control Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

A.J.P. Jeckmans. *Cryptographically-Enhanced Privacy for Recommender Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

C.-P. Bezemer. *Performance Optimization of Multi-Tenant Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

T.M. Ngo. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

A.W. Laarman. *Scalable Multi-Core Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

J. Winter. *Coalgebraic Characterizations of Automata-Theoretic Classes*. Faculty of Science, Mathematics and Computer Science, RU. 2014-07

W. Meulemans. *Similarity Measures and Algorithms for Cartographic Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2014-08

A.F.E. Belinfante. *JTorX: Exploring Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

A.P. van der Meer. *Domain Specific Languages and their Type Systems*. Faculty of Mathematics and Computer Science, TU/e. 2014-10

B.N. Vasilescu. *Social Aspects of Collaboration in Online Software Communities*. Faculty of Mathematics and Computer Science, TU/e. 2014-11

F.D. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2014-12

N. Noroozi. *Improving Input-Output Conformance Testing Theories*. Faculty of Mathematics and Computer Science, TU/e. 2014-13

M. Helvensteijn. *Abstract Delta Modeling: Software Product Lines and Beyond*. Faculty of Mathematics and Natural Sciences, UL. 2014-14

P. Vullers. *Efficient Implementations of Attribute-based Credentials on Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2014-15

F.W. Takes. *Algorithms for Analyzing and Mining Real-World Graphs*. Faculty of Mathematics and Natural Sciences, UL. 2014-16

M.P. Schraagen. *Aspects of Record Linkage*. Faculty of Mathematics and Natural Sciences, UL. 2014-17

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World*. Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction*. Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems*. Faculty of Mechanical Engineering, TU/e. 2015-03